INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA CENTRAL
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LENNOARD RAÍ LOPES E SILVA

SNAPPIER - BIBLIOTECA MULTIPLATAFORMA PARA A CRIAÇÃO DE APLICATIVOS SEGUINDO O CONCEITO SERVER-DRIVEN UI

TERESINA
2025

LENNOARD RAÍ LOPES E SILVA

SNAPPIER - BIBLIOTECA MULTIPLATAFORMA PARA A CRIAÇÃO DE APLICATIVOS SEGUINDO O CONCEITO SERVER-DRIVEN UI

Relatório Técnico de Software apresentado ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí – IFPI, Campus Teresina Central, como requisito para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Prof. Me. Rogério da Silva

TERESINA
2025

LENNOARD RAÍ LOPES E SILVA

SNAPPIER - BIBLIOTECA MULTIPLATAFORMA PARA A CRIAÇÃO DE APLICATIVOS SEGUINDO O CONCEITO SERVER-DRIVEN UI

Relatório Técnico de Software apresentado ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí – IFPI, Campus Teresina Central, como requisito para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Aprovado em 12 de Fevereiro de 2025.

BANCA EXAMINADORA

Prof. Me. Rogério da Silva (Orientador)
Instituto Federal do Piauí (IFPI)

Prof. Me. Ely da Silva Miranda
Instituto Federal do Piauí (IFPI)

Prof. Dr. Otilio Paulo da Silva Neto
Instituto Federal do Piauí (IFPI)

Este trabalho é dedicado a todos que me inspiraram ou forneceram conhecimento para que o mesmo pudesse ser concluído.

AGRADECIMENTOS

Agradeço aos meus pais, Francisco das Chagas Lopes e Silva e Maryluce Barbosa e Silva, por mostrar a importância do ensino durante toda a minha vida e pelos incentivos a buscar e continuar nesta graduação. Sem este apoio, não teria o conhecimento nem a força de espírito necessária para concretizar esta conquista.

Ao Instituto Federal do Piauí, deixo meus agradecimentos pela oportunidade de poder fazer parte da instituição como aluno.

Ao meu orientador, Prof. Me. Rogério da Silva e ao Prof. Me. Osires Pires Coelho Filho, deixo meus agradecimentos por introduzirem e iniciarem a minha jornada de formação profissional em análise e desenvolvimento de sistemas com o estudo dos algoritmos. Agradeço também por me mostrarem, em várias oportunidades, amostras de aplicações no "mundo real", algo pelo qual sou especialmente grato.

Ao Prof. Me. Ely da Silva Miranda, agradeço os conhecimentos, conceitos e convenções importantes no mundo de desenvolvimento de software a mim ensinados durante a minha passagem pelo Instituto.

Ao Prof. Dr. Otilio Paulo da Silva Neto, agradeço pelos conhecimentos passados em arquitetura de sistemas e as apresentações gerais das novidades tecnológicas em desenvolvimento de software.

Ao meu líder técnico, Juarez Pereira de Oliveira Neto, agradeço pelo acolhimento, lapidação, parceria e mentorias realizadas na empresa onde estou atualmente empregado.

Por fim, agradeço também a todos os professores do Campus Teresina Central do IFPI, em especial, do curso Análise e Desenvolvimento de Sistemas, pelos conhecimentos, orientações e sugestões compartilhadas ao longo desses anos na instituição.

Gather ye rosebuds while ye may,
Old time is still a-flying;
And this same flower that smiles today,
Tomorrow will be dying.
Robert Herrick

RESUMO

O atraso na entrega de novos recursos e atualizações na interface de usuário de aplicativos, devido a ciclos de desenvolvimento longos e burocráticos, é cada vez mais comum, especialmente ao distribuir em várias plataformas com políticas diversas. Esse desafio é intensificado pela complexidade dos aplicativos modernos. Os métodos tradicionais geralmente envolvem um tempo significativo em processos de codificação, teste e aprovação, o que dificulta a capacidade de responder rapidamente ao *feedback* do usuário e às pressões competitivas. Reconhecendo essa realidade, tecnologias emergentes como a interface de usuário voltada ao servidor e o desenvolvimento multiplataforma oferecem caminhos promissores para uma entrega mais ágil e modificações em tempo real ao evitarem a burocracia das lojas de aplicativos e dos canais de distribuição. Este trabalho apresenta uma biblioteca autoral projetada para facilitar o desenvolvimento de aplicativos multiplataforma, utilizando os princípios do Server-Driven UI e a tecnologia do Kotlin Multiplatform. A biblioteca proposta visa acelerar o desenvolvimento e a distribuição de aplicativos e permitir atualizações da interface de usuário em tempo real. Em avaliações de desempenho, abordando uma preocupação comum no desenvolvimento multiplataforma, a biblioteca demonstrou uma melhoria de 26,67% no tempo médio de inicialização quando utilizada em um aplicativo, em comparação com soluções similares desenvolvidas com Flutter.

Palavras-chaves: server-driven UI; multiplataforma; biblioteca; aplicativo.

ABSTRACT

Delays in delivering new features and user interface updates in applications, due to long and complex development cycles, is increasingly common, especially when distributing across multiple platforms with varying policies. This challenge is intensified by the complexity of modern applications. Traditional methods typically involve significant time in coding, testing, and approval processes, which hampers the ability to respond quickly to user feedback and competitive pressures. Recognizing this, emerging technologies like Server-Driven UI and multiplatform development offer promising avenues for faster product delivery and real-time modifications by bypassing the administrative hurdles of app stores and distribution channels. This paper presents an original library designed to facilitate multiplatform application development using Server-Driven UI principles and the Kotlin Multiplatform technology. The proposed library aims to accelerate application development and distribution, enabling real-time user interface updates. In performance evaluations, addressing a common concern in multiplatform development, the library demonstrated a 26.67% improvement in average startup time when used in an application, compared to similar solutions developed with Flutter.

Keywords: server-driven UI; multiplatform; library; application.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AOT	Ahead-of-time
API	Application Programming Interface
BDC	Backend Driven Content
DIP	Dependency Inversion Principle
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IFPI	Instituto Federal de Educação, Ciência e Tecnologia do Piauí
JDK	Java Development Kit
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
KMM	Kotlin Multiplatform Mobile
KMP	Kotlin Multiplatform
MIT	Massachusetts Institute of Technology
OCP	Open-Closed Principle
OO	Object-Oriented
REST	Representational State Transfer
SDK	Software Development Kit
SDUI	Server-Driven UI
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

LISTA DE SÍMBOLOS

%	Porcentagem
®	Marca registrada
™	Marca comercial

SUMÁRIO

1 INTRODUÇÃO

No mundo digital atual, o uso de aplicativos e programas de computador é cada vez mais comum e acessível, no entanto, criar e manter experiências de usuário eficazes representa um grande desafio. Isso ocorre devido à rápida evolução das necessidades dos usuários, à competição intensa (CHERNONOG, 2024) e às dificuldades específicas de manutenção impostas pelas tecnologias e plataformas (WANG *et al.*, 2018).

O desenvolvimento de aplicativos tradicionais geralmente envolve um ciclo longo e complexo de codificação, teste e distribuição. Esse processo pode atrasar significativamente a entrega de novos recursos aos clientes. A situação é ainda mais crítica quando mudanças na interface exigem urgência para responder ao *feedback* dos usuários, atender a exigências legais ou manter a competitividade no mercado.

Uma inovação positiva para atender essa demanda, com ênfase na distribuição e apresentação de novos aplicativos para as massas de usuários, foi a introdução de Lojas ou Repositórios de aplicativos *online*. O primeiro registro desses repositórios aponta o surgimento no final da década de 90, sendo entregue através do SUSE, uma distribuição Linux (LISBOA, 2021). A prática tornou-se popular e permanece ativa como funcionalidade de várias distribuições Linux até os dias de hoje, além de ter rompido as barreiras desse sistema operacional e influenciado a criação de várias Lojas de Aplicativos diferentes em outros tipos de sistemas.

Com as atuais ameaças digitais, na forma de *malware*, a moderação e a aplicação de políticas de segurança nas lojas de aplicativos tornaram-se extremamente necessárias e, como esperado, rigorosas. Essa moderação estabelece uma rotina de verificação de mau comportamento e controle de qualidade mínima para esses aplicativos, que, em alguns casos, pode demorar dias ou até mesmo semanas (IBRAHIM, 2024). Durante esse período, a publicação e atualização dos aplicativos são impossíveis de serem realizadas. Essa demora é particularmente frustrante quando uma simples atualização em elementos da interface de usuário, ou *User Interface* (UI), precisa passar pelo mesmo processo, independentemente da complexidade das mudanças. Os efeitos são multiplicados quando há a necessidade de distribuir os aplicativos em várias plataformas, cada uma com suas próprias políticas.

Aplicativos atuando tanto no cenário nacional quanto internacional, como iFood, Duolingo, Nubank e Airbnb, começaram a adotar uma tecnologia capaz de atualizar a interface de usuário sob demanda, sem precisar da intervenção do usuário ou implantações em lojas de aplicativos. Essa tecnologia é conhecida como *Server-Driven UI* (SDUI), e tem esse nome pois a responsabilidade de definir o conteúdo visualizado pelo usuário foi transferida para o servidor, permitindo atualizações dinâmicas e mais rápidas, sem a necessidade de intervenção do usuário.

Rodrigo Sampaio, Escritor Técnico Sênior no Nubank, comenta que o time de engenharia de software do banco está gradualmente ampliando a utilização desse tipo de tecnologia para tornar as funcionalidades do aplicativo altamente personalizáveis, permitindo que decisões de produto sejam tomadas no mesmo dia:

> BDC internal use cases keep on growing, and over time we have more teams interested in building new screens and flows using it. To keep track on that path, we are continuously investing to make Backend Driven Content more robust and efficient to meet more requirements from different teams and use cases. (SAMPAIO, 2023)

Já Michele Zhu, autora no Blog do Duolingo, comenta que o SDUI revolucionou o desenvolvimento do aplicativo do Duolingo ao permitir atualizações mais rápidas, experiências de usuário consistentes, experimentos simplificados e lançamentos de funcionalidades em todas as versões do aplicativo. Isso foi concretizado ao mesmo tempo em que a necessidade de implementações separadas para Android e iOS foi reduzida, economizando tempo e recursos.

> In a world where app updates can be a lengthy process, Duolingo's new usage of server-driven UI (SDUI) is a game-changer. By handling UI from the backend, we've sped up development and opened doors for quick experiments and consistent user experiences. (ZHU, 2024)

Considerando o cenário em que a rápida evolução das necessidades do usuário e a intensa competitividade de mercado desafiam as equipes de desenvolvimento, observa-se a necessidade de soluções que agilizem o processo de criação e atualização de interfaces de usuário que estejam disponíveis para a comunidade desenvolvedora de aplicativos em geral. A complexidade adicional imposta pelas políticas rigorosas das lojas de aplicativos e a diversidade de plataformas reforçam a demanda por métodos mais eficientes.

Nesse contexto, o desenvolvimento de uma biblioteca multiplataforma, capaz de auxiliar na criação e renderização de interfaces dinâmicas, apresenta-se como uma solução relevante. O uso do Server-Driven UI permite transferir a configuração da interface dos aplicativos para o servidor, abordando desafios como a rapidez no desenvolvimento, a entrega ágil de ideias dos times de produto e a personalização e segmentação em escala.

Com isso, o Objetivo Geral deste trabalho é desenvolver e distribuir uma biblioteca chamada "Snappier", projetada para facilitar a criação e atualização sob demanda de interfaces de usuário em múltiplas plataformas. Para oferecer essa funcionalidade, a biblioteca utiliza o conceito de Server-Driven UI, para transferir a responsabilidade da configuração da interface dos aplicativos para o servidor. Com isso, a abordagem atende a parâmetros críticos enfrentados por equipes de desenvolvimento de software, como rapidez de desenvolvimento, entrega das ideias dos times de produto, personalização, segmentação e escalabilidade. Possíveis desafios para os desenvolvedores introduzidos pela abordagem multiplataforma como um todo são importantes, porém ficarão de fora do escopo deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentadas as novidades tecnológicas e os aspectos inerentes do desenvolvimento multiplataforma além das outras tecnologias utilizadas para a construção do Snappier, da mesma forma que descreve o funcionamento e a utilização dessas tecnologias pela biblioteca.

2.1 INOVAÇÕES E TENDÊNCIAS TECNOLÓGICAS

No cenário atual de desenvolvimento de aplicativos, cada vez mais as soluções multiplataforma viram tendência, dadas as suas vantagens como código base unificado, facilidade de iniciação e diminuição do tempo de desenvolvimento (ROGERS; GRATCH, 2022). Neste cenário, dois jogadores atualmente se destacam com posições sólidas no mercado: Flutter e React, com React Native.

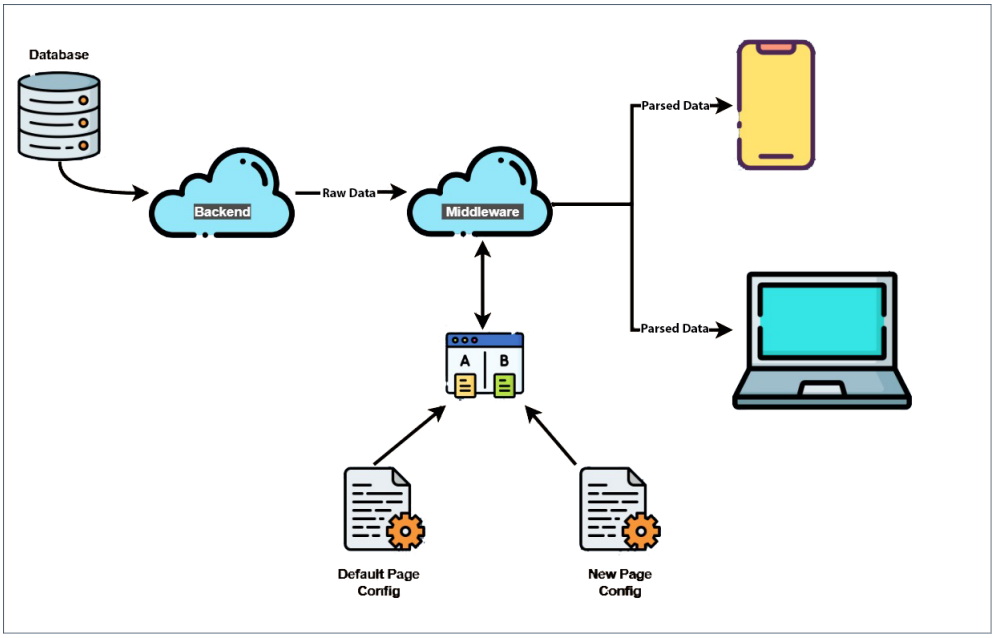
Esses *frameworks* permitem o desenvolvimento de aplicações com criação de UI rápida, consistente e personalizada entre as plataformas, porém, não se destacam tanto se a regra de negócio precisar ser compartilhada. Neste momento, um outro jogador propõe-se com alternativas, o Kotlin Multiplatform, ou KMP, utilizando a linguagem de programação Kotlin para o compartilhamento da regra de negócio entre as plataformas, garantindo performance próxima à nativa, com grande suporte de empresas como a JetBrains e Google, além de uma comunidade de desenvolvedores vasta, com grande herança de ecossistemas Android.

Em conjunto com o KMP, o Compose Multiplatform representa um passo significativo em direção ao desenvolvimento de interfaces de usuário e regras de negócio verdadeiramente unificadas entre várias plataformas, permitindo o compartilhamento e reúso dos componentes de UI e garantindo a aparência consistente, independentemente do dispositivo utilizado.

2.1.1 Server-Driven UI

Server-Driven UI (SDUI) — "Interface de usuário orientada ao servidor" — em desenvolvimento de software é um conceito cujo objetivo é tornar o servidor o regente completo, ou tão quanto possível, do conteúdo visualizado pela aplicação do lado do cliente (SAMPAIO, 2023). Em outras palavras, os dados e as instruções sobre como uma tela em particular deve ser apresentada são pedidos pelo cliente ao servidor, ao invés de serem criados localmente, o que permite que atualizações de layout, estrutura de navegação e até mesmo conteúdo sejam aplicadas dinamicamente, conforme o exemplo da Figura 1:

Figura 1 – Exemplo de arquitetura de um sistema Server-Driven UI



Fonte: Build a Server Driven UI (SARKAR, 2023) (adaptado)

A autonomia do servidor em cenários desse tipo é particularmente útil em ambientes onde as interfaces precisam ser constantemente personalizadas para diferentes grupos de usuários ou onde a manutenção da consistência e fidelidade visual e de conteúdo em múltiplas plataformas é essencial. Esta abordagem traz vários benefícios em comparação com a forma tradicional de desenvolvimento de software, incluindo:

a) Menor dependência dos usuários e das lojas de aplicativos para atualização: é razoável que a demora na revisão dos aplicativos na loja seja necessária para garantir segurança e barrar aplicativos maliciosos (RAHMAN *et al.*, 2016), mas por vezes atualizações apenas incluem pequenos consertos ou alterações e são punidas com o mesmo tempo de espera;

b) Maior controle sobre o conteúdo exibido: ter a possibilidade de fornecer uma única fonte consumida pelos aplicativos para a apresentação do conteúdo aproxima e empodera o time de produto e de negócios do artefato final que será apresentado ao cliente. Michelle Zhu também comenta sobre isso no Blog do Duolingo, destacando a facilidade do sistema criado em lidar com designs criativos e cada parte individual deles:

> While many companies have built SDUI systems, Duolingo's is uniquely flexible to support the app's creative designs. The system's granularity allows control over everything—from UI layout to interactivity—via a single backend response. (ZHU, 2024);

c) Evita vazamento de funcionalidades novas: aplicativos de grande relevância e ciclo de desenvolvimento e distribuição bem definidos podem ter a necessidade de utilizar a táctica de *feature-flags*, quando uma funcionalidade nova do aplicativo é empacotada e entregue aos clientes finais, podendo ser ativada ou desativada remotamente (SEITZ, 2022). Por haver código dessa funcionalidade no artefato final entregue ao usuário, pode ser feito trabalho de engenharia reversa para obter detalhes da mesma;

d) Segmentação de usuários: a exibição de conteúdos personalizados a usuários com características diferentes tem o potencial de gerar maior satisfação e aumentar a retenção. Steven Shugan, especialista em modelos de marketing, compara esta técnica ao conceito de lealdade e relacionamento:

> Many industries from Internet retailing to high-end personal services have the capability to customize their service to individual customers. [...] Customizing services to meet the needs of individual customers is probably closer to the concept of loyalty and a relationship (SHUGAN, 2005, p. 189);

e) Simplifica a execução de testes A/B. Testes A/B são uma forma de comparar duas versões de uma funcionalidade de um produto lançado para determinar qual tem melhor desempenho, baseando-se em dados e métricas específicas. Assumindo que a segmentação de usuários pode entregar conteúdos diferentes para usuários diferentes, a criação de métricas para cada uma dessas variações torna-se trivial dentro de um ambiente Server-Driven, tornando possível identificar qual variante tem melhor desempenho.

2.1.2 Desenvolvimento multiplataforma

O desenvolvimento multiplataforma está relacionado ao processo de criação e desenvolvimento de sistemas que podem ser executados em múltiplos sistemas operacionais, além de plataformas web (VANZIN; BONIATI; SILVA, 2012). Esta abordagem permite aos desenvolvedores reduzir o tempo de desenvolvimento ao diminuir a necessidade de criar uma base de código específica para cada plataforma (RONKAINEN *et al.*, 2013), além de garantir um público maior, com a possibilidade de distribuição em várias plataformas simultaneamente e simplificar a manutenção e atualização do software, já que as mudanças no código são refletidas em todas as plataformas ao mesmo tempo.

Há também desafios introduzidos com a abordagem multiplataforma, em especial a otimização da performance e a implementação de funções específicas a cada plataforma. O código compartilhado nem sempre será suficiente para todas as implementações, necessitando de fragmentos de código específicos, que podem introduzir complexidades e necessidades de expertises não previstas (CORRAL; JANES; REMENCIUS, 2012).

2.2 FERRAMENTAS E TECNOLOGIAS

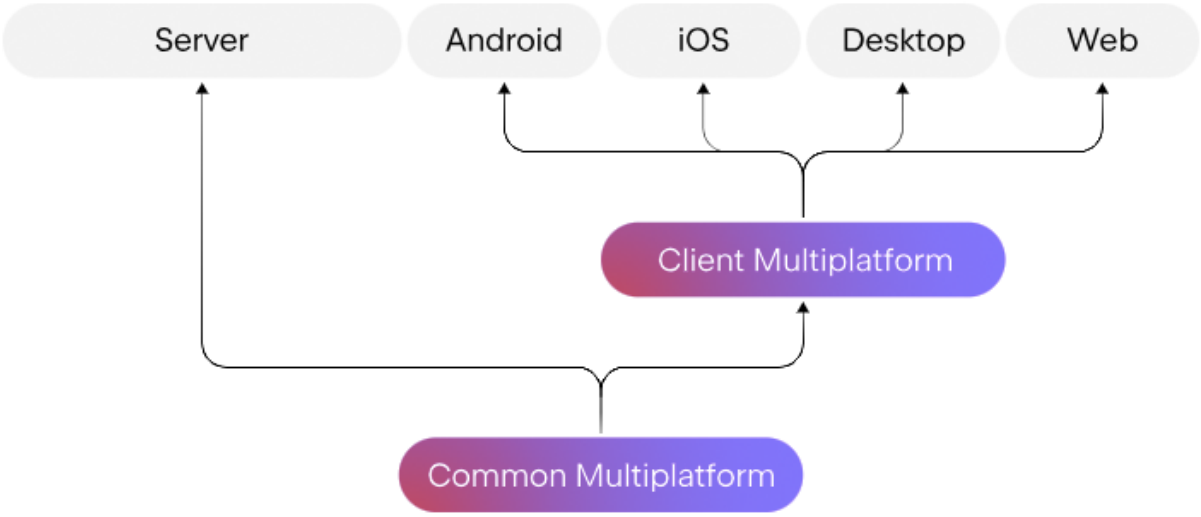
A biblioteca Snappier, desenvolvida neste trabalho, aproveita os princípios do SDUI, onde o servidor dita o layout e o comportamento da UI dinamicamente. Essa forma de desenvolvimento é ideal para aplicativos que exigem atualizações frequentes da UI sem precisar lançar novas versões dos aplicativos. A arquitetura é multiplataforma, utilizando o Kotlin Multiplatform para dar suporte ao Android, iOS, Desktop via JRE e potencialmente outras mais, com renderização da UI feita através do Compose Multiplatform (JETBRAINS, 2023).

2.2.1 Kotlin Multiplatform

Kotlin Multiplatform (KMP), anteriormente conhecido como Kotlin Multiplatform Mobile (KMM) é um componente da Linguagem de Programação Kotlin que permite o compartilhamento de código comum entre diferentes plataformas. O KMP possui suporte à compilação para Android, iOS, Web e Desktop (Windows, Linux e macOS), promovendo a redução do esforço e do tempo de desenvolvimento necessário para o suporte individual de cada uma delas (POLYAKOV, 2020).

Esta tecnologia baseia-se no conceito de *código comum* e *código específico*. No código comum, é escrita a regra de negócio, os modelos de dados e componentes cruciais que serão compartilhados entre as plataformas escolhidas utilizando Kotlin. Já o código específico é utilizado para interações especiais com o sistema operacional de destino, como chamadas de APIs nativas ou a criação da interface do usuário (PAVLOV, 2021). A Figura 2 ilustra esse conceito:

Figura 2 – Estratégia de funcionamento do Kotlin Multiplatform



Fonte: Kotlin Multiplatform | Kotlin Documentation (POLYAKOV, 2020)

Uma das técnicas utilizadas para conseguir atingir o funcionamento multiplataforma com Kotlin é o mecanismo *expect/actual* (PAVLOV, 2023). Isto permite que os desenvolvedores definam declarações abstratas no módulo de código comum que "esperam" implementações concretas dos módulos de plataformas específicas, permitindo assim o gerenciamento de APIs que são distintas em plataformas diferentes, como, por exemplo, o acesso ao sistema de arquivos.

2.2.2 Compose Multiplatform

Compose Multiplatform é um framework multiplataforma fundado no KMP para a criação de interfaces de usuário baseado no Jetpack Compose (JETBRAINS, 2023), uma ferramenta para construção de UI nativa recomendada pelo Google para o desenvolvimento de aplicativos Android, criada utilizando a linguagem de programação Kotlin. Diferentemente do framework de UI tradicional do Android, que utiliza arquivos XML para o design do layout, o Jetpack Compose usa uma abordagem declarativa; dessa forma, em vez de descrever como construir uma interface com instruções passo a passo, declara-se como ela deverá se apresentar, levando em consideração um estado específico, que é mutável e observável. Quando este estado muda, o framework automaticamente atualiza a UI refletindo as mudanças.

A JetBrains S.R.O. desenvolveu o Compose Multiplatform com o objetivo de compartilhar as interfaces de usuário criadas utilizando Jetpack Compose com outras plataformas, agilizando a criação de UIs ao reutilizar as habilidades consolidadas vistas no desenvolvimento de aplicativos para Android.

O Jetpack Compose em si funciona com seu próprio *runtime*, responsável por gerenciar e renderizar o estado da interface de usuário. Esse runtime utiliza um mecanismo sofisticado de recomposição em que a UI é apenas atualizada quando o seu estado muda (GOOGLE, 2024), evitando chamadas de renderização desnecessárias em todos os objetos na tela. Isto é possível utilizando técnicas para literalmente "lembrar" o estado dos componentes visuais durante as recomposições, prevenindo a sua recriação desnecessária.

As ferramentas de UI do Compose Multiplatform fornecem abstrações sobre os frameworks específicos das outras plataformas com as quais têm compatibilidade, por exemplo, o próprio Jetpack Compose no Android ou Skia no Desktop. Dessa forma, ao utilizá-lo, apenas uma única API unificada para criação de interfaces gráficas existe, e essa API, caso necessário, permite implementações específicas e nativas a cada plataforma.

2.3 TRABALHOS RELACIONADOS

Embora exista pouca literatura acadêmica sobre o tema deste trabalho, há projetos notáveis atuando no mercado atual. Esta seção pretende identificar e comparar alguns desses projetos, destacando as vantagens, desvantagens e inovações, com a esperança de fazê-lo de forma justa e em boa-fé.

2.3.1 Beagle

Beagle (SOUZA, 2020) é um framework de implementação do conceito *Backend for Frontend*, termo sinônimo de Server-Driven UI, de código aberto disponível para Android, iOS e Web, publicado pela empresa brasileira ZUP em 2020 para ajudar as empresas a reduzirem a complexidade associada à atualização de interfaces de usuário em aplicativos do lado do cliente. Este framework oferece como vantagens as especificações de componentes abundantes; isso significa que o backend pode extensivamente definir a aparência e o posicionamento do componente visual, além de possuir uma documentação excelente, somado ao fato de ser um software de código aberto.

Em contrapartida, o Beagle apenas permite a requisição de "telas" inteiras, sendo assim não é possível requisitar apenas fragmentos, caso seja necessário. Outro ponto é que só é possível utilizar APIs do tipo REST, deixando de lado outros tipos de tecnologias ou utilizar JSON direta ou localmente, além de necessitar de implementação específica e nativa para cada plataforma.

Snappier permite a renderização de componentes individuais, seções ou telas inteiras, tornando possível a renderização híbrida entre componentes nativos e Server-Driven ao mesmo tempo. Além disso, há a possibilidade de utilização de qualquer tipo de fonte de dados, sendo ela remota ou local. Snappier também dá suporte a qualquer tipo de representação de dados, desde que sigam o protocolo estabelecido pela biblioteca.

2.3.2 Judo

O Judo (COOMBS, 2021) é uma tecnologia criada pela empresa Rover Labs para o desenho de interfaces de usuário para iOS que utiliza o conceito *"no-code"*, uma tecnologia emergente facilitadora da criação de aplicativos com o mínimo possível de código escrito, onde, ao invés de criar código para replicar a interface de usuário, cria-se uma UI através de uma ferramenta capaz de gerar código. Isto aumenta o envolvimento de outros profissionais na criação do software em si, não necessariamente do campo da ciência da computação (ROKIS; KIRIKOVA, 2022).

Neste produto destacam-se a facilidade de criação de interfaces de usuário e a exportação de código nativo, preparado para o novo kit de renderização de interfaces

da Apple, o SwiftUI. Judo também permite a conexão de APIs REST ou GraphQL, facilitando a incorporação de dados dinâmicos na interface de usuário do aplicativo e permitindo o desenvolvimento de experiências de usuários baseadas em dados. Entretanto, é fácil notar a falta de compatibilidade com outras plataformas, a necessidade de compilação após a geração de código e a precificação individual cobrada mensalmente.

Mesmo que o Judo seja mais apropriado para *designers* ou times com experiência em programação de computadores limitada, a abordagem *no-code* traz consigo uma limitação dos designs de UI, além de a personalização ser limitada a apenas aquilo que o SwiftUI oferece. A biblioteca Snappier suporta uma quantidade maior de plataformas, é grátis e de código aberto, permitindo que os desenvolvedores façam modificações para atender aos seus requisitos específicos e atingir um público maior.

2.3.3 Tabela comparativa

Tabela 1 – Tabela comparativa entre Beagle, Judo e Snappier

	Beagle	**Judo**	**Snappier**
Desenvolvedor	Zup Innovation	Rover Labs Inc.	Lennoard Silva
Plataformas	Android, iOS, Web	macOS, iOS	Android, iOS, Web, Desktop
Código aberto	Sim	Não	Sim
Precificação	Grátis	Pago	Grátis
Integração	Server-Driven UI	No-Code	Server-Driven UI
Fonte de dados	APIs REST	APIs REST e GraphQL	Qualquer fonte
Flexibilidade	Telas inteiras	Componentes individuais	Componentes individuais, seções ou telas inteiras
Licença	Apache License 2.0	Conferir os Termos de Serviço (proprietário)	MIT License

Fonte: Próprio autor (2025)

Analisando a Tabela 1, pode-se inferir que cada solução apresenta vantagens e limitações que podem influenciar sua adoção conforme as necessidades específicas de um projeto. O Beagle possui suporte a componentes predefinidos, ampla documentação e facilita a implementação de interfaces dinâmicas, mas restringe a granularidade das requisições e a flexibilidade no consumo de dados. O Judo, por sua vez, foca na abordagem no-code, tornando a criação de interfaces mais acessível para profissionais fora da área de desenvolvimento, embora limite a personalização e exija compilação para a aplicação das mudanças. A integração forte com ecossistemas da Apple é um ponto forte do Judo, porém a falta de suporte a outros sistemas e a precificação devem ser levadas em consideração. Já a biblioteca Snappier busca um equilíbrio entre flexibilidade e compatibilidade multiplataforma, permitindo a renderização híbrida de componentes e oferecendo suporte a diferentes formatos de dados.

3 METODOLOGIA

Este capítulo contém as técnicas, conceitos tecnológicos, especialidades e conhecimentos utilizados para a construção da biblioteca, bem como uma visão geral da organização, funcionamento e arquitetura utilizada.

3.1 TÉCNICAS DE ENGENHARIA DE SOFTWARE

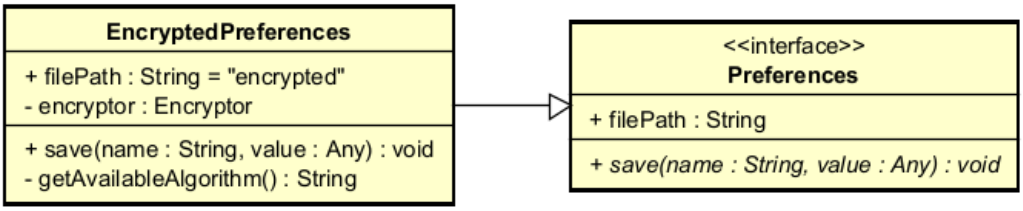
A biblioteca Snappier utiliza em seu funcionamento duas notáveis técnicas e estratégias vistas na literatura de desenvolvimento de software, sendo elas o Polimorfismo e a Deserialização. Nesta seção, será feita uma visão geral de como a biblioteca tira vantagem delas para atingir seu objetivo.

3.1.1 Polimorfismo

Polimorfismo, do grego *"poly"* (muitos) e *"morph"* (formas), é um conceito em Programação Orientada a Objetos que permite que instâncias de diferentes classes sejam tratadas como objetos de uma classe pai em comum, em outras palavras, objetos polimórficos podem "assumir a forma" de outra classe acima na hierarquia. Utilizar o polimorfismo traz vantagens como a melhora da usabilidade, flexibilidade e simplicidade do código, pois permite que interfaces genéricas acomodem várias implementações distintas.

Outra vantagem do polimorfismo está na organização da estrutura comportamental e de dados. Ao definir interfaces com métodos ou campos abstratos cujas implementações podem variar, é possível trabalhar apenas com essas abstrações, algo importante no desenvolvimento de bibliotecas, onde as implementações específicas podem não ser conhecidas. No exemplo abaixo, `EncryptedPreferences` é um objeto polimórfico, pois pode assumir-se como `Preferences` e fornece as implementações concretas dos métodos e os valores reais dos campos definidos em sua superclasse:

Figura 3 – Exemplo de uma implementação especializada em notação UML



Fonte: Próprio autor (2025)

Usar o tipo mais genérico como argumento de um método ou dependência de uma classe permite implementar estratégias que aceitam diferentes tipos de objetos, desde que sejam subtipos do tipo genérico especificado. Isso possibilita a integração de novos tipos ao sistema, sem modificar o código existente, respeitando o princípio aberto/fechado (*"aberto para extensão, fechado para modificação"*) (MARTIN, 1996). Além disso, implementações especializadas podem manter suas propriedades e comportamentos individuais, garantindo que a lógica de cada subtipo seja preservada e aplicada corretamente quando necessário. A Figura 4 mostra um exemplo fictício de uma implementação personalizada que utiliza criptografia ao salvar preferências.

Figura 4 – Exemplo de uma implementação especializada na linguagem Kotlin

```kotlin
interface Preferences {
    val filePath: String
    fun savePreference(name: String, value: Any)
}

class EncryptedPreferences(
    private val encryptor: Encryptor
) : Preferences {
    override val filePath: String = "/users/0/tokens.conf"
    override fun savePreference(name: String, value: Any) {
        FileOutputStream(File(filePath), append: true).use {
            it.write("$name=".toByteArray())
            it.write(encryptor.encrypt(value))
        }
    }

    fun getAvailableAlgorithm(): String = "AES"
}
```

Fonte: Próprio autor (2025)

Seguindo esse exemplo, um método que aceite qualquer tipo de `Preferences` estaria usufruindo do polimorfismo, pois qualquer implementação poderia satisfazer seus requisitos. Além disso, as particularidades de cada implementação ainda estão disponíveis dentro do método, desde que haja visibilidade apropriada. A Figura 5 mostra um exemplo disso, aproveitando as definições anteriores. Nesse exemplo, `getAvailableAlgorithm()` é uma singularidade apenas da implementação de preferências criptografadas, estando disponível após uma verificação de instância:

Figura 5 – Exemplo de polimorfismo na linguagem Kotlin

```kotlin
fun main() {
    val fakeEncryptor = Encryptor { it.toString().toByteArray() }
    val preferences: Preferences = EncryptedPreferences(fakeEncryptor)
    saveRefreshToken(preferences = preferences, refreshToken = "123456789")
}

fun saveRefreshToken(preferences: Preferences, refreshToken: String) {
    val (prefName, prefValue) = "refreshToken" to refreshToken

    println("Saving preference $prefName")
    if (preferences is EncryptedPreferences) {
        println("Using encryption algorithm ${preferences.getAvailableAlgorithm()}")
    }
    preferences.savePreference(name = prefName, value = prefValue)
}
```

Fonte: Próprio autor (2025)

3.1.2 Deserialização

Em ciência da computação, deserialização refere-se ao processo de conversão de dados representados em formatos específicos (como JSON ou XML) em objetos utilizados em linguagens de programação de alto nível. É a contrapartida da serialização. Objetos serializáveis são aqueles cujas propriedades podem ser representadas em um desses formatos, sendo possível revertê-los de volta para objetos de alto nível, dentro da linguagem de programação utilizada. A linguagem de programação Java, com a qual o Kotlin tem interoperabilidade, possui suporte à serialização através da interface `Serializable`. Entretanto, seu uso não é recomendado, pois não fornece nenhuma forma de intervenção sobre como os dados são serializados e deserializados (WOOLCOCK, 2014).

A biblioteca `kotlinx.serialization` é comumente utilizada em Kotlin por fornecer funcionalidades a várias plataformas, controle sobre a serialização, além de conter suporte a diversos formatos de serialização (JETBRAINS, 2021). Snappier utiliza o Kotlin Serialization em conjunto com a biblioteca Ktor para o consumo de APIs de forma assíncrona, onde a deserialização correta de objetos é garantida através de contratos e adaptadores de serialização. Estes possibilitam a utilização de polimorfismo ao especificar como um determinado elemento deve ser convertido para uma classe de dados específica, podendo manter as características herdadas. A Figura 6 mostra um adaptador de serialização que utiliza o polimorfismo para transformar um `PostElement` em sua implementação específica (texto, imagem ou vídeo), a partir de um identificador do tipo de postagem:

Figura 6 – Exemplo de Adaptador de Serialização implementando polimorfismo

```kotlin
object PostSerializer : KSerializer<PostElement?> {
    override val descriptor = PostElement.serialDescriptor

    override fun serialize(encoder: Encoder, value: PostElement?) {
        encoder.encodeSerializableValue(PostElement.serializer(), value!!)
    }

    override fun deserialize(decoder: Decoder): PostElement? {
        val postElement = decoder.decodeSerializableValue(PostElement.serializer())
        val postType = PostType.parse(postElement.id)

        return when (postType) {
            PostType.TEXT → decoder.decodeSerializableValue(TextPostElement.serializer())
            PostType.IMAGE → decoder.decodeSerializableValue(ImagePostElement.serializer())
            PostType.VIDEO → decoder.decodeSerializableValue(VideoPostElement.serializer())
            else → null
        }
    }
}
```

Fonte: Próprio autor (2025)

Neste exemplo, o método de deserialização funciona ao primeiro decodificar um `PostElement`, uma classe que representa um elemento de postagem e que está em uma hierarquia mais alta, não possuindo nenhuma funcionalidade ou representação especializada. Em seguida, o tipo de postagem é determinado através de um identificador único pertencente a ele. Baseado nesse tipo de postagem, a função tenta decodificar o subtipo apropriado, podendo ser uma postagem de texto, imagem ou vídeo, utilizando seus serializadores específicos. O retorno da função continua sendo `PostElement`, porém a sua implementação específica foi determinada através dos identificadores.

3.2 ARQUITETURA DO PROJETO

A biblioteca organiza e renderiza interfaces de usuário utilizando uma estrutura baseada em Elementos e Componentes, que trabalham em conjunto para criar e gerenciar a interface. Os elementos representam os dados brutos, originados de uma fonte externa, como um servidor ou banco de dados local, e incluem identificadores únicos para cada item. Já os componentes são responsáveis por definir os aspectos visuais e comportamentais da interface, determinando como os dados dos elementos serão exibidos e interagindo com o sistema por meio de instruções de renderização enviadas para o Compose Multiplatform.

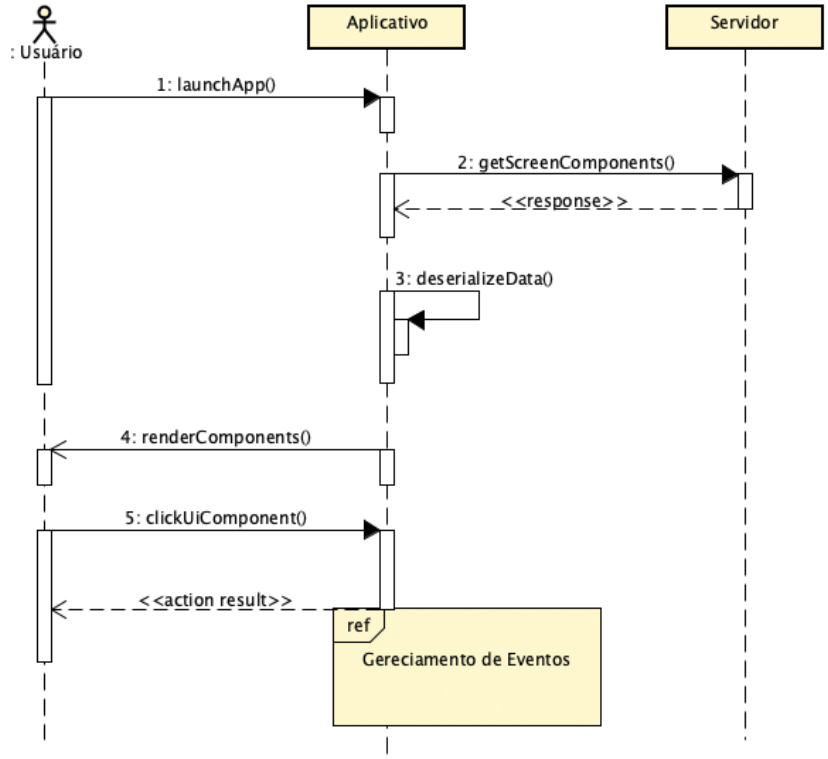
A biblioteca Snappier implementa um mecanismo interno de canais de comunicação, que permite a troca de informações entre componentes, e suporta interações diretas entre eles, o usuário e o sistema operacional. Isso ajuda a promover um alto

nível de desacoplamento e facilita a manutenção do sistema.

O fluxo de interação típico entre o usuário e o aplicativo ocorre da seguinte maneira: ao abrir o aplicativo, os recursos necessários, incluindo os da biblioteca, são carregados. O aplicativo então realiza uma requisição HTTP para um servidor remoto ou acessa um banco de dados local, buscando uma lista de elementos que representam os dados a serem exibidos. Essa lista é processada pela biblioteca, que utiliza os identificadores únicos de cada elemento para localizar e associar os componentes visuais registrados correspondentes. Em seguida, a biblioteca executa, em sequência, as instruções de renderização de cada componente, montando a interface de maneira dinâmica.

Após a fase de renderização, os componentes e o usuário estão aptos a postarem eventos entre si, que são gerenciados pelo mecanismo de comunicação, podendo gerar mensagens ou ações para o sistema operacional. A partir deste ponto, o usuário pode interagir com o aplicativo. A Figura 7 corresponde ao diagrama de sequência que ilustra os passos descritos no fluxo normal de operação, a partir da tela inicial de um aplicativo criado com Snappier: [1]

Figura 7 – Diagrama de sequência da requisição da tela inicial



Fonte: Próprio autor (2025)

[1] O "Gerenciamento de Eventos" foi utilizado no diagrama como Interação por questões de simplicidade

3.2.1 Elementos

Elementos são a representação de dados de um Componente, objeto ao qual está diretamente associado. Eles contêm, como base, o identificador único e a lista de conteúdos utilizados pelo seu Componente correspondente, além de parâmetros extras fornecidos pelo servidor que poderão ser utilizados para auxiliar sua renderização e seu comportamento.

Esta é a interface pública que permite a extensão da biblioteca através do polimorfismo, para que seja possível ampliar e acomodar qualquer representação de dados, sem exigir mudanças significativas na estrutura interna da biblioteca. Isso significa que novos tipos de dados ou componentes podem ser introduzidos e integrados ao sistema de forma fluida, mantendo a consistência e a compatibilidade com as implementações existentes.

Os dados podem ser representados e transportados de qualquer forma, por exemplo, utilizando JSON e APIs REST. A biblioteca define o contrato base de todos os elementos, mas também permite suas extensões através de dados extras que serão interpretados pelos componentes. No exemplo da Figura 8, o elemento possui o identificador "`cart_screen`" e uma lista com dados relacionados ao seu conteúdo, nesse caso, os botões, textos e imagens:

Figura 8 – Exemplo de Elemento em notação JSON

Fonte: Próprio autor (2025)

3.2.2 Componentes

Componentes representam as características e arranjos visuais e comportamentais de uma parte da interface gráfica. Em outras palavras, o usuário do aplicativo interage com ele através dos componentes. Cada componente é identificado exclusivamente com um código identificador (*id*), utilizado pela biblioteca para guardar referência a ele, sempre que o motor de renderização receber uma instrução para renderizá-lo. Isto garante que os componentes corretos sejam exibidos e que seu estado possa ser rastreado posteriormente em eventos de comunicação.

Os componentes também contêm as suas instruções de renderização, que serão transmitidas ao Compose Multiplatform pelo gerenciador de renderização da biblioteca. Eles definem *o que* mostrar na tela e *como*, utilizando o conteúdo vindo do elemento ao qual estão associados, da forma que melhor se adequarem.

Essas instruções são implementações do contrato definido pela biblioteca para os componentes visuais, onde ela está apenas interessada no resultado, sem especificar de que forma as instruções serão colocadas. Isto é possível pois não há dependência de classes concretas, apenas abstrações (MARTIN; MARTIN, 2006), exemplificando o princípio de inversão de dependência (DIP), definido por Robert C. Martin na sua publicação "Design Principles and Design Patterns":

> If the OCP states the goal of OO architecture, the DIP states the primary mechanism. Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes (MARTIN, 2000, p. 12).

Figura 9 – Exemplo de um componente capaz de renderizar vídeos

```
class SnappierVideoComponent : SnappierObservableComponent( id: "snappier_video") {
    @Composable
    override fun View(data: Element, extras: Map<String, Any?>?) {
        data.contents.firstOrNull()?.let { content →
            content.videos?.firstOrNull()?.let { videoData →
                NativeVideoPlayer(
                    modifier = videoData.constraintsModifier(),
                    video = videoData
                )

                LaunchedEffect(Unit) {
                    videoData.events.find { it.trigger == EventTrigger.OnDraw }?.let {
                        emmitEvent(it)
                    }
                }
            }
        }
    }
}
```

Fonte: Próprio autor (2025)

A biblioteca Snappier utilizará para o trabalho de renderização qualquer classe que esteja apta a fornecer instruções para tal, segundo o contrato do componente. No exemplo da Figura 9, o componente tem como responsabilidade definir as suas instruções de renderização a partir de um elemento (nesse caso, um reprodutor de vídeo) e especificar o seu identificador único. Os componentes também podem emitir eventos de comunicação, que são utilizados pela biblioteca para fornecer uma interface de comunicação entre eles mesmos e o sistema operacional. Esta funcionalidade está descrita na próxima subseção.

3.2.3 Eventos

Em determinado momento, os elementos da tela podem ter a necessidade de manter comunicações entre eles mesmos ou com o sistema operacional. Por causa da natureza dinâmica do SDUI, os componentes ativos não têm conhecimento de outros com quem possam estar compartilhando espaço na tela atual. Snappier dá suporte à comunicação entre componentes utilizando um sistema baseado no modelo Comunicador/Receptor em conjunto com o modelo Intenção/Ação, este último, bastante utilizado no desenvolvimento de aplicativos para o sistema operacional Android (KAUR; SHARMA, 2014).

A comunicação entre componentes ocorre por meio de interfaces que definem "comunicadores", cujas implementações devem fornecer canais de comunicação com dados genéricos (representados na biblioteca como dicionários) acessíveis a todos os ouvintes interessados em um canal específico. Esses ouvintes, chamados "receptores de comunicação", identificam dados provenientes de componentes específicos e executam ações apropriadas com base nas informações recebidas. Assim, um componente comunicador pode enviar mensagens a qualquer componente da árvore de renderização que esteja interessado.

Por exemplo, se uma caixa de seleção for marcada ou desmarcada, ela pode enviar uma mensagem através de um desses canais, permitindo que um componente de texto escutando o canal atualize seu conteúdo ou aparência conforme o estado da caixa de seleção. Essa abordagem desacopla os componentes, tornando o sistema mais modular e fácil de manter, pois os componentes não precisam de referências diretas entre si e podem se comunicar por meio de canais definidos sob demanda.

Quando componentes precisam que o sistema operacional realize ações, a biblioteca usa o modelo Intenção/Ação, representado na forma de "eventos". Eventos contêm "ações", que definem o que deve ser feito, e "gatilhos", que especificam quando essas ações devem ser executadas. Um exemplo comum de evento em aplicativos é o clique (ou toque). Por exemplo, quando um botão é clicado, ele dispara um evento para o gerenciador de renderização, que então estabelece a comunicação entre o aplicativo e o sistema operacional. O evento pode incluir ações específicas, como atualizar a

interface de usuário ou interagir com hardware, como a câmera ou o GPS. Informações adicionais, como a localização ou dados inseridos pelo usuário, podem ser enviadas junto com o evento, garantindo que o contexto correto esteja disponível quando a ação precisar ser executada.

3.2.4 Renderização

O processo de renderização ocorre desde a criação e registro dos componentes até a sua chamada para renderização, responsabilidade do Jetpack Compose. A biblioteca inclui e registra componentes básicos como texto, imagem e vídeo assim que a sua classe principal for instanciada. Para registrar componentes personalizados, é necessário implementar a interface `Component`, que fornece os métodos necessários para o funcionamento da biblioteca em relação à UI.

Como mencionado anteriormente, a delegação de instruções de renderização permite à biblioteca utilizá-las através da interface de componente, sem a necessidade de ter conhecimento dos detalhes específicos de cada um. O método responsável por isso recebe como parâmetro um `Element`, representando os dados necessários para desenhar visualmente a UI, e um dicionário, contendo parâmetros extras e opcionais, que podem ser utilizados para, por exemplo, tomar decisões sobre o que desenhar na tela ou utilizá-los durante a comunicação entre componentes.

Na Figura 10 está uma parte da lógica de renderização a partir de um elemento. Nela, o componente correspondente é recuperado através do identificador e renderizado. Acontece também a configuração da estratégia de comunicação e a injeção de dados extras.

Figura 10 – Parte da função de renderização (adaptado)

```
@Composable  lennoard *
fun SnappierView(element: Element) {
    val component by registeredComponent(element.id)
    var extras by remember { mutableStateOf<Map<String, Any?>?>( value: null) }
    if (component is Communicator) {
        val receiver = CommunicationReceiver { data, targetComponentIds →
            val registeredComponents = SnappierComponentRegisterer.getAll()
            registeredComponents.forEach {
                if (it.id in targetComponentIds.orEmpty()) {
                    extras = data
                }
            }
        }
        DisposableEffect( key1: true) {
            component.attachReceiver(receiver)
            onDispose {
                component.detachReceiver(receiver)
            }
        }
    }
    component?.View(element, extras)
}
```

Fonte: Próprio autor (2025)

4 A BIBLIOTECA SNAPPIER

Este capítulo apresenta o estado atual da biblioteca Snappier, destacando sua arquitetura implementada, funcionamento e resultados alcançados. A biblioteca foi projetada para simplificar o desenvolvimento de interfaces de usuário multiplataforma com atualização em tempo real, usando o conceito de Server-Driven UI. Neste capítulo, são mostrados como os componentes foram implementados e como a comunicação interna foi organizada. Além disso, também demonstra fluxos práticos de uso, a distribuição da biblioteca e a análise de desempenho da solução comparada às alternativas existentes.

4.1 ESTADO DO SOFTWARE

A biblioteca Snappier está disponível publicamente no GitHub sob os termos da Licença MIT, uma licença popular que garante permissões completas para uso, modificação, venda e redistribuição tanto em software livre quanto em software proprietário (BUNCH, 2024). Os artefatos para as plataformas compatíveis podem ser acessados via GitHub Packages, incluindo suporte para desenvolvimento multiplataforma com Kotlin Multiplatform. Embora a biblioteca atenda aos objetivos propostos, seu desenvolvimento continua em evolução, com melhorias e novas funcionalidades sendo exploradas. Sua utilização está restrita às configurações de plataformas mostradas na Tabela 2:

Tabela 2 – Restrições e notas para uso da biblioteca por plataforma

Plataforma	**Restrição**	**Notas**
Android	SDK mínimo: 21 (Android 5.0)	SDK recomendado: 24 (Android 7.0)
iOS	Versão mínima do iOS: 12.4	Versão recomendada do iOS: 14.1
Desktop	Necessário JDK 11 para o desenvolvimento	Necessário JRE para a execução
Web	Necessário Kotlin/Wasm (WebAssembly)	Suporte baseado em Kotlin/JS pode ser implementado individualmente

Fonte: Próprio autor (2025)

Estas restrições refletem diretamente o suporte oferecido pelo Compose Multiplatform às plataformas compatíveis com a versão atual da biblioteca. No caso específico da plataforma Web, o Kotlin/JS não é suportado nativamente. No entanto, a arquitetura modular e de código aberto da biblioteca permite que desenvolvedores interessados estendam suas funcionalidades conforme necessário. Como a Snappier foi projetada para ser expansível, novas implementações podem ser incorporadas progressivamente para atender às necessidades de diferentes projetos. Dessa forma, a comunidade tem a liberdade de contribuir e adaptá-la para novas plataformas ou cenários específicos, ampliando suas possibilidades de aplicação.

A utilização da biblioteca é recomendada tanto para o uso pessoal, quanto profissional. Seu modelo de código aberto a torna uma opção acessível para desenvolvedores individuais e pequenas equipes que desejam explorar a abordagem SDUI sem custos adicionais. No ambiente profissional, ela pode ser utilizada para acelerar o desenvolvimento de interfaces dinâmicas e reduzir a necessidade de atualizações constantes via lojas de aplicativos, além de oferecer uma solução flexível para empresas que buscam maior controle sobre a distribuição de interfaces de usuário, possibilitando personalização em larga escala.

A Snappier foi desenvolvida para uso em projetos do Kotlin Multiplatform, portanto, ela tem forte dependência dele e do Compose Multiplatform. No entanto, essa escolha também introduz desafios, pois a necessidade de utilizar o KMP significa que as equipes precisam estar familiarizadas com esse ecossistema, que ainda está em evolução e pode ter limitações em comparação com frameworks nativos maduros.

Por ser uma solução multiplataforma, o desempenho pode variar dependendo da plataforma-alvo, mesmo que o KMP e o Compose Multiplatform sejam eficientes. Apesar dessas limitações, a flexibilidade e a capacidade de evolução da biblioteca Snappier permitem que ela continue sendo aprimorada para superar esses desafios e oferecer uma solução para aplicativos dinâmicos baseados no SDUI.

4.2 ACESSIBILIDADE

A acessibilidade desempenha um papel fundamental no desenvolvimento de aplicativos (BAAZEEM; AL-KHALIFA, 2015), e os sistemas operacionais modernos disponibilizam diversas funcionalidades voltadas para essa necessidade. A biblioteca busca incorporar esses recursos sempre que possível, aproveitando as capacidades do Jetpack Compose. Dessa forma, a biblioteca contribui para que os aplicativos desenvolvidos com sua tecnologia possam oferecer experiências mais inclusivas. Entre os recursos suportados, destacam-se:

a) Descrições de conteúdo dos elementos visuais: a descrição em texto plano é utilizada por mecanismos de leitura de tela quando essa função está habilitada. Isto é particularmente útil para elementos do tipo imagem ou ícones, que normalmente não podem ser descritos automaticamente;

b) Suporte a mudança de foco de componentes, utilizado por tecnologias assistivas como métodos de entrada físicos (por exemplo, um teclado) para trocar o foco na direção especificada pelo usuário, garantindo a navegabilidade ao utilizar métodos de entrada diferentes do toque;

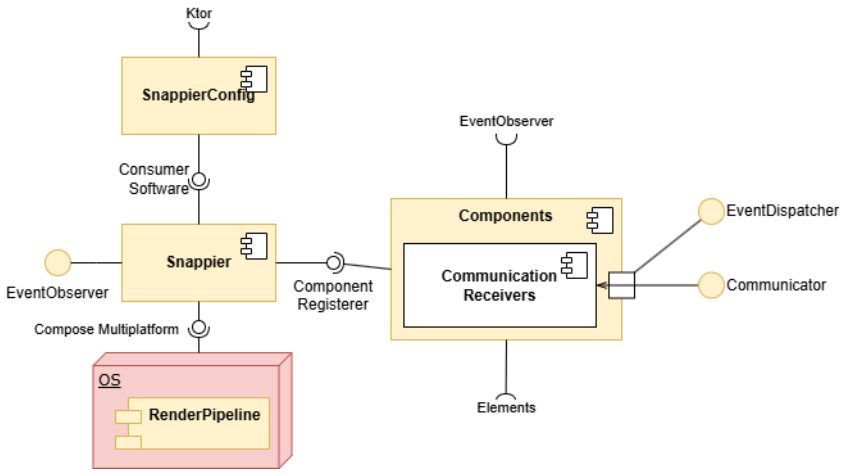
c) Área de toque expandida: elementos que possuem ação registrada ao evento de toque ("elementos clicáveis") têm o tamanho da sua área de toque com no mínimo 48 unidades de altura e 48 unidades de largura para

acomodar usuários com deficiências motoras, seguindo a recomendação do Material Design (GOOGLE, 2021).

4.3 ARQUITETURA IMPLEMENTADA

Esta seção aborda a implementação concreta da arquitetura na biblioteca, mostrando diagramas, componentes e o fluxo de dados. A arquitetura foi dividida em três camadas principais: Fonte de Dados, Gerenciador de Componentes e Motor de Renderização. Os dados brutos são obtidos via requisições HTTP ou banco de dados local e convertidos em elementos de UI pela biblioteca. Os componentes visuais e comportamentais são renderizados dinamicamente utilizando o Compose Multiplatform. A comunicação entre componentes ocorre via canais de mensagens, enquanto eventos tratam das interações com o sistema operacional. A Figura 11 ilustra a arquitetura de alto nível, através de um diagrama de componentes, destacando os principais componentes e como eles interagem:

Figura 11 – Diagrama de componentes da biblioteca



Fonte: Próprio autor (2025)

Neste diagrama, o componente `Snappier` é o ponto de partida e o núcleo da biblioteca, onde os elementos centrais da arquitetura estão organizados. Ele registra os componentes através do seu `ComponentRegisterer` e interage com o Compose Multiplatform para renderizar as interfaces. O ponto de configuração da biblioteca é o `SnappierConfig`, ele permite que o software que usa a biblioteca a configure para suas necessidades específicas. Este componente utiliza a biblioteca Ktor (JETBRAINS, 2025) para lidar com a comunicação entre cliente e servidor. O componente `RenderPipeline` é responsável por traduzir as instruções de renderização geradas pela Snappier através do Compose Multiplatform para a linguagem nativa do sistema operacional.

Os `CommunicationReceivers` gerenciam a troca de informações entre componentes e elementos. Eles são registrados para escutar eventos específicos e interagir com o restante da arquitetura por meio de canais de comunicação. O `EventDispatcher` encaminha eventos disparados pelos usuários ou pelo sistema operacional para os componentes relevantes, enquanto o `Communicator` facilita a comunicação entre os componentes da interface. Estes dois atuam como intermediários na propagação de eventos e dados.

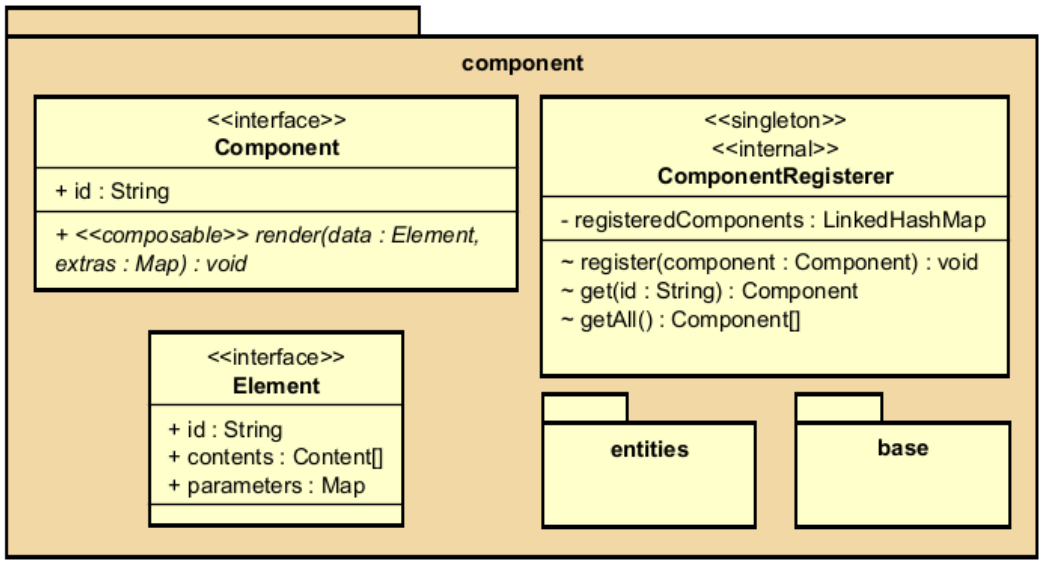
4.3.1 Diagramas de classe

Esta subseção mostra os diagramas de classe pertencentes à arquitetura base e funcionamento principal da biblioteca. Cada diagrama mostrado pertence a um pacote específico no código da biblioteca.

4.3.1.1 Pacote de componentes

Este pacote contém todas as classes responsáveis pelo sistema de componentização da biblioteca com seus dados filhos. Possui dois subpacotes: `entities` e `base` que serão abordados em mais detalhes abaixo. Ele também contém o ponto de partida para o registro de novos componentes criados, o `ComponentRegisterer`, uma classe com uma única instância mantida internamente pela biblioteca para o controle de componentes.

Figura 12 – Diagrama de classe do pacote `component`



Fonte: Próprio autor (2025)

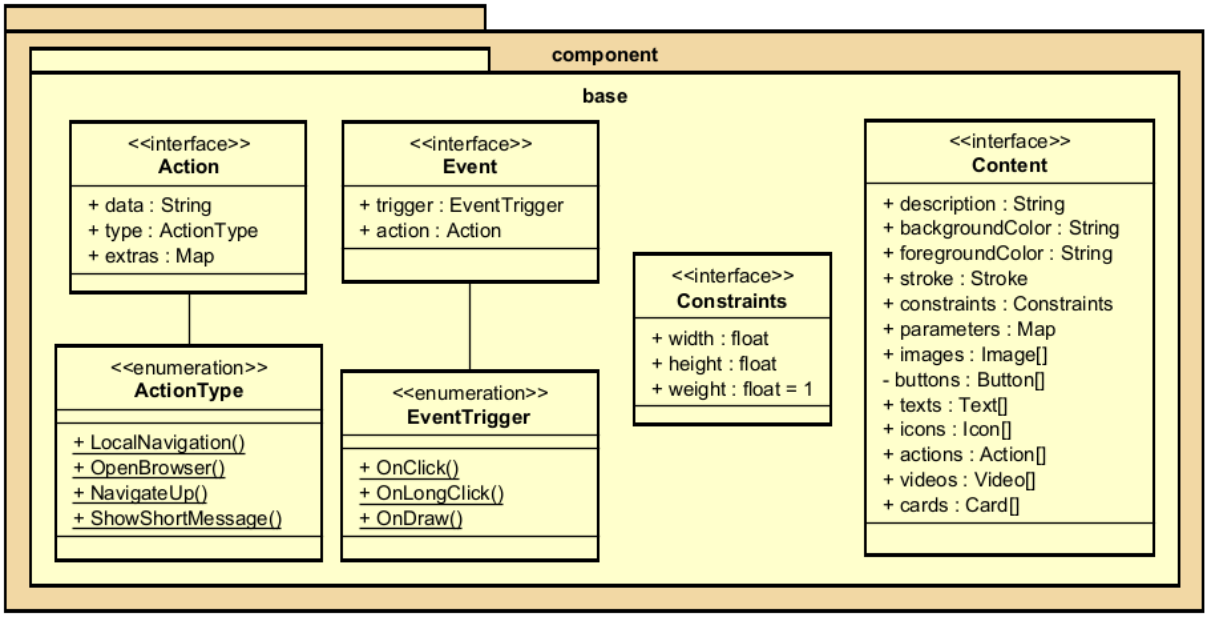
A Figura 12 representa o diagrama de classes do pacote de componentes. As interfaces principais do sistema estão definidas nele. `Component` representa a interface base para todos os componentes do sistema, possui um único atributo que o identifica,

sendo utilizado no registro e na busca e um único método responsável por renderizar o componente na interface de usuário. `Element` define a estrutura de dados associada a cada componente, possui um atributo identificador, uma lista de conteúdos que pode conter e parâmetros adicionais necessários para o seu funcionamento. A classe central `ComponentRegisterer` centraliza o controle sobre os componentes, utilizando os métodos `register` e `get`, garantindo que apenas componentes registrados sejam utilizados. Esta classe possui apenas uma instância e é de uso interno da biblioteca, não estando visível ao software consumidor.

4.3.1.2 Pacote de componentes base

Este pacote está dentro do pacote `component` e contém as classes básicas de dados que permitem as funcionalidades de comunicação entre componentes, navegação e a renderização de conteúdo. A Figura 13 representa o diagrama de classes desse pacote:

Figura 13 – Diagrama de classe do pacote `component/base`



Fonte: Próprio autor (2025)

O pacote base é essencial para oferecer suporte às funcionalidades fundamentais do Snappier, permitindo definir e executar ações através da interface `Action`, que representa uma ação que pode ser executada dentro de um componente em conjunto com a enumeração `ActionType`, que define os tipos de ações que podem ser executadas, tais como navegação para outras telas do aplicativo ou chamadas de funções específicas do sistema operacional. Os eventos e seus gatilhos são especificados com o `Event` e `EventTrigger`. Já o conteúdo renderizável é criado e utilizado usando a

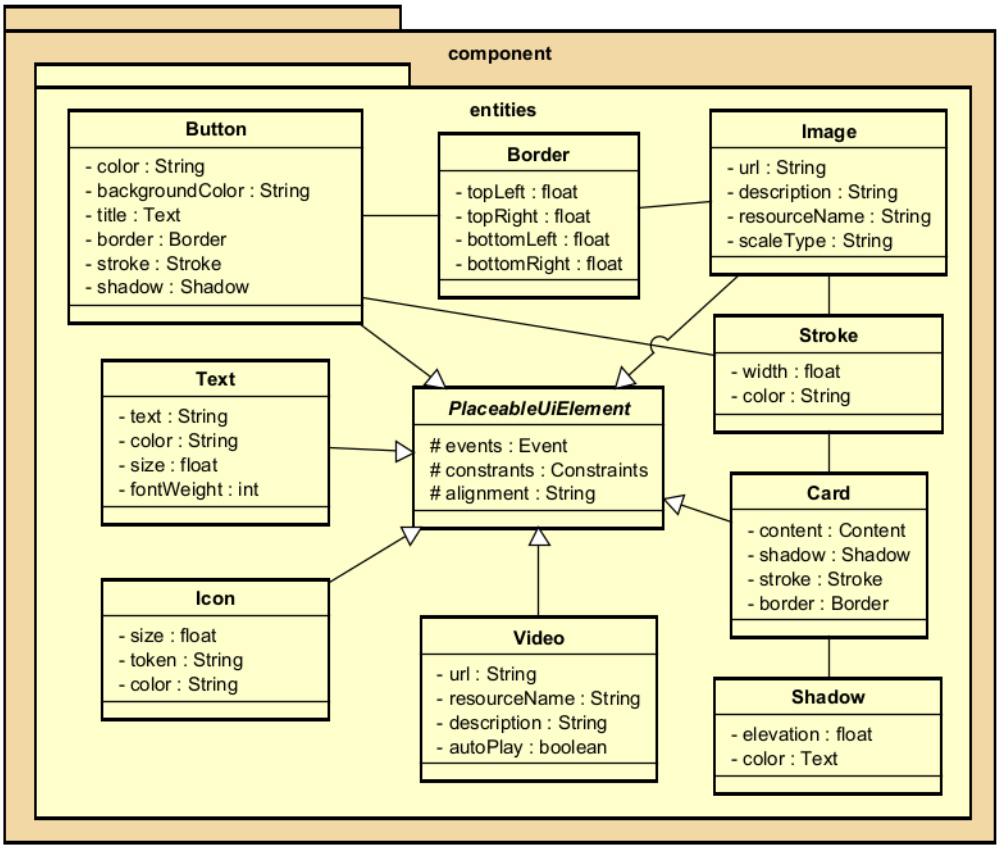
interface `Content`, com suporte a múltiplos tipos de mídia, elementos e flexibilidade no layout dos componentes através de `Constraints`.

Notavelmente, as representações de dados estão definidas na biblioteca como interfaces, permitindo que os desenvolvedores implementem suas próprias soluções, mesmo para classes cujo propósito é meramente a transferência de dados. Esta abstração contribui para a flexibilidade e extensibilidade da biblioteca. Por exemplo, a interface `Content` define os atributos básicos de conteúdo renderizável, porém, caso seja necessário suportar novos tipos de conteúdo no futuro (como animações), é possível criar uma nova classe ou estrutura que implemente essa interface, sem alterar as definições já existentes.

4.3.1.3 Pacote de entidades dos componentes

Este pacote também está dentro do pacote `component` e contém as classes de dados utilizadas exclusivamente para o arranjo dos componentes da tela. Estão presentes nele as representações de texto, imagem, vídeo e suas decorações.

Figura 14 – Diagrama de classe do pacote `component/entities`



Fonte: Próprio autor (2025)

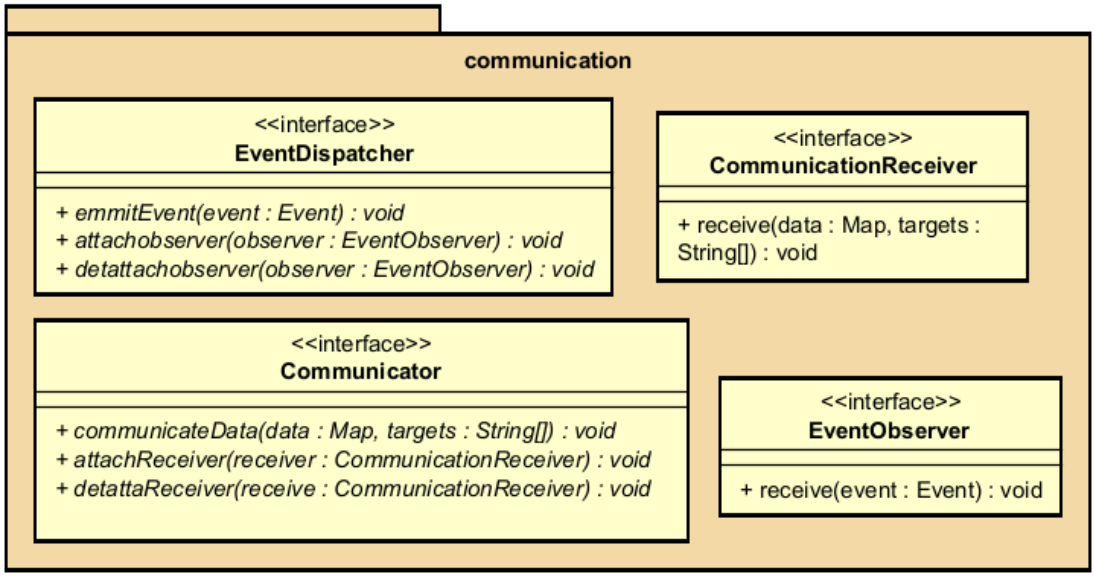
Segundo os dados da Figura 14, podemos destacar três tipos de classes: uma classe base `PlaceableUiElement`, as classes que representam elementos

específicos e as classes decorativas. A classe `PlaceableUiElement` é a classe base abstrata utilizada pela biblioteca para elementos que podem ser posicionados na tela, possuindo uma lista de eventos e restrições de layout. Os elementos específicos são classes que representam os diferentes tipos de componentes que podem ser renderizados, como textos, imagens, botões ou vídeos. Todos esses elementos têm a capacidade de serem colocados na tela, pois herdam de `PlaceableUiElement` e também ganham atributos representando suas configurações de posicionamento através da composição. Por fim, as classes decorativas fornecem estilos adicionais para os elementos visuais, tais como bordas, contornos e sombras.

4.3.1.4 Pacote de comunicação

Neste pacote estão todas as interfaces responsáveis por gerenciar e facilitar a comunicação entre os componentes da aplicação e também com o sistema operacional. Os componentes podem ser eles mesmos "comunicadores" ou "receptores de comunicação" e suas classes estão representadas neste pacote, como mostra a Figura 15:

Figura 15 – Diagrama de classe do pacote `communication`



Fonte: Próprio autor (2025)

Neste pacote, o `EventDispatcher` é a interface responsável por gerenciar e emitir eventos para os `EventObservers` registrados. Estes, por sua vez, representam observadores que escutam e reagem a eventos emitidos pelo emissor de eventos. O `Communicator` funciona como um emissor de dados, ou seja, uma entidade capaz de enviar e gerenciar dados para outros componentes ou receptores, enquanto o `CommunicationReceiver` é o destino desses dados. Essa relação permite a troca de mensagens entre componentes específicos, utilizando uma lista de identificadores

como parâmetro dos métodos de comunicação. Novamente, o uso de interfaces permite que os componentes sejam flexíveis e reutilizáveis, uma vez que qualquer classe pode implementar essas interfaces e se comportar tanto como um comunicador quanto como um receptor, sem dependências rígidas.

4.4 IMPLEMENTAÇÃO EM PROJETOS MULTIPLATAFORMA

Para utilizar a biblioteca Snappier em um projeto Kotlin Multiplatform, é necessário seguir alguns passos básicos, que incluem importar a biblioteca, configurar sua instância principal e ajustá-la para atender às necessidades específicas do projeto. Isso pode ser feito chamando o método construtor que a biblioteca fornece para a inicialização, sendo recomendado configurar a instância no ponto de entrada principal do aplicativo. Caso o projeto precise realizar requisições HTTP externas (como buscar dados de uma API), a biblioteca Ktor pode ser integrada diretamente durante a criação da instância. Na figura 16, é demonstrado um exemplo de criação da instância da biblioteca, configurando o cliente HTTP, especificando os observadores de eventos e um componente visual padrão em casos de erro:

Figura 16 – Exemplo de utilização da biblioteca em um projeto multiplataforma

```
@Composable
fun App() {
    val snappier = Snappier {
        onErrorComposable {
            Box(modifier = Modifier.fillMaxSize(), contentAlignment = Alignment.Center) {
                Text( text: "Error: $it")
            }
        }
        withHttpClient {
            install(Logging) { logger = Logger.DEFAULT }
            install(Auth) { basicAuthCredentials(username = "user", password = "password") }
        }
        setHttpMethod(HttpMethod.Get)
    }.apply {
        addObserver { event → println("Received event: $event") }
    }

    MaterialTheme {
        snappier.SnappierView("http://localhost:8081/api")
    }
}
```

Fonte: Próprio autor (2025)

A Snappier utiliza um sistema de componentização, onde cada elemento da interface do usuário pode ser registrado e reutilizado. Para criar um novo componente, deve-se criar uma classe que implemente a interface `Component` e, então, registrá-la

utilizando o método `register` da biblioteca. Para renderizar conteúdo, a biblioteca fornece métodos que aceitam um ou vários elementos da mesma forma que uma URL, no caso de chamadas de API. É possível escutar os eventos enviados pelos componentes adicionando um observador. A biblioteca fornece um método para isso na sua instância principal.

4.5 ANÁLISE DE DESEMPENHO

Nesta seção, será feita a comparação do desempenho de um aplicativo de exemplo criado utilizando a biblioteca Snappier com dois outros aplicativos. Um foi criado utilizando o desenvolvimento nativo, na plataforma Android, enquanto o outro foi criado utilizando o desenvolvimento híbrido, com o framework Flutter.

As métricas utilizadas na análise são o tempo médio de inicialização e o tempo médio de renderização. Estas métricas foram escolhidas por serem indicadores de desempenho que afetam a experiência do usuário. O tempo de inicialização é o tempo estabelecido entre a criação do processo do aplicativo e o momento em que ele tem o seu primeiro quadro mostrado na tela. Já o tempo de renderização mede quanto tempo leva para exibir a interface do usuário depois que o aplicativo é iniciado ou uma instrução de renderização é solicitada. Uma renderização eficiente garante que a UI seja suave e responsiva, o que é importante em aplicativos que utilizam o Server-Driven UI, pois exigem atualizações frequentes ou conteúdo dinâmico.

Durante o teste, o aplicativo criado, ao ser executado, faz apenas uma requisição para um servidor que retorna os dados para renderização de uma tela de exemplo. Em seguida, o aplicativo utiliza uma técnica diferente, dependendo do objeto do teste, para renderizá-la. A Tabela 3 mostra as configurações dos dispositivos de teste utilizados.

Tabela 3 – Especificações do dispositivo de teste

	Dispositivo hospedeiro	**Dispositivo virtual**
Sistema Operacional	macOS 12.6.3	Android 14.0 (Nível de API 34)
Memória	20GB 2400 MHz DDR4	4GB
Processador	Intel® Core™ i7-8565U (8 Threads)	Intel® Core™ i7-8565U (4 Threads)
Arquitetura	`x86_64`	`x86_64`
Processador gráfico	Intel® UHD Graphics 620 Mobile 2048 MB	Android Emulator Open GL ES Translator

Fonte: Próprio autor (2025)

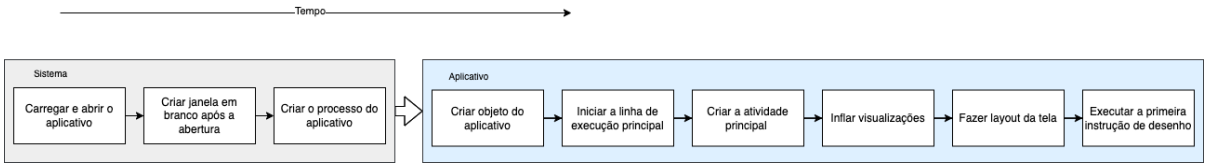
Na esperança de deixar o plano de jogo justo, o dispositivo escolhido para a realização dos testes foi um dispositivo Android virtual (emulador), por fornecer um ambiente de teste consistente ao eliminar as possíveis variabilidades em diferentes dispositivos físicos, tais como especificações de hardware e configurações de software

variadas. O emulador executa o sistema operacional Android Open Source Project básico na versão 14, que garante que personalizações extras como as encontradas em dispositivos físicos não afetem os resultados.

4.5.1 Tempo de inicialização

O tempo de inicialização no sistema operacional Android é definido como o tempo entre o sistema carregar e abrir o aplicativo, criar o seu processo e ter o seu primeiro quadro desenhado na tela. Existem vários estágios para a inicialização de um aplicativo, como mostrado na Figura 17:

Figura 17 – Representação das etapas da inicialização de aplicativos no Android



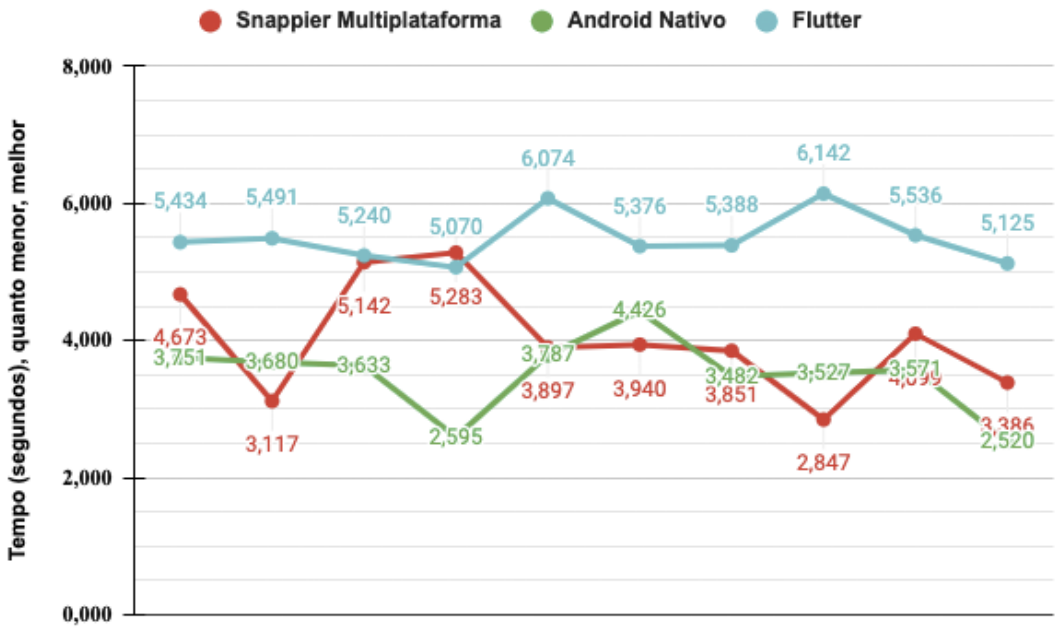
Fonte: App startup time (GOOGLE, 2018) (adaptado)

Primeiro acontece a criação do processo para o aplicativo pelo sistema Android. Logo em seguida, os componentes principais (como Atividades e Serviços) são inicializados. Após isso, o aplicativo carrega os seus componentes para a interface de usuário, suas dependências e quaisquer dados necessários para a inicialização. Somente após esses passos, o primeiro quadro é renderizado, tornando o aplicativo visível ao usuário.

Algumas dessas etapas podem ser puladas durante uma inicialização, resultando em uma inicialização "quente". Isto geralmente ocorre quando o aplicativo está parcialmente em execução (por exemplo, no plano de fundo) ou o seu processo já foi criado e colocado em *cache* pelo sistema operacional em algum momento do passado. Em contraste, quando todas estas etapas são executadas de ponta a ponta, tem-se uma inicialização a frio. Este tipo de inicialização é a mais lenta possível e denota que o aplicativo será inicializado "do zero", sem prévias técnicas de *caching* ou otimizações de código AOT (*ahead-of-time*), quando o sistema operacional realiza otimizações antes de o aplicativo ser executado. Este estado é garantido, por exemplo, na primeira inicialização do aplicativo após ser instalado ou quando o dispositivo é reiniciado (GOOGLE, 2018).

A fim de testar o tempo de inicialização médio do aplicativo de exemplo, dez inicializações a frio foram realizadas em cada objeto de teste, somando um total de trinta. A cada coleta de dados, o dispositivo foi reiniciado, garantindo a remoção de processos criados e dados em cache, o que permite que o próximo teste seja executado nas mesmas condições do anterior. O teste foi monitorado utilizando a ferramenta Perfetto e produziu estes resultados:

Figura 18 – Tempos de inicialização em segundos dos objetos de teste



Fonte: Próprio autor (2025)

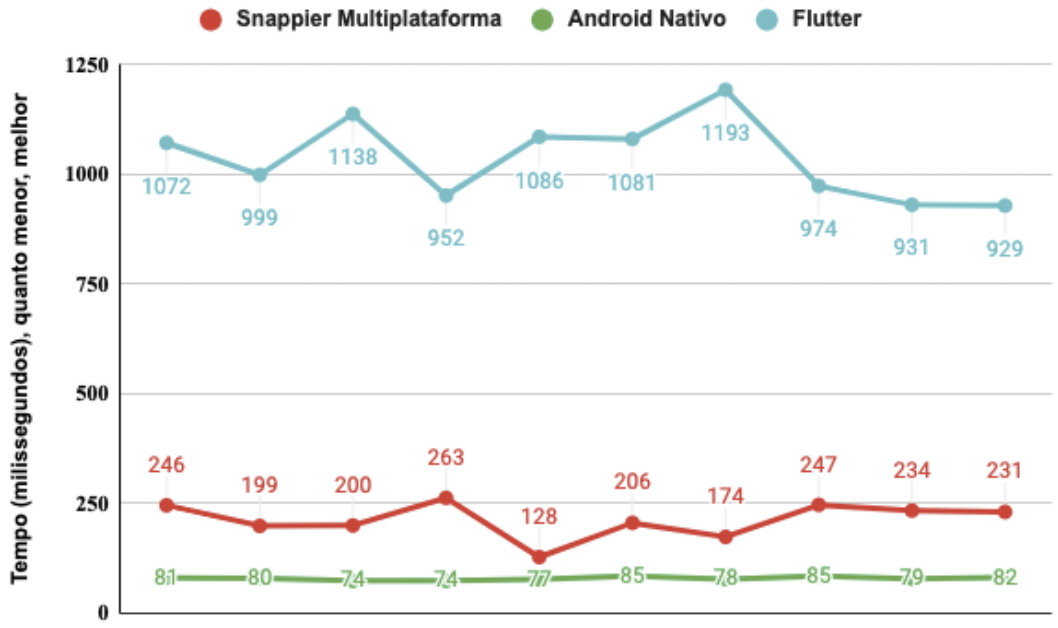
A partir dos dados da Figura 18 podemos inferir que o objeto de teste com Android Nativo tem desempenho constantemente melhor, como esperado. Em quase todas as execuções, este objeto de teste mostra o menor tempo de inicialização, variando de 2,520 a 4,426 segundos e atingindo a média de 3,497.

O teste com o aplicativo criado utilizando a biblioteca Snappier, apesar de indicar a maior variação de resultados entre os três objetos de teste com valores entre 2,847 e 5,283 segundos, mostrou um meio-termo, fornecendo a vantagem de compatibilidade multiplataforma sem comprometer severamente a velocidade de inicialização, com uma média de 4,024 segundos, 15,07% a mais em relação ao Android Nativo. Por fim, o objeto de teste Flutter ficou com 5,488 segundos de média, um aumento de 56,93% em relação ao Android nativo e de 36,38% em relação à Snappier. Com valores entre 5,070 e 6,142 segundos, este objeto de teste foi o mais constante, embora também tenha sido o que apresentou os valores mais altos, sugerindo que pode não desempenhar tão bem em se tratando de tempo de inicialização.

4.5.2 Tempo de renderização

Nesta subseção será destacado o teste do tempo de renderização. Este tempo será considerado como o tempo decorrido do método `onCreate` da atividade principal dos objetos de teste. Este método é chamado logo após a fase de inicialização do aplicativo, e servirá, de forma geral, como forma de mensurar o tempo necessário para configurar e mostrar o conteúdo de uma tela específica. O tempo decorrido será medido utilizando o método `measureTimeMillis`, incluído na biblioteca padrão do Kotlin. Novamente, dez inicializações foram realizadas para cada objeto de teste.

Figura 19 – Tempo de execução do método `onCreate` nos objetos de teste



Fonte: Próprio autor (2025)

Segundo os dados da Figura 19, os resultados do teste mostram que o objeto Android Nativo tem novamente o melhor desempenho, com os menores e mais estáveis tempos de renderização, variando de 74 a 85 milissegundos. O aplicativo criado utilizando a biblioteca Snappier mostra desempenho próximo, com tempos de renderização entre 128 e 263 milissegundos, cerca de 2,6 vezes mais lento e um pouco mais variável que o Android Nativo, mas muito mais rápido que o Flutter, que tem os maiores e mais inconsistentes tempos de renderização, variando de 929 a 1193 milissegundos, com média de 1035, quase 10 vezes mais, indicando uma lacuna de desempenho significativa em comparação com os outros objetos de teste.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho buscou desenvolver a biblioteca Snappier, uma contribuição significativa para o campo do desenvolvimento de interfaces baseadas em Server-Driven UI com suporte ao ecossistema Kotlin Multiplatform. Durante o processo de criação, foi possível unir conceitos teóricos de polimorfismo e deserialização com práticas de engenharia de software, resultando em uma solução modular, extensível e eficiente para aplicativos multiplataforma.

Ser capaz de adaptar-se rapidamente, entregando experiências personalizadas, consistentes e escaláveis é uma vantagem competitiva significativa em um ambiente onde as demandas do usuário são altas. Para equipes de desenvolvimento de software que buscam permanecer à frente na era digital, adotar a abordagem SDUI pode levar a fluxos de trabalho mais eficientes, com experiências de usuário personalizadas e uma resposta mais ágil ao cenário de constantes mudanças nas necessidades do usuário. Este trabalho fornece as ferramentas para atingir isso.

Para alcançar os objetivos, foram utilizadas duas tecnologias de desenvolvimento multiplataforma, o Kotlin Multiplatform e o Compose Multiplatform, ambos trabalhando em conjunto para permitir a criação de código de UI escalável e personalizável a partir de dados obtidos utilizando o SDUI. O artefato final gerado por este trabalho é uma biblioteca para aplicativos multiplataforma criados utilizando o Kotlin Multiplatform publicada no GitHub, com possibilidade de implementações em plataformas individuais, se necessário.

Foi realizada uma análise de desempenho através de testes de inicialização a frio, utilizando o desenvolvimento com Android Nativo e Flutter como comparação. Os testes de desempenho realizados destacaram a eficiência da biblioteca, tanto em relação ao tempo de inicialização quanto ao tempo de renderização. Os resultados mostraram que a Snappier se posiciona como uma solução competitiva em relação a ferramentas existentes, como Beagle e Judo, mostrando uma redução de 26,67% no tempo médio de inicialização de aplicativos quando comparado com Flutter, posicionando-a como uma solução que não sacrifica o desempenho de inicialização para fazer o que se propõe, enquanto oferece vantagens em termos de flexibilidade e alinhamento com as tendências tecnológicas atuais.

Como uma solução emergente, há desafios e oportunidades de evolução. Como trabalho futuro, pode-se sugerir melhorias na arquitetura do projeto, tais como a implementação de cache externo configurável pelo consumidor, suporte a uso offline, potencialmente com interfaces de usuário prontas e configuráveis em caso de erro e suporte a temas dinâmicos configurados pelo cliente. Além disso, pode haver uma plataforma de fácil utilização para o design das UIs que serão utilizadas pela biblioteca

e os aplicativos com o objetivo de fazer com que profissionais de fora do ramo da computação consigam criar e implantar componentes de UI aptos a serem utilizados pela Snappier.

REFERÊNCIAS

BAAZEEM, I. S.; AL-KHALIFA, H. S. Advancements in web accessibility evaluation methods: how far are we? In: **Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services**. [S.l.: s.n.], 2015. p. 1–5.

BUNCH, R. **Exploring the MIT Open Source License: A Comprehensive Guide**. Massachusetts Institute of Technology 255 Main Street, Room NE18-501 Cambridge, MA 02142-1601, 2024. Disponível em: https://tlo.mit.edu/understand-ip/exploring-mit-open-source-license-comprehensive-guide. Acesso em: 15 jan 2025.

CHERNONOG, T. Supply chains of mobile apps: competition, private labels and bypassing when the app's quality is co-created. **Annals of Operations Research**, Springer, v. 332, n. 1, p. 859–890, 2024.

COOMBS, J. **Judo**. 2021. Disponível em: https://www.judo.app. Acesso em: 3 mar 2024.

CORRAL, L.; JANES, A.; REMENCIUS, T. Potential advantages and disadvantages of multiplatform development frameworks–a vision on mobile environments. **Procedia Computer Science**, Elsevier, v. 10, p. 1202–1207, 2012.

GOOGLE. **App startup time**. 2018. Disponível em: https://developer.android.com/topic/performance/vitals/launch-time. Acesso em: 1 set 2024.

GOOGLE. **Assistive technology**. 2021. Disponível em: https://m3.material.io/foundations/overview/assistive-technology. Acesso em: 19 ago 2024.

GOOGLE. **Thinking in Compose**. 2024. Disponível em: https://developer.android.com/develop/ui/compose/mental-model. Acesso em: 17 set 2024.

IBRAHIM, H. **Google Play review times: Expectations and tips to streamline approval**. 2024. Disponível em: https://median.co/blog/google-play-review-times-what-to-expect-and-how-to-streamline-approval. Acesso em: 18 nov 2024.

JETBRAINS. **Kotlin Serialization**. 2021. Disponível em: https://kotlinlang.org/docs/serialization.html. Acesso em: 17 ago 2024.

JETBRAINS. **Compose Multiplatform: Declarative framework for sharing UIs across multiple platforms. Based on Kotlin and Jetpack Compose**. 2023. Disponível em: https://www.jetbrains.com/lp/compose-multiplatform. Acesso em: 17 set 2024.

JETBRAINS. **Ktor: Build Asynchronous Servers and Clients in Kotlin**. 2025. Disponível em: https://ktor.io. Acesso em: 02 feb 2025.

KAUR, P.; SHARMA, S. Google android, a mobile platform: A review. **2014 Recent Advances in Engineering and Computational Sciences (RAECS)**, IEEE, p. 3, 2014.

LISBOA, D. C. A. **Você sabe qual foi a primeira loja de aplicativos da história?** Rua Formosa, 79 - Rudge Ramos, São Bernardo do Campo/SP, 2021. Disponível em: https://canaltech.com.br/apps/primeira-loja-de-aplicativos-da-historia-186185. Acesso em: 31 ago 2024.

MARTIN, R. C. The open-closed principle. **More C++ gems**, v. 19, n. 96, p. 9, 1996.

MARTIN, R. C. Design principles and design patterns. **Object Mentor**, v. 1, n. 34, p. 597, 2000.

MARTIN, R. C.; MARTIN, M. **Agile principles, patterns, and practices in C**. [S.l.]: Pearson Education, 2006. 201 p.

PAVLOV, D. **The basics of Kotlin Multiplatform project structure**. Praha 4, Na Hřebenech II 1718/8, PSČ 140 00, Czech Republic, 2021. Disponível em: https://kotlinlang.org/docs/multiplatform-discover-project.html. Acesso em: 30 jan 2025.

PAVLOV, D. **Expected and actual declarations**. Praha 4, Na Hřebenech II 1718/8, PSČ 140 00, Czech Republic, 2023. Disponível em: https://kotlinlang.org/docs/multiplatform-expect-actual.html. Acesso em: 27 jan 2025.

POLYAKOV, A. **Kotlin Multiplatform**. Praha 4, Na Hřebenech II 1718/8, PSČ 140 00, Czech Republic, 2020. Disponível em: https://kotlinlang.org/docs/multiplatform.html. Acesso em: 11 ago 2024.

RAHMAN, M. *et al.* Fairplay: Fraud and malware detection in google play. In: SIAM. **Proceedings of the 2016 SIAM International Conference on Data Mining**. [S.l.], 2016. p. 99–107.

ROGERS, M. P.; GRATCH, J. A snapshot of current and trending practices in mobile application development. **Journal of Computing Sciences in Colleges**, Consortium for Computing Sciences in Colleges, v. 37, n. 6, p. 54–66, 2022.

ROKIS, K.; KIRIKOVA, M. Challenges of low-code/no-code software development: A literature review. In: SPRINGER. **International Conference on Business Informatics Research**. [S.l.], 2022. p. 3–17.

RONKAINEN, J. *et al.* Experiences on mobile cross-platform application development using phonegap. In: **Proceedings of The Sixth International Conference on Advances in Human oriented and Personalized Mechanisms, Technologies, and Services**. [S.l.: s.n.], 2013.

SAMPAIO, R. **Backend Driven Content: why we developed a Server Driven UI framework at Nubank**. 2023. Disponível em: https://building.nubank.com.br/server-driven-ui-framework-at-nubank. Acesso em: 5 set 2024.

SARKAR, S. **Build a Server Driven UI**. 2023. Disponível em: https://techitup.io/blog/Build-a-Server-Driven-UI-TFdlnm. Acesso em: 11 ago 2024.

SEITZ, B. Effective feature management. **Feature Toggles**, 2022.

SHUGAN, S. M. **Brand loyalty programs: are they shams?** [S.l.]: INFORMS, 2005. 185–193 p.

SOUZA, R. de. **Server Driven UI com Beagle Web: Motivação e conceitos iniciais**. 2020. Disponível em: https://medium.com/zup-it/server-driven-ui-com-beagle-web-motiva%C3%A7%C3%A3o-e-conceitos-iniciais-1f4af7f47e19. Acesso em: 3 mar 2024.

VANZIN, R.; BONIATI, B. B.; SILVA, T. L. da. Estudo de frameworks de desenvolvimento multiplataformas para dispositivos móveis. **Anais do Encontro Anual de Tecnologia da Informação**, v. 2, n. 1, p. 237–237, 2012.

WANG, H. *et al.* Beyond google play: A large-scale comparative study of chinese android app markets. In: **Proceedings of the Internet Measurement Conference 2018**. [S.l.: s.n.], 2018. p. 293–307.

WOOLCOCK, K. **Serialization in Java (Binary and XML)**. [S.l.]: Iowa State Univeristy, 2014.

ZHU, M. **How server-driven UI keeps our shop fresh**. 2024. Disponível em: https://blog.duolingo.com/server-driven-ui. Acesso em: 07 jan 2025.