PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE UM APLICATIVO UTILIZANDO O FRAMEWORK FLUTTER E ARQUITETURA LIMPA

CARLOS EDUARDO DE OLIVEIRA BUENO

GOIÂNIA
2021

CARLOS EDUARDO DE OLIVEIRA BUENO

DESENVOLVIMENTO DE UM APLICATIVO UTILIZANDO O FRAMEWORK FLUTTER E CLEAN ARCHITECTURE

Trabalho de Conclusão de Curso apresentado à Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, como parte dos Requisitos para obtenção do título de Bacharel em Ciências da Computação.

Orientador(a): Rafael Leal Martins

GOIÂNIA

2021

RESUMO

Este trabalho apresenta o desenvolvimento de um aplicativo mobile que realiza a autenticação de usuário através de e-mail e senha. A aplicação utiliza respeita princípios arquiteturais da Arquitetura Limpa para o desenvolvimento da aplicação. A utilização da Arquitetura Limpa tem o intuito de fazer com que sistemas tenham camadas com responsabilidades bem definidas e não dependam de tecnologias que não fazem parte da regra de negócio da aplicação, facilitando assim a evolução e manutenção. A aplicação é desenvolvida utilizando o framework Flutter, que possibilita a criação de aplicações híbridas para dispositivos móveis. Este relatório faz um estudo sobre tipos de aplicações mobile, arquiteturas em camadas, coesão, acoplamento e Arquitetura Limpa. Por fim é apresentado uma aplicação real onde são implementados conceitos de Arquitetura Limpa.

Palavras-Chave: Arquitetura. Arquitetura Limpa. Flutter. Desenvolvimento Mobile.

SUMÁRIO

1 INTRODUÇÃO

Existem, no Brasil, 234 milhões de smartphones (celulares inteligentes) em uso (FGVcia, 2020). E, segundo Statista, existem 3.80 bilhões de smartphones em uso no mundo, ou seja, 48.81% de toda a população mundial. Levando essa crescente adoção de smartphones em consideração, o desenvolvimento de soluções de software para estes dispositivos tornou-se um grande atrativo.

Com a crescente demanda para criação de aplicações mobile (Grand View Research, 2020) a implementação de boas arquiteturas de software pode fazer com que menos esforço e tempo sejam necessários a longo prazo (Robert C. Martin, 2012). Devido a importância da arquitetura em projetos de software, o presente trabalho faz um estudo de arquiteturas em camadas, apresenta conceitos de coesão e acoplamento e por fim demonstra uma aplicação da Arquitetura Limpa (Robert C. Martin, 2012) através de um caso de uso de autenticação utilizando o framework Flutter para o desenvolvimento de um aplicativo.

1.1 Objetivos Gerais

Desenvolver uma aplicação utilizando os conceitos de Arquitetura Limpa onde o usuário fará a autenticação e, em caso de sucesso, será redirecionado para a página inicial da aplicação.

1.2 Objetivos Específicos

Os objetivos específicos são:

- Fazer um estudo e utilizar o framework Flutter na versão 2.0;
- Estudar a linguagem Dart;
- Apresentar um estudo sobre Arquitetura em Camadas;
- Apresentar um estudo sobre coesão e acoplamento;
- Utilizar a Arquitetura Limpa na criação do aplicativo;
- Utilizar o software Android Studio como ambiente de desenvolvimento.

1.3 Justificativa

Projetos de software mal arquitetados tendem, com o passar do tempo, tornarem-se difíceis de serem mantidos. Isso se deve a, muitas vezes, a criação de código com alto acoplamento e baixa coesão, o que dificulta a sua alteração e evolução. Para solucionar esse problema, a utilização da Arquitetura Limpa, abordagem que visa separar o desenvolvimento da aplicação em camadas para que, quando necessário, evoluções e manutenções sejam feitas de maneira menos dispendiosa.

1.4 Metodologia

Este trabalho utiliza de coleta de informações com o objetivo de auxiliar no desenvolvimento de um caso de uso utilizando o padrão de arquitetura Arquitetura Limpa a fim de demonstrar uma aplicação uma aplicação bem estruturada respeitando os princípios de coesão e acoplamento:

- Pesquisa bibliográfica da linguagem/framework através da própria documentação e artigos na internet;
- Pesquisa bibliográfica sobre Arquitetura Limpa através de livros e artigos na internet;
- Estudar Arquitetura em Camadas, coesão e acoplamento;
- Implementação de protótipos através de uma ferramenta de prototipação;
- Desenvolvimento da aplicação.

2 REFERENCIAL TEÓRICO

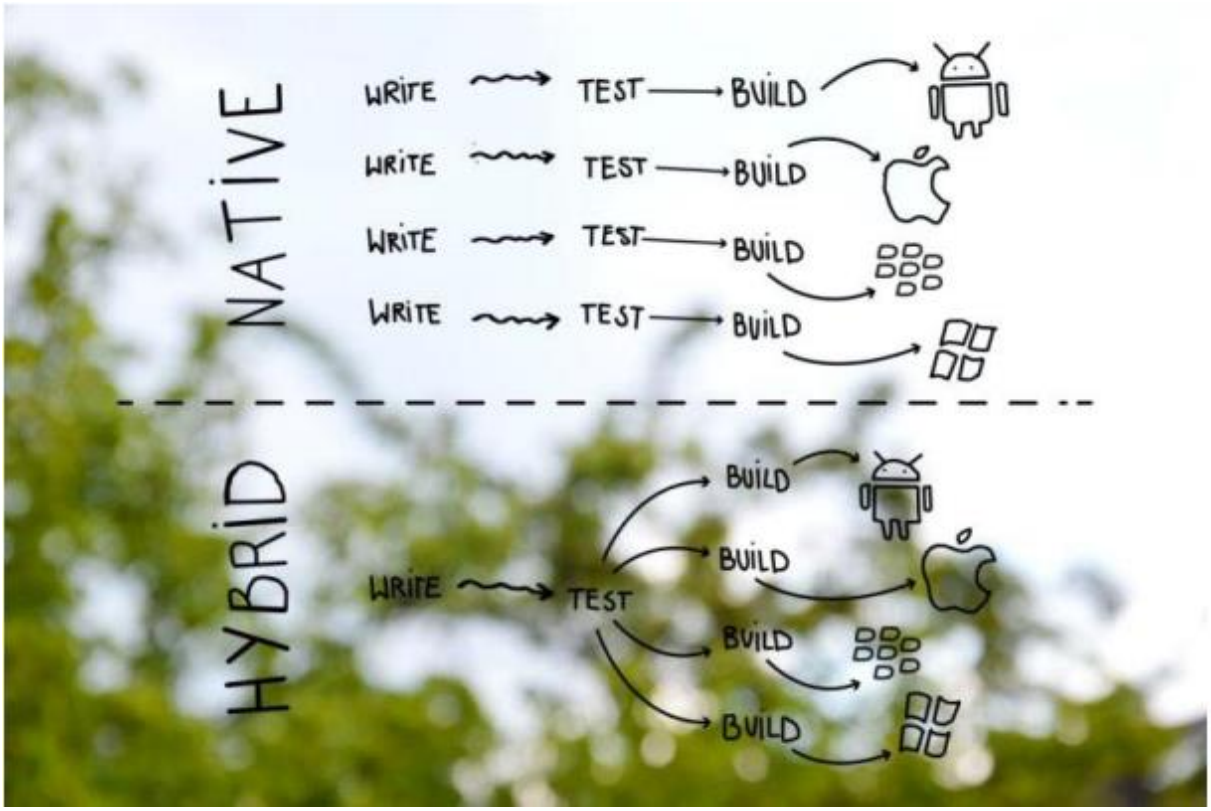
Neste capítulo é abordado o referencial teórico utilizado para o desenvolvimento e execução do projeto apresentado, apresentando conceitos essenciais para melhor compreensão da aplicação proposta.

2.1 Aplicações híbridas e nativas

Para o desenvolvimento de aplicações móveis pode-se escolher entre duas principais abordagens, a nativa ou híbrida. Para o desenvolvimento de aplicações nativas o desenvolvedor precisa gerar artefatos exclusivos para cada plataforma na qual o software será disponibilizado. Como diz Madureira (2017), nas aplicações nativas é necessário programar na linguagem de cada sistema operacional, como o Java ou Kotlin no Android e o Swift no iOS, cada plataforma apresentando suas próprias ferramentas e elementos de interface.

Já para as aplicações híbridas, necessita apenas de escrever o código na linguagem da tecnologia escolhida e, em suma, disponibilizar a aplicação para diversas plataformas a partir do mesmo código-fonte. O desenvolvimento de aplicações híbridas podem ter três abordagens (Taqtile, 2016): encapsular um sistema web em containers nativos e apresentá-los como aplicativos (são chamados de progressive web apps ou simplesmente PWA); escrever a aplicação em determinada linguagem e então este ser traduzido para a linguagem nativa da plataforma; e por fim um misto entre a segunda opção e código nativo da plataforma. A Figura 1 ilustra a diferença entre a abordagem nativa e híbrida no desenvolvimento de aplicações mobile.

Figura 1 – Diferença entre o desenvolvimento híbrido e nativo



Fonte: RADEJ (2017)

2.2 Flutter

Flutter é um framework desenvolvido pela Google primeiramente anunciado em 2015 em uma apresentação de Eric Seidel. Inicialmente, na fase experimental, levava o nome de Sky e posteriormente nomeado para Flutter. Na documentação oficial da ferramenta dá-se a definição (em tradução direta): “Flutter é o kit de ferramentas de IU portátil do Google para criar aplicativos bonitos e compilados de forma nativa para dispositivos móveis, web e desktop a partir de uma única base de código. O Flutter é usado por desenvolvedores e organizações em todo o mundo e é gratuito e de código aberto.”

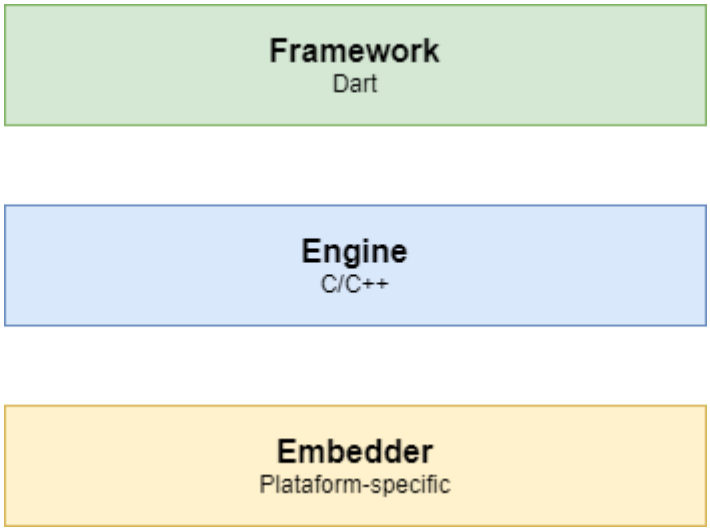
Em dezembro de 2018 foi lançada a versão 1.0 do framework como primeira versão estável da ferramenta. De acordo com a própria documentação, o lançamento da versão estável teve uma grande adoção da ferramenta com milhares de aplicativos sendo desenvolvidos.

2.2.1 Arquitetura do Flutter

De acordo com a documentação oficial as aplicações desenvolvidas em Flutter, durante a fase de desenvolvimento, são executadas em uma Virtual Machine (VM) que possibilita uma recompilação rápida apenas dos componentes que sofreram mudanças, não necessitando reconstruir toda a aplicação. Entretanto, na etapa de releasing (lançamento da aplicação) todo o código é compilado diretamente para código de máquina (intel x64, ARM ou JavaScript).

O Flutter é construído com C, C++, Dart e Skia (uma motor gráfico de renderização 2D) e, como mostrado na Figura 2, possui uma arquitetura de camadas.

Figura 2 - Arquitetura do Flutter



Fonte: Adaptado de Documentação Flutter

Na camada mais baixa do diagrama está o **Embedder**. Cada plataforma possui um Embedder específico que pode ser escrito em Java e C++ para Android, Objective-C/Objective-C++ para iOS e macOS, e C++ para Windows e Linux. O Embedder provê um ponto de entrada para a aplicação e também é responsável por coordenar acesso a serviços do sistema operacional, como: renderização, acessibilidade, entrada de dados e também gerencia o laço de eventos (Event Loop).

Ainda tomando como referência a Figura 2, tem-se a Engine ou também chamada de Flutter Engine. Este componente, em sua maior parte, é escrito em C++ e tem as primitivas necessárias para suportar todas aplicações Flutter. Este mecanismo é responsável por rasterizar os componentes da tela sempre que for preciso atualizar a interface. Nesta etapa está incluído a implementação de baixo nível

da API do Flutter, como gráficos através do Skia, layout de textos, entrada e saída de arquivos e dados de rede e etc.

No topo do diagrama está o Framework Flutter. Este é o componente mais utilizado pelos desenvolvedores. A Figura 3 ilustra os principais componentes do framework.

Figura 3 - Flutter Framework

Fonte: Nix Software Engineering

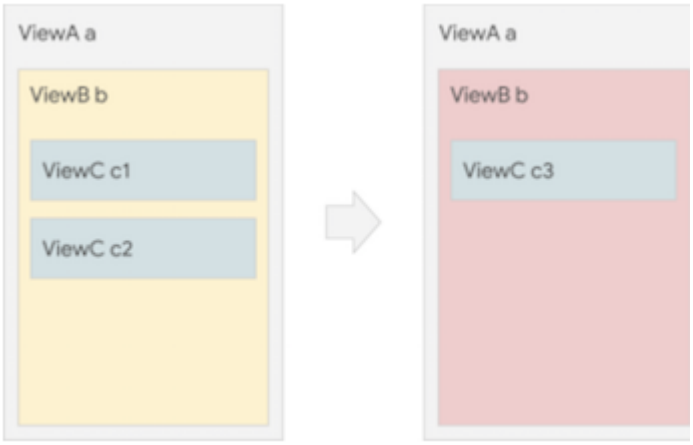
De baixo para cima, temos:

- **Foundation:** provê abstrações para serem utilizadas por serviços como Animation (animação), Painting (pintura) e Gestures (gestos).
- **Rendering:** prove abstração para trabalhar com layouts. Nesta camada é possível criar uma árvore de componentes renderizados.
- **Widgets:** Nesta camada possibilita criar abstração de composição. Basicamente as classes utilizadas nesta camada serão tratadas como render objects na camada de renderização.
- **Material e Cupertino:** Esta camada provê um conjunto de widgets pré definidos de acordo com Material e iOS design e quem podem ser utilizadas na criação de interfaces.

2.2.2 Interface de Usuário declarativa

De acordo com a documentação do Flutter, para fazer a construção de telas, utiliza-se uma abordagem declarativa, onde o desenvolvedor descreve o estado atual da tela e o framework é responsável por transicionar entre os outros estados. Este difere da abordagem comumente utilizada, o estilo imperativo onde é manualmente construída uma interface e seu estado é mudado através de métodos de modificação. Considere o exemplo da Figura 4.

Figura 4 - Estrutura de interface



Fonte: Documentação Flutter

Na abordagem imperativa para fazer as mudanças mostradas na Figura 4, é necessário acessar a ViewA através de um seletor e depois invocar métodos modificadores, o Quadro 1 demonstra um exemplo de código imperativo.

Quadro 1 - Código imperativo

```
b.setColor(red)
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)
```

Fonte: Documentação Flutter.

Já na abordagem declarativa, configurações de tela (como os widgets do Flutter) são imutáveis. Para mudar a interface é necessária toda reconstrução da interface e então

ser criada uma nova árvore de componentes. O código presente no Quadro 2 exemplifica a abordagem declarativa.

Quadro 2 - Código declarativo

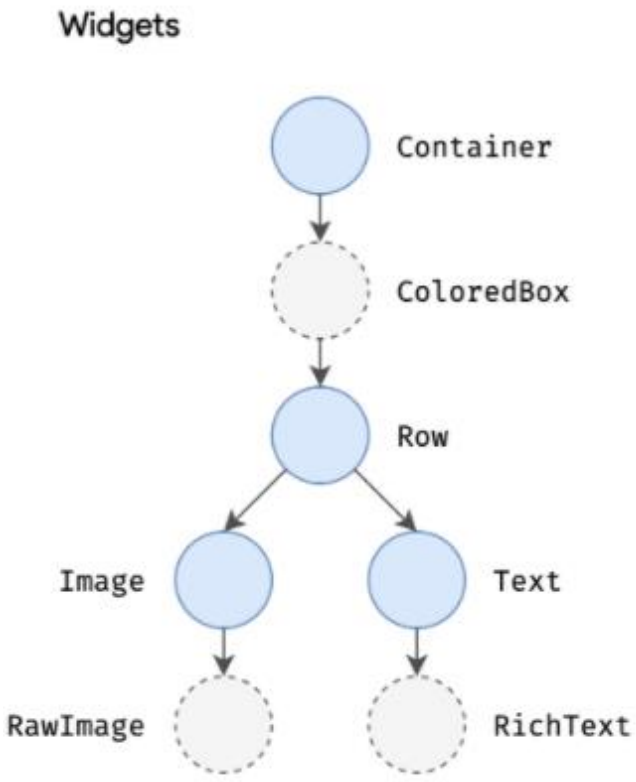
```
return ViewB(
  color: red,
  child: ViewC(...),
)
```

Fonte: Documentação Flutter.

2.2.3 Widgets

O Flutter utiliza uma abordagem de criação de interfaces de usuário declarativa, ou seja, as telas são desenvolvidas inteiramente através de linhas de código utilizando composição de widgets (unidade básica de construção do Flutter), a Figura 5 exemplifica essa composição.

Figura 5 – Composição de Widgets



Fonte: Documentação do Flutter

O código apresentado no Quadro 3 demonstra como é feita a construção de interfaces em Flutter.

Quadro 3 - Código Flutter

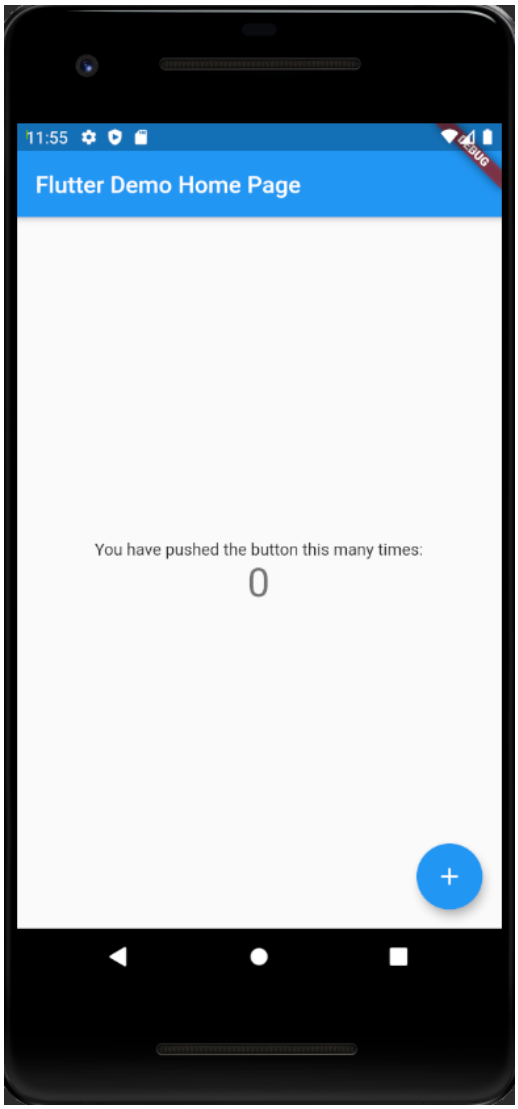
```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ), // AppBar
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'You have pushed the button this many times:',
          ), // Text
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headline4,
          ), // Text
        ], // <Widget>[]
      ), // Column
    ), // Center
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: Icon(Icons.add),
    ), // FloatingActionButton
  ); // Scaffold
}
```

Fonte: Framework Flutter

A linguagem utilizada para construção de aplicações em Flutter, e o Dart (apresentado no próximo tópico). O método build é responsável por construir a tela que será exibida ao usuário e, como dito anteriormente, a construção é feita através de composição de elementos, isso pode ser visto através da passagem consecutiva de parâmetros para os construtores dos objetos. Além da abordagem de composição o Flutter também se destaca por possibilitar o desenvolvimento de telas complexas com alta performance e também um ambiente de desenvolvimento bastante produtivo ao utilizar o hot-reload, função que possibilita que não se perca o estado atual da aplicação ao atualizar algum elemento visual. A Figura 6 demonstra a tela gerada a partir do código anterior.

Levando em consideração as características mencionadas anteriormente e o grande crescimento da tecnologia, o Flutter, juntamente com a linguagem Dart, se mostrou como uma ótima opção para o desenvolvimento da aplicação proposta por este trabalho.

Figura 6 – Tela exemplo aplicação em Flutter



Fonte: Autoria própria.

2.3 Dart

Lançado em 2011, Dart é uma linguagem de código aberto orientada a objetos desenvolvida pela Google. Inicialmente desenvolvida com o objetivo de substituir o

JavaScript como principal linguagem embutida nos navegadores web. Devido a suas características como: produtividade de desenvolvimento; orientação a objetos; alta performance e alocação rápida, fez com que a linguagem Dart tornasse a linguagem base de codificação de aplicativos utilizando o framework Flutter.

No código presente no Quadro 4, pode-se observar várias características da linguagem.

Quadro 4 - Exemplo de código Dart

```dart
// Define a function.
void printInteger(int aNumber) {
  print('The number is $aNumber.'); // Print to console.
}

// This is where the app starts executing.
void main() {
  var number = 42; // Declare and initialize a variable.
  printInteger(number); // Call a function.
}
```

Fonte: Documentação Flutter

O Dart utiliza uma sintaxe inspirada na linguagem C e tem como principais características: Tipagem forte, funções de nível de topo (não necessitando obrigatoriamente de uma classe para execução das funções), tipos genéricos, orientação a objetos, funções de primeira classe e etc.

Por utilizar soluções como AOT (Ahead Of Time), JIT (Just In Time) e permitir que não seja necessário a separação de layout declarativo como para JSX ou XML e a pequena curva de aprendizado para programadores pois utiliza características tanto de linguagens estáticas quanto de linguagens dinâmicas, a utilização da linguagem Dart no framework Flutter se tornou inquestionável. A seguir são listadas as principais características da linguagem:

- Possui uma sintaxe C-like, ou seja, a escrita do código assemelha-se muito com programas escritos em C e/ou linguagens derivadas de C;
- Paradigma orientado a objeto com alguns conceitos funcionais;
- Fortemente tipada com inferência de tipos;
- Multiplataforma.

2.4 Arquitetura de Software

Segundo a definição dada pela IEEE, pode-se dizer que: “Arquitetura de software pode ser definida como a organização fundamental de um sistema, seus componentes, as relações entre eles e o ambiente que guia seu design e evolução.” (IEEE Standard 1471).

As decisões arquitetônicas impactam não só nas atividades de desenvolvimento mas também têm relação com a manutenção, atualização e entrega e operação do software (Júnior, 2021). Dada a importância da arquitetura de software no desenvolvimento de sistemas, diversos modelos de arquitetura surgiram, dentre eles a Arquitetura Limpa.

2.4.1 Acoplamento e Coesão

Em engenharia de software, acoplamento diz respeito ao quão duas classes ou módulos são dependentes entre si. Para artefatos com baixo acoplamento, uma mudança em um componente terá pequeno impacto nos demais. Já, em artefatos com alto acoplamento, torna-se difícil a manutenibilidade, já que uma mudança pode atingir diversos artefatos do sistema (GOEDEGEBURE, 2018).

Existem diversos tipos de acoplamentos dentre eles estão: acoplamento por chamada de rotina, ocorre quando um componente utiliza de uma função (rotina) de outro componente; acoplamento de dados, ocorre quando dois componentes compartilham informações entre si via parâmetro; acoplamento de estrutura de dados, ocorre quando componentes compartilham uma mesma estrutura de dados composta e usam apenas parte dela; acoplamento de controle, ocorre quando um componente passa uma flag a outro, indicando o que fazer; acoplamento externo, acoplamento ocorre quando nosso código depende de componentes de terceiros, como frameworks, libs, bancos de dados, etc; acoplamento global, ocorre quando vários componentes compartilham um estado global; e também o acoplamento de conteúdo que ocorre quando um componente conhece, modifica ou é dependente de detalhes internos de outro componente, quebrando seu encapsulamento.

Já a coesão está diretamente ligada ao princípio da responsabilidade única, introduzido por Robert C. Martin no início dos anos 2000 e diz que um componente deve ter apenas uma única responsabilidade e fazê-la de forma satisfatória.

Existem 3 princípios fundamentais para manter a coesão de componentes (MARTIN, 2018), sendo eles: O princípio de Reuso/Lançamento equivalente (REP, na sigla em inglês); O princípio de fechamento comum (CCP); e o princípio de reuso comum (CRP).

O REP diz que as classes e módulos formadas por esse esse componente, devem pertencer à um grupo coeso, ou seja, o componente não deve ser contido de uma mistura aleatória de classes e módulos.

O CCP utiliza do princípio SRP (Single Responsibility Principle) voltado a componentes (MARTIN, 2018). Assim como o SRP diz que uma classe não deve conter múltiplas razões para mudar, então o CCP diz que um componente não deve ter múltiplas razões para mudar.

O CRP, é um princípio que visa auxiliar a tomada de decisão de quais classes e módulos devem ser colocados em um componente. Este princípio afirma que classes e módulos que tendem a ser utilizados juntos, devem ser postos no mesmo componente.

Como dito no livro O programador pragmático (HUNT e THOMAS, 2010): “Queremos projetar componentes que sejam auto suficientes: independentes e com uma finalidade exclusiva bem definida”. Conceitos como acoplamento e coesão tornam-se imprescindíveis.

2.4.2 Arquitetura em camadas

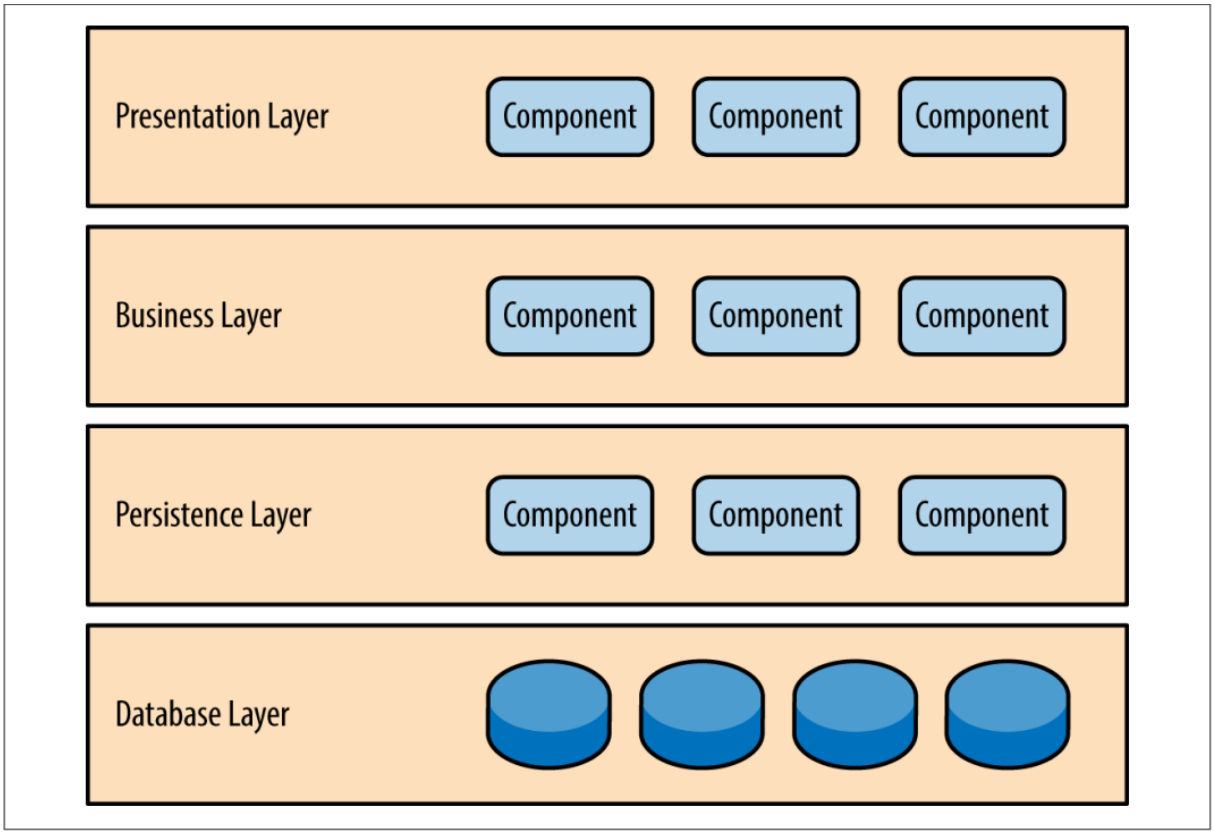
Uma arquitetura em camadas tem seus componentes organizados em camadas horizontais onde cada uma é responsável por executar um papel específico na aplicação, sendo ele apresentação de conteúdo ou lógica de negócio (RICHARDS, 2015).

O padrão de arquitetura em camadas não especifica um número exato de camadas, entretanto a maioria das aplicações consistem em 4 camadas, sendo elas: camada de apresentação (presentation), lógica de negócios (business), persistência

(persistence), e banco de dados (RICHARDS, 2015). A Figura 7 ilustra a disposição destas camadas.

Uma das características poderosas do padrão de arquitetura em camadas é a separação de responsabilidade entre os componentes. Componentes de uma camada específica lida apenas com responsabilidades daquela camada, isto faz com que o sistema seja fácil de ser desenvolvido, testado, e mantido, devido a comportamentos e escopos bem definidos (RICHARDS, 2015).

Figura 7 - Arquitetura em camadas



Fonte: Software Architecture Patterns (2015, página 2)

- **Camada de apresentação (Presentation Layer):** Esta camada é responsável por apresentar os dados aos usuários, podendo ser, por exemplo, uma interface web, e também solicitar e enviar dados para a camada de negócios;
- **Camada de negócios (Business Layer):** Esta camada é responsável por executar regras associadas ao negócio do sistema e também requisitar dados à camada de persistência e enviar dados à camada de apresentação;

- **Camada de persistência (Persistence Layer):** Esta camada é responsável por fazer a conexão com a camada de banco de dados e também fornecer dados à camada de negócios;
- **Camada de banco de dados:** Por fim, esta camada é responsável por conectar e armazenar dados em um sistema de gerenciamento de banco de dados.

Adotar uma arquitetura em camadas é um bom modo de começar um projeto, é uma forma rápida e sem muita complexidade de criar um sistema, entretanto, uma vez que o sistema cresce em escala e complexidade, logo você perceberá que ter poucas camadas com bastante código não é o suficiente, e será preciso pensar mais a fundo sobre modularização (MARTIN, 2018).

2.4.3 Arquitetura Limpa

Proposto por Robert C. Martin em seu livro Arquitetura Limpa: O guia do artesão para estrutura e design de software, em meados de 2012, este modelo de arquitetura visa ser usado no design de codificação a fim de facilitar a manutenção, testes e evolução de software através de um sistema de camadas e baixo grau de dependência.

Esta arquitetura foi derivada de outras arquiteturas já existentes, como a Arquitetura em Cebola (Jeffrey Palermo, 2008) e a Arquitetura Hexagonal (Alistair Cockburn, 2005) que compartilham de ideias similares com a arquitetura limpa.

A seguir algumas das vantagens destacadas por Robert C. Martin ao usar uma arquitetura em camadas:

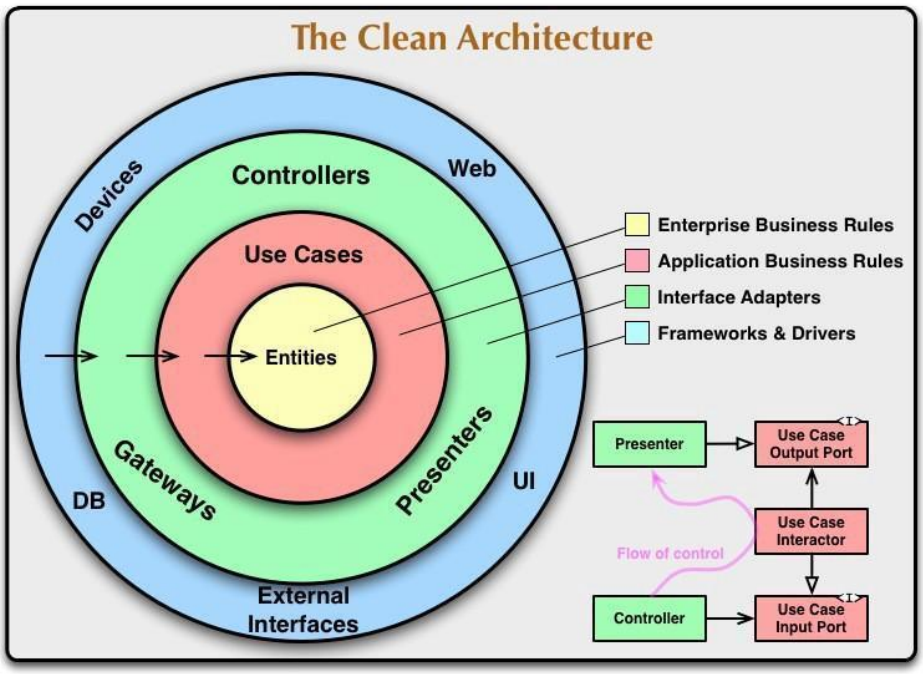
- **Independente de Frameworks:** Os frameworks devem ser vistos como ferramentas e sua arquitetura não pode se basear em um específico e se limitar à suas especificidades, fazendo assim com que seja simples uma troca de framework caso seja necessário;
- **Testabilidade:** A lógica de negócio pode ser testada sem a interface de usuário, banco de dados, servidor web, ou qualquer outro elemento externo;
- **Independente de interface de usuário:** A interface de usuário pode ser mudada facilmente sem precisar mudar o resto do sistema. Uma

interface web, por exemplo, poderia ser substituída por uma interface de linha de comando sem mudar as regras de negócio;

- **Independente do Banco de Dados:** O banco de dados pode ser trocado facilmente por qualquer outro banco de dados. As regras de negócios não estão acopladas ao banco de dados;
- **Independente de qualquer agente externo:** A lógica de negócio não deve ter conhecimento nenhum de qualquer agente externo.

Uma das principais preocupações da Arquitetura Limpa é o SoC (Separation of Concerns), princípio de engenharia de software que visa separar as preocupações, ou seja, modularizar um sistema de forma que cada parte seja responsável por resolver apenas um problema. A Figura 8 mostra como deve ser feita a divisão do sistema em camadas.

Figura 8 – Arquitetura Limpa



Fonte: Blog Robert C. Martin

Como pode ser visto a arquitetura é dividida nas seguintes camadas:

- **Entities (Entidades):** As entidades encapsulam as regras de negócios do sistema, contratos de serviços e outras funções referentes a manipulação de

dados do sistema. Elas são as menos prováveis de haver mudanças quando algo externo é modificado;

- **Casos de Uso (Use cases):** Nessa parte estão representadas ações dentro do sistema. Aqui são encapsulados e implementados todos os casos de uso que serão invocados pelas camadas externas. Estes casos de uso orquestram fluxos de dados para as regras de negócio, validação e posteriormente persistências. Uma modificação nesta camada não deve provocar modificações na camada de entidades e uma modificação externa não deve resultar em modificações nesta camada;
- **Adaptadores de Interface (Interface Adapters):** Esta camada é responsável por prover código que fará conversão do formato de dados utilizado pelas entidades e casos de usos para o formato mais conveniente para algum agente externo como banco de dados ou interfaces web.
- **Frameworks e Drivers (Frameworks & Drivers):** Ferramenta mais externa que geralmente é composta por bancos de dados, frameworks e etc. Aqui é onde o código Flutter será alocado.

A principal regra da Arquitetura Limpa é a regra de dependência mostrada pelas setas horizontais na Figura 8. As dependências das camadas acontecem de “fora para dentro”, ou seja, cada camada tem conhecimento apenas a camada um nível abaixo e nunca o inverso, isto é possível através do princípio de inversão de dependência (DIP em inglês). Utilizando dessa técnica e conceitos de orientação a objetos como interface e injeção de dependências, é possível criar um sistema independente de frameworks e de fácil manutenção e evolução.

Portanto, visando a criação de um sistema com partes bem definidas e com fácil manutenibilidade, criação de teste e modificação de tecnologias, a Arquitetura Limpa será empregada para o desenvolvimento de um caso de uso aplicando os conceitos de Arquitetura Limpa.

3 APLICAÇÃO ARQUITETURA LIMPA

Para melhor entendimento da Arquitetura Limpa e como aplicá-la em uma aplicação Flutter, nos próximos tópicos será demonstrado como pode-se definir uma organização estrutural e relação entre os componentes de modo que siga os princípios estabelecidos pela Arquitetura Limpa de Robert C.Martin.

3.1 O Caso de Uso

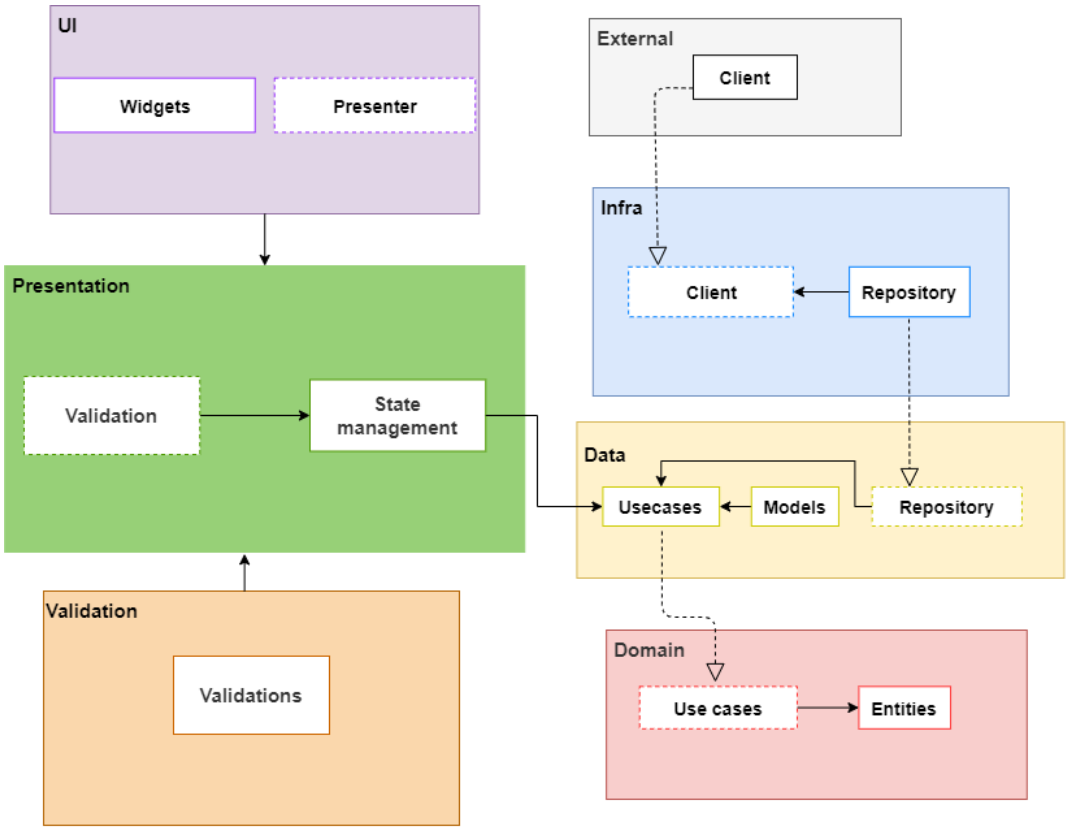
Para demonstrar uma aplicação real da Arquitetura Limpa em um projeto Flutter, foi desenvolvido um aplicativo simples com um caso de uso bastante comum: Autenticação de Usuário.

A aplicação permite que o usuário entre com e-mail e senha e faça o login. Caso este digite uma informação errada, o aplicativo mostrará uma mensagem indicando o erro cometido. Caso contrário, a aplicação exibe uma mensagem de sucesso e redireciona para a página inicial

3.2 Visão geral da arquitetura

A aplicação consistirá de 6 camadas principais, sendo elas: Domain, Data, Infra, UI, Presentation e Validation. A Figura 9 ilustra a arquitetura geral da aplicação.

Figura 9 - Visão geral da Arquitetura



Fonte: Autoria própria.

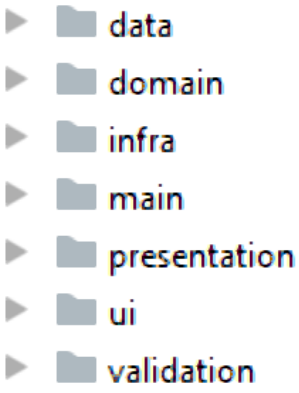
- **Domain (Domínio):** Nesta camada são definidos as entidades que serão utilizadas pelos casos de uso e também são definidos contratos dos casos de uso, esta camada se relaciona diretamente com a camada de Entidades mostrado na Figura 8;
- **Data (dados):** Esta camada é responsável por prover a implementação dos casos de usos assim como criar adaptadores que transformam dados em formatos externos para formatos compatíveis com as entidades. Aqui também serão feitas interfaces de contratos para os repositórios. Esta camada relaciona-se com a camada de Casos de Uso (Use Cases) mostrado na Figura 8, ou seja, são as regras de negócio da aplicação e adaptadores de interface;

- **Infra:** Esta camada é responsável por implementar os contratos de repositórios definidos na camada de dados. Aqui serão utilizadas bibliotecas externas como requisições web e persistência de dados. Esta camada relaciona-se com o último nível mostrado na Figura 8 (Frameworks e Drivers);
- **UI (User Interface):** Esta camada é responsável pela criação das interfaces de usuário e também da definição de contratos dos presenters e, relacionando-a com o modelo de Arquitetura Limpa, esta pertence a camada de Frameworks e Drivers;
- **Presentation:** A camada de Presentation é responsável por fazer a conexão entre a camada de dados (Data) e a interface do usuário (UI). Nesta camada dados são providos dos casos de uso para a interface e da interface para os casos de uso. Também são definidos contratos para validações de dados.
- **Validation:** Esta camada tem como funcionalidade validar informações inseridos na camada de Presentation, como: E-mail, telefone, e etc.

3.3 Estrutura de pastas do projeto

Definir a estrutura de pastas de modo que possa ser identificado rapidamente onde determinada funcionalidade deve estar, é imprescindível para uma boa arquitetura. Em um projeto Flutter onde serão aplicados os conceitos da Arquitetura Limpa, a Figura 10 ilustra uma visão geral da estruturação proposta por este trabalho.

Figura 10 - Estrutura geral de pastas no Android Studio



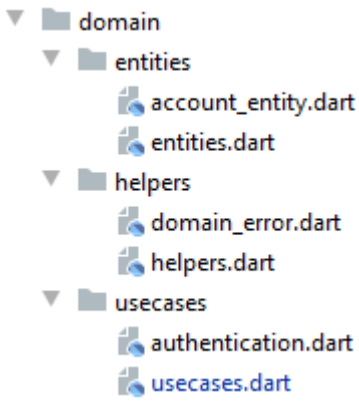
Fonte: Autoria própria.

- **data:** Nesta pasta tem-se a implementação dos casos de usos definidos na camada de domínio (domain). Nesta camada também são definidos alguns adaptadores e definição de contratos que serão implementados na camada de infra;
- **domain:** Está camada será responsável por alocar os códigos de regras de negócios empresarial, conceitos específicos da aplicação, como entidades e definição dos contratos dos casos de uso;
- **infra:** A camada de infra relaciona-se diretamente com a camada de Framework e Drivers (Figura 8), aqui serão feitas implementações concretas das abstrações definidas na camada de dados (data);
- **main:** Esta, por muitas vezes, é considerada a camada “suja” da arquitetura, pois aqui é onde se faz invocação e injeção de dependência das classes, consequentemente esta camada conhece todas as demais;
- **presentation:** Esta camada é responsável por fazer a ligação entre os casos de usos e a camada de interface de usuário, fazendo invocações de casos de uso e fornecendo dados à interface de usuário;
- **ui (User Interface):** Aqui é onde realmente o framework Flutter será utilizado pois esta camada é responsável pela criação da interface do usuário. Esta camada relaciona-se diretamente com a camada de Framework e Drivers em Arquitetura Limpa;
- **validation:** Nesta camada estão validadores que podem ser utilizados pelos presenters. Aqui são definidas validações de email, campos obrigatórios e etc.

3.4 Camada de domínio

Como dito anteriormente, essa camada relaciona-se diretamente com a camada de Entidades do modelo proposto por Robert C. Martin. Nesta camada estão as entidades utilizadas pelo nosso caso de uso autenticação. A organização de pastas nesta camada é mostrada na Figura 11.

Figura 11 - Estrutura de pastas camada domain no Android Studio



Fonte: Autoria própria.

Como pode ser visto na Figura 11, na camada de domain existem 3 subpastas, sendo elas: entities (entidades), helpers (ajudantes) e usecases (casos de usos). Cada subpasta contém um arquivo com o mesmo nome da pasta que se encontra, isto é feito para facilitar na importação e exportação de classes e interfaces. Nesta camada está definida a entidade que será usada pelo caso de uso de autenticação, esta entidade é chamada de AccountEntity, o Quadro 5 mostra o código dessa entidade.

Quadro 5 - Código AccountEntity

```
class AccountEntity extends Equatable {
  final String token;

  AccountEntity(this.token);

  @override
  List<Object> get props => [token];
}
```

Fonte: Autoria própria.

A entidade recebe uma propriedade token se o caso de uso for executado com sucesso. Já, na pasta de usecases, será definido os contratos do caso de uso autenticação, como é mostrado no Quadro 6:

Quadro 6 - Interface caso de uso autenticação.

```
abstract class Authentication {
  Future<AccountEntity> auth(String email, String password);
}
```

Fonte: Autoria própria.

O código anterior define uma classe abstrata que define os contratos a serem respeitados (valores de entrada e retorno). Portanto, como pode ser visto, o caso de uso espera dois valores de entrada e retornará a entidade definida anteriormente. Na pasta helpers são definidos os erros que podem ocorrer na execução do caso de uso.

Quadro 7 - Possíveis erros do caso de uso

```
enum DomainError {
  unexpected,
  invalidCredentials,
  emailInUse,
}
```

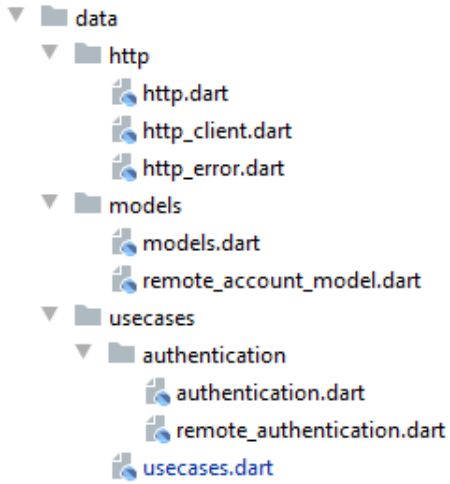
Fonte: Autoria própria

Como pode ser visto no Quadro 7, o caso uso pode gerar 3 erros: unexpected (inesperado), invalidCredentaisl (credenciais inválidas), emailInUse (email em uso). Na camada de dados os casos de uso receberão uma implementação concreta juntamente com a utilização de adaptadores.

3.5 Camada de Dados

A camada de dados relaciona-se com as camadas de Regras de Negócio da Aplicação e Adaptadores de Interface da Arquitetura Limpa pois esta é responsável por prover a implementação do caso de uso de autenticação e também adaptar o formato de dados vindos da entidade e vice versa.

Figura 12 - Estrutura de pastas camada data no Android Studio



Fonte: Autoria própria.

Dentro da pasta de usecases, estão as implementações concretas dos casos de usos definidos na camada de domínio. Para o caso de uso de autenticação é feita uma implementação de autenticação remota, o Quadro 8 mostra como pode ser feito:

Quadro 8 - Implementação caso de uso Authentication

```
class RemoteAuthentication implements Authentication {
  final HttpClient httpClient;
  final String url;

  RemoteAuthentication({@required this.httpClient, @required this.url});

  Future<AccountEntity> auth(String email, String password) async {
    try {
      final httpResponse = await httpClient.request(
        url: url,
        method: 'post',
        body: RemoteAuthenticationParams.fromDomain(email, password).toJson(),
      );

      return RemoteAccountModel.fromJson(httpResponse).toEntity();
    } on HttpError catch (error) {
      throw error == HttpError.unauthorized
          ? DomainError.invalidCredentials
          : DomainError.unexpected;
    }
  }
}
```

Fonte: Autoria própria.

Como pode ser visto, esta classe implementa o caso de uso definido na camada de domínio, logo estará respeitando os contratos definidos anteriormente. Essa classe faz uso de uma cliente http e para isso é utilizado o conceito de inversão de dependências (uns dos principais princípios da Arquitetura Limpa), onde o cliente http deverá ser fornecido através do construtor da classe (injeção de dependência) fazendo assim com que a classe tenha coesão e esteja desacoplada.

As classes RemoteAuthenticationParams e RemoteAccountModel se comportam como adaptadores de interface, convertendo dados do formato de entidade para serem utilizados remotamente e convertendo dados do formato remoto para os formatos de acordo com as entidades o Quadro 9 e Quadro 10 mostram os códigos dessas classes.

Quadro 9 - Código RemoteAuthenticationParams

```
class RemoteAuthenticationParams {
  final String email;
  final String password;

  RemoteAuthenticationParams({
    @required this.email,
    @required this.password,
  });

  factory RemoteAuthenticationParams
      .fromDomain(String email, String password) =>
      RemoteAuthenticationParams(email: email, password: password);

  Map toJson() => {'email': email, 'password': password};
}
```

Fonte: Autoria própria.

Quadro 10 - Código RemoteAccountModel

```dart
class RemoteAccountModel {
  final String accessToken;

  RemoteAccountModel(this.accessToken);

  factory RemoteAccountModel.fromJson(Map json) {
    if (!json.containsKey('accessToken')) {
      throw HttpError.invalidData;
    }
    return RemoteAccountModel(json['accessToken']);
  }
  AccountEntity toEntity() => AccountEntity(accessToken);
}
```

Fonte: Autoria própria.

Na pasta http, são definidos os contratos de clientes http e também enumerados os erros que podem ser lançados pelas classes que implementam a interface.

Os possíveis erros que podem ocorrer são:

- **Bad Request:** Quando a requisição não foi bem sucedida devido a um erro do usuário. Este erro é representado pelo código 400;
- **Not Found:** Este erro é lançado quando não for encontrado um recurso no sistema. É representado pelo código 404;
- **Server Error:** Ocorre quando há um erro interno no sistema. É retornado o código 500;
- **Unauthorized:** Ocorre quando a credencial fornecida por um usuário não é válida. Retorna o código 401;
- **Invalid Data:** Ocorre quando um dado fornecido pelo usuário não é válido para o sistema. Pode ser representado pelo código 400 ou 422;
- **Forbidden:** Ocorre quando, mesmo autenticado, um recurso não é permitido ser acessado por determinado usuário. Retorna o código 403.

Na camada de definição do cliente Http os erros são enumerados de acordo com o Quadro 11.

Quadro 11 - Possíveis erros de requisição.

```
enum HttpError {
  badRequest,
  notFound,
  serverError,
  unauthorized,
  invalidData,
  forbidden
}
```

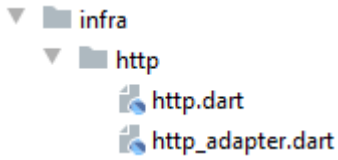
Fonte: Autoria própria.

Como pôde ser visto, esta camada aplica vários conceitos da Arquitetura Limpa, como: Desenvolvimento dos casos de uso, inversão de dependência e utilização de adaptadores de interface.

3.6 Camada de Infraestrutura

Nesta camada serão implementados o que na Arquitetura Limpa são considerados “detalhes” de arquitetura, como clientes http e bancos de dados. Na aplicação exemplo deste trabalho utilizaremos um cliente http que implementará os contratos definidos na camada de dados (data).

Figura 13 - Estrutura de pastas camada infraestrutura no Android Studio



Fonte: Autoria própria.

O Quadro 12 demonstra a implementação do contrato definido na camada de dados para clientes Http.

Quadro 12 - Implementação HttpClient

```
class HttpAdapter implements HttpClient {
  final Client client;

  HttpAdapter(this.client);

  Future<Map> request({@required String url, @required String method,Map body})
    async {
    final headers = {
      'content-type': 'application/json',
      'accept': 'application/json',
    };
    final jsonBody = body != null ? jsonEncode(body) : null;
    var response = Response('', 500);

    try {
      if (method == 'post') {
        response = await client.post(url, headers: headers, body: jsonBody);
      }
    } catch (e) {
      throw HttpError.serverError;
    }
    return _handleResponse(response);
  }

  Map _handleResponse(Response response) {...}
}
```

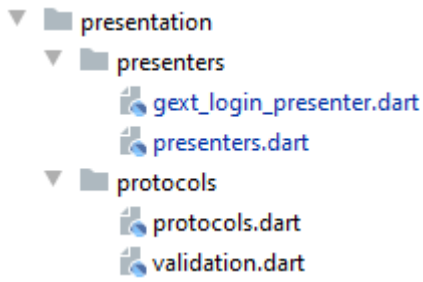
Fonte: Autoria própria.

Como pode ser visto a classe implementa a classe HttpClient definida anteriormente e provê uma implementação concreta utilizando uma biblioteca externa para o método *request*. Dessa forma, caso seja necessária uma mudança na forma de fazer requisições, será preciso apenas definir uma classe que implemente corretamente a interface HttpClient.

3.7 Camada de Presentation

Esta camada se associa a camada de Adaptadores de Interface apresentada no diagrama da Arquitetura Limpa e será responsável por interligar a camada de regras de negócio da aplicação à camada de interface do usuário.

Figura 14 - Estrutura de pastas camada presentation no Android Studio



Fonte: Autoria própria.

Para o gerenciamento de estado e roteamento, foi utilizado um micro framework do ecossistema Flutter, o GetX um pacote que, como é dito em sua documentação “combina um gerenciador de estado de alta performance, injeção de dependência inteligente e gerenciamento de rotas de uma forma rápida e prática”.

No Quadro 13 abaixo é possível ver como é feita a conexão entre as camadas de validação e casos de usos. A propriedade *authentication* representa uma instanciação do caso de uso definido anteriormente na camada de dados (data). Vale notar também que o *GetxLoginPresenter* é uma implementação de *LoginPresenter*, interface que por sua vez será definida na camada de UI garantindo assim a inversão de dependência.

Quadro 13 - Implementação Login Presenter

```
class GetxLoginPresenter extends GetxController implements LoginPresenter {
  final Validation validation;
  final Authentication authentication;

  String _email;
  String _password;
  var _emailError = Rx<UIError>();
  var _passwordError = Rx<UIError>();
  var _mainError = Rx<UIError>();
  var _navigateTo = RxString();
  var _isFormValid = false.obs;
  var _isLoading = false.obs;

  Stream<UIError> get emailErrorStream => _emailError.stream;
  Stream<UIError> get passwordErrorStream => _passwordError.stream;
  Stream<UIError> get mainErrorStream => _mainError.stream;
  Stream<String> get navigateToStream => _navigateTo.stream;
  Stream<bool> get isFormValidStream => _isFormValid.stream;
  Stream<bool> get isLoadingStream => _isLoading.stream;

  GetxLoginPresenter({
    @required this.validation,
    @required this.authentication,
  });
```

Fonte: Autoria própria.

A camada presentation é responsável por invocar o caso de uso e, caso a autenticação ocorra com sucesso, navegar para a página inicial. O código a seguir demonstra como o caso de uso é executado.

Quadro 14 - Comunicação entre presenter e Caso de Uso

```
Future<void> auth() async {
  try {
    _isLoading.value = true;
    final account = await authentication.auth(_email, _password);
    _navigateTo.value = '/home';
  } on DomainError catch (error) {
    switch (error) {
      case DomainError.invalidCredentials:
        _mainError.value = UIError.invalidCredentials;
        break;
      default:
        _mainError.value = UIError.unexpected;
        break;
    }
    _isLoading.value = false;
  }
}
```

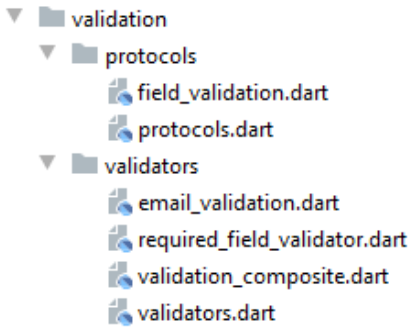
Fonte: Autoria própria.

Além da execução do caso de uso, o presenter também é responsável por tratar possíveis erros e notificar a UI, caso ocorra.

3.8 Camada de Validação (Validation)

A camada de validação agrupa todas as regras de validações e pode ser usada por um ou mais presenters, basta ser adicionada por injeção de dependência. Nesta camada foi utilizado um padrão de projeto chamado *composite*, onde o objeto de validação é formado por vários outros objetos similares de validação como validação de email e campo obrigatório.

Figura 15 - Estrutura de pastas camada validation no Android Studio



Fonte: Autoria própria.

Quadro 15 - Implementação Validation

```dart
class ValidationComposite implements Validation {
  final List<FieldValidation> validations;

  ValidationComposite(this.validations);

  @override
  ValidationError validate({@required String field, @required String value}) {
    ValidationError error;

    for (final validation in validations.where((v) => v.field == field)) {
      error = validation.validate(value);
      if (error != null) {
        return error;
      }
    }
    return error;
  }
}
```

Fonte: Autoria própria.

Como pode ser visto no Quadro 15, o *ValidationComposite* recebe um conjunto de validadores e, no método *validate* é testado se os valores são válidos de acordo com seus valores e campo. O código do Quadro 16 é um exemplo de validador de e-mail utilizado para validar a entrada do usuário.

Quadro 16 - Exemplo implementação do FieldValidation

```
class EmailValidation extends Equatable implements FieldValidation {
  final String field;

  EmailValidation(this.field);

  ValidationError validate(String value) {
    final regex = RegExp(
        r"^[a-zA-Z0-9.a-zA-Z0-9.!#$%&'*+-/=?^_`{|}~]+@[a-zA-Z0-9]+\.[a-zA-Z]+");
    final isValid = value?.isNotEmpty != true || regex.hasMatch(value);

    return isValid ? null : ValidationError.invalidField;
  }

  @override
  List<Object> get props => [field];
}
```

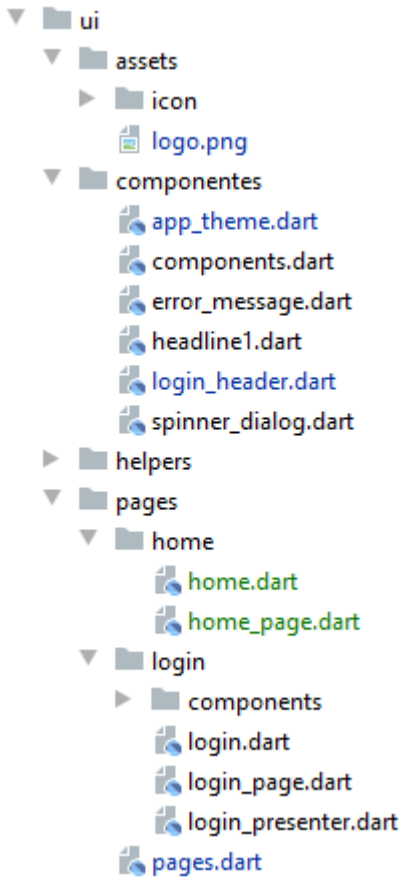
Fonte: Autoria própria.

Também foi feita uma herança da classe Equatable, biblioteca que facilita fazer comparações entre objetos da mesma classe através da comparação de um conjunto de propriedades.

3.9 Camada de Interface do Usuário

Finalmente a Interface de Usuário é onde será feito o código utilizando os componentes do Flutter. Esta camada, na Arquitetura Limpa, é tratada como “detalhe” logo, caso for necessário, pode ser facilmente substituída por outro framework.

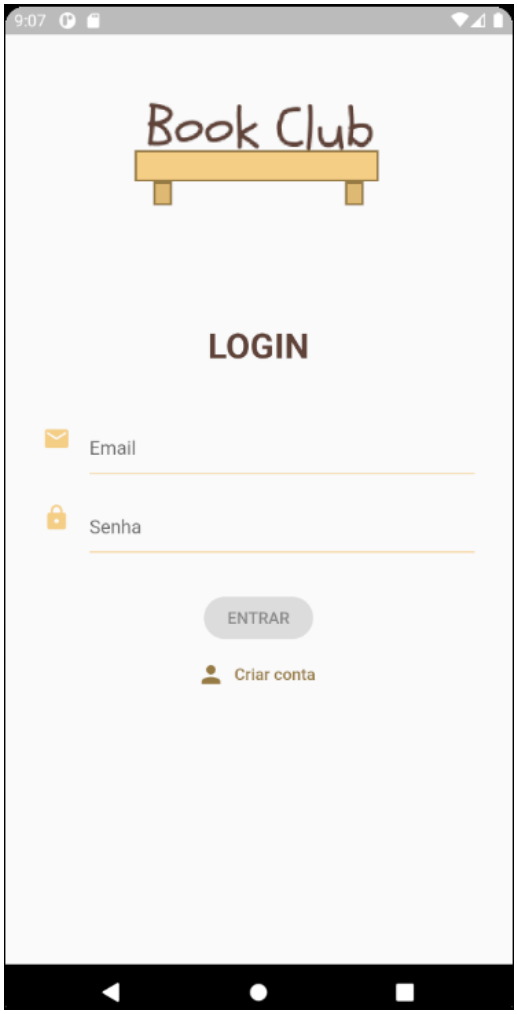
Figura 16 - Estrutura de pastas camada de UI no Android Studio



Fonte: Autoria própria.

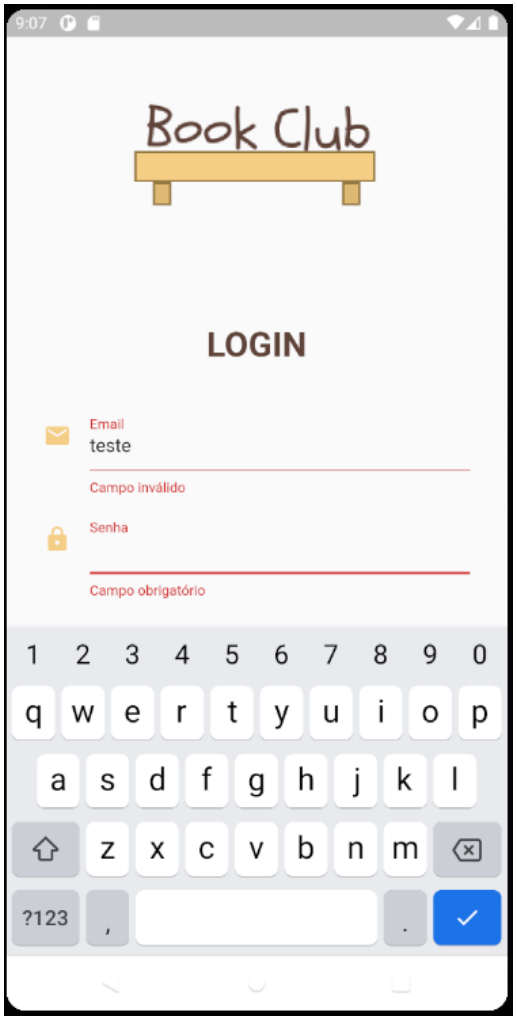
A tela desenvolvida para o caso de uso de autenticação é mostrado na Figura 17. Ao preencher os dados é feita uma validação no formato de e-mail e, caso esteja inválido, é mostrado uma mensagem de erro, isto pode ser visto na Figura 18.

Figura 17 - Página Login



Fonte: Autoria própria.

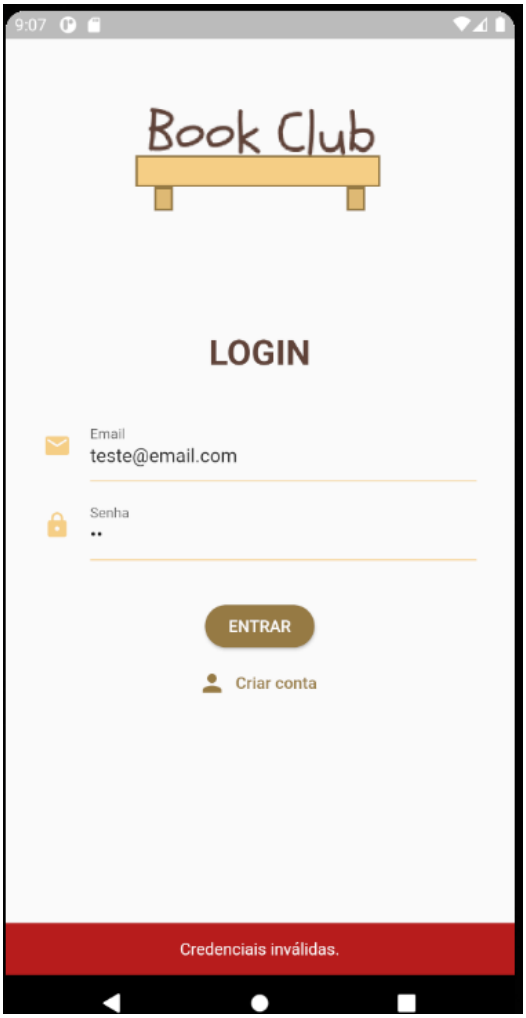
Figura 18 - Campos inválidos



Fonte: Autoria própria.

A Figura 19 mostra uma mensagem de erro quando as credenciais dadas pelo usuário são inválidas.

Figura 19 - Credenciais inválidas.



Fonte: Autoria própria.

Como dito anteriormente, a interface consegue invocar o caso de uso através da camada de presentation. O código do Quadro 17 mostra como é invocado o caso de uso ao pressionar o botão ENTRAR.

Quadro 17 - Invocação caso de uso

```
stream: presenter.isFormValidStream,
builder: (context, snapshot) {
  return RaisedButton(
    onPressed: snapshot.data == true ? presenter.auth : null,
    child: Text(R.strings.enter.toUpperCase()),
  ); // RaisedButton
},
```

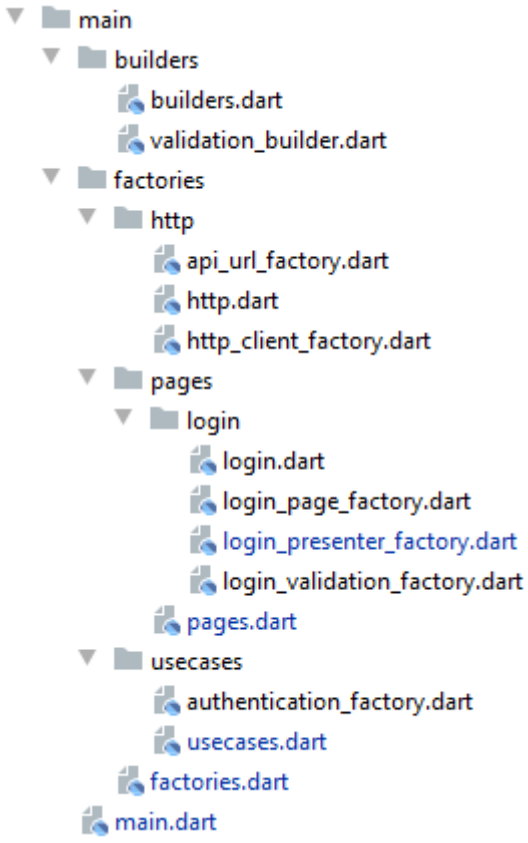
Fonte: Autoria própria.

Caso os dados estejam válidos, o método *auth* será invocado e então ocorrerá o método de autenticação.

3.10 Junção de camadas

Ao aplicarmos a Arquitetura Limpa em um projeto, obtém-se um código totalmente desacoplado e para fazer a interligação desses componentes é necessário uma camada responsável por instalar e fazer injeções de dependências. A camada main do projeto de estudo deste trabalho, é responsável por fazer toda a instanciação das classes das camadas anteriores.

Figura 20 - Estrutura de pastas camada main no Android Studio



Fonte: Autoria própria.

Para a criação das interfaces de usuário, presenters, e casos de uso, foi utilizado o padrão de projeto factory (fábrica) onde, nesta implementação, funções são responsáveis por instanciar as classes e injetar suas dependências. O código abaixo

mostra a função responsável por instanciar o caso de uso de autenticação que, neste caso, é uma autenticação remota.

Quadro 18 - Factory do caso de uso

```
Authentication makeRemoteAuthentication() {
  return RemoteAuthentication(
    httpClient: makeHttpAdapter(),
    url: makeApiUrl('login'),
  );
}
```

Fonte: Autoria própria.

Esta camada pode ser considerada a camada “suja” da arquitetura pois precisa conhecer todos os detalhes das demais.

4 CONSIDERAÇÕES FINAIS

O presente trabalho objetivou fazer um estudo de arquiteturas em camadas e aplicação em um caso de uso de autenticação utilizando a Arquitetura Limpa, cujo foco é arquitetar sistemas com camadas bem definidas, coeso e desacoplado. Utilizando desta arquitetura foi desenvolvida uma aplicação com camadas bem definidas onde o usuário é capaz de inserir suas credenciais e, em caso de sucesso, ser redirecionado para a página inicial da aplicação. A documentação do framework Flutter e livros sobre a Arquitetura Limpa forneceram todo o suporte para o desenvolvimento da aplicação e implementação da arquitetura.

A utilização do ambiente de desenvolvimento integrado Android Studio juntamente com bibliotecas do framework Flutter, fez-se suficiente para o desenvolvimento da aplicação proposta.

O estudo e entendimento de arquitetura em camadas e conceitos de coesão e acoplamento foram de suma importância para tomadas de decisões arquiteturais ao decorrer do desenvolvimento da aplicação.

Esta pesquisa necessita, ainda, de realizar avaliações a respeito de vantagens e desvantagens entre outras arquiteturas como por exemplo Arquitetura de Cebola e Arquitetura Hexagonal.

A aplicação da Arquitetura Limpa demonstrou -se viável no desenvolvimento de aplicações móveis utilizando o framework Flutter visto que este possibilita que o código do projeto seja estruturado de acordo com a necessidade do desenvolvedor, bastando respeitar os princípios arquiteturais.

REFERÊNCIAS

BANKMYCELL. **How many smartphones are in the world?** [S. L.]: Bankmycell, [20--]. Disponível em: https://www.bankmycell.com/blog/how-many-phones-are-in-the-world. Acesso em: 25 maio 2021.

CADU. **Amadurecendo com Separation Of Concerns**. [S. L.]: Devmedia, 2010. Disponível em: https://www.devmedia.com.br/amadurecendo-com-separation-of-concerns/18699. Acesso em: 25 nov. 2020.

COCKBURN, Alistair. **Hexagonal architecture**. [S. L.]: Alistair Cockburn, [20--]. Disponível em: https://alistair.cockburn.us/hexagonal-architecture/. Acesso em: 25 maio 2021.

DOCUMENTATION, Flutter. **FAQ**. [S. L.]: Flutter Documentation, [20--]. Disponível em: https://flutter.dev/docs/resources/faq. Acesso em: 25 nov. 2020.

DOCUMENTATION, Flutter. **Flutter architectural overview**. [S. L.]: Flutter Documentation, [20--]. Disponível em: https://flutter.dev/docs/resources/architectural-overview. Acesso em: 25 nov. 2020.

DOCUMENTATION, Flutter. **What technology is Flutter built with?** [S. L.]: Google, [20--]. Disponível em: https://flutter.dev/docs/resources/faq#what-technology-is-flutter-built-with. Acesso em: 25 mar. 2021.

ELEMAR JÚNIOR,. **O que é arquitetura de software**. [S. L.]: [S.I.], 2021. Disponível em: https://arquiteturadesoftware.online/introducao-v1-02/#O_que_e_arquitetura_de_software. Acesso em: 25 maio 2021.

FGV. **Brasil tem 424 milhões de dispositivos digitais em uso, revela a 31ª Pesquisa Anual do FGVcia**. [S. L.]: Fgv, 2020. Disponível em: https://portal.fgv.br/noticias/brasil-tem-424-milhoes-dispositivos-digitais-uso-revela-31a-pesquisa-anual-fgvcia. Acesso em: 25 maio 2021.

FOLDOC. **Declarative language**. [S. L.]: Foldoc, [20--]. Disponível em: http://foldoc.org/declarative%2Blanguage. Acesso em: 25 maio 2021.

FOSTAINI, Renicius Pagotto. **Clean Architecture e suas premissas**. [S. L.]: Medium, 2018. Disponível em: https://medium.com/@renicius.pagotto/clean-architecture-e-suas-premissas-6beb933c72b1. Acesso em: 25 nov. 2020.

LELER, Wm. **Why Flutter Uses Dart**. [S. L.]: Hackernoon, 2017. Disponível em: https://hackernoon.com/why-flutter-uses-dart-dd635a054ebf. Acesso em: 25 nov. 2020.

MADUREIRA, Daniel. Aplicativo nativo, web App ou aplicativo híbrido? .Net.8 mar. 2017. Disponível em: <https://usemobile.com.br/aplicativo-nativo-web-hibrido/>. Acesso em: 10
out, 2017.

MARTIN, Robert C.. **The Clean Architecture**. [S. L.]: [S.I.], 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 25 nov. 2020.

ONLINE BOOK CLUB. **Online Book Club**. [S. L.]: Online Book Club, 2020. Disponível em: https://onlinebookclub.org/. Acesso em: 25 nov. 2020.

PALERMO, Jeffrey. **The Onion Architecture : part 1**. [S. L.]: [S.I.], 2008. Disponível em: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. Acesso em: 25 maio 2021.

PAREDES, Arthur. **20 ferramentas de prototipagem, UX e usabilidade na web**. [S. L.]: Iebschool, 2019. Disponível em: https://www.iebschool.com/pt-br/blog/analitica-web/usabilidade-e-ux/20-ferramentas-de-prototipagem-e-usabilidade-na-web/. Acesso em: 25 nov. 2020.

RADEJ, Pawel. 4 pontos principais porque as aplicações híbridas são lucrativas. eCodile. 21 fev. 2017.Disponível em: <http://ecodile.com/2017/02/21/4-major-points-why-hybridapplications-are-profitable/ > Acesso em: 01 out, 2017.

RESEARCH, Grand View. **Market Analysis Report**. [S. L.]: Grandviewresearch, 2020. Disponível em: https://www.grandviewresearch.com/industry-analysis/mobile-application-market. Acesso em: 25 mar. 2021.

RICHARDS, Mark. **Software Architecture Patterns**. [S. L.]: Oreilly, 2015.

SANTO, Rafael Amadigi dal. **Acoplamento de Software**. [S. L.]: Medium, 2019. Disponível em: https://medium.com/mercos-engineering/acoplamento-de-software-db9d2ba264fd. Acesso em: 25 maio 2021.

SITE, Getx. **GetX**. [S. L.]: [S.I.], [20--]. Disponível em: https://pub.dev/packages/get. Acesso em: 25 maio 2021.

SOMBINI, Eduardo. **Jovens leem mais no Brasil, mas hábito de leitura diminui com a idade**. São Paulo: Folha, 2019. Disponível em: https://www1.folha.uol.com.br/seminariosfolha/2019/09/jovens-leem-mais-no-brasil-mas-habito-de-leitura-diminui-com-a-idade.shtml#:~:text=A%20quantidade%20anual%20m%C3%A9dia%20de,Livro%20a%20cada%20quatro%20anos. Acesso em: 25 nov. 2020.

SQUAD. **A brief Introduction to Flutter**. [S. L.]: [S.I.], 2019. Disponível em: https://www.dreamsquadgroup.com/whats-flutter-brief-introduction/. Acesso em: 25 nov. 2020.

TAQTILE. **Híbrido vs Nativo**. [S. L.]: Medium, 2016. Disponível em: https://medium.com/taqtilebr/h%C3%ADbrido-vs-nativo-c8591df0dce6. Acesso em: 25 nov. 2020.

THOMAS, Andrew Hunt David. **O Programador Pragmático de aprendiz a mestre**. [S. L.]: Bookman, 2010.

TURMINHA DO MPF. **Conheça a história do livro**. [S. L.]: Ebc, 2016. Disponível em: https://memoria.ebc.com.br/infantil/2016/02/conheca-historia-do-livro. Acesso em: 25 nov. 2020.

UNITED, Nix. **What is Flutter and Why Use Flutter for App Development**. [S. L.]: Nix United, [20--]. Disponível em: https://nix-united.com/blog/the-pros-and-cons-of-flutter-in-mobile-application-development/. Acesso em: 25 maio 2021.

WILLYAN, Deivid. **Desvendando a Arquitetura Limpa de Uncle Bob**. [S. L.]: Medium, 2020. Disponível em: https://deividchari.medium.com/desvendando-a-arquitetura-limpa-de-uncle-bob-3e60d9aa9cce. Acesso em: 25 nov. 2020.

ZARELLI, Guilherme Biff. **Descomplicando a Clean Architecture**. [S. L.]: Medium, 2020. Disponível em: https://medium.com/luizalabs/descomplicando-a-clean-architecture-cf4dfc4a1ac6. Acesso em: 25 nov. 2020.