UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL KAWAGOE GOMES MULLER

ORGANIZADOR DE ROTINAS DINÂMICO

BAURU
Janeiro/2023

RAFAEL KAWAGOE GOMES MULLER

ORGANIZADOR DE ROTINAS DINÂMICO

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.
Orientador: Profª. Drª. Simone das Graças Domingues Prado

BAURU
Janeiro/2023

M958o

Muller, Rafael Kawagoe Gomes
Organizador de rotinas dinâmico / Rafael Kawagoe Gomes Muller. -- Bauru, 2023
61 p. : il.

Trabalho de conclusão de curso (Bacharelado - Ciência da Computação) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências, Bauru
Orientadora: Simone das Graças Domingues Prado

1. Aplicativo Mobile. 2. Engenharia de Software. 3. Flutter. 4. Clean Code. 5. Clean Architecture. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências, Bauru. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Rafael Kawagoe Gomes Muller

ORGANIZADOR DE ROTINAS DINÂMICO

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Profª. Drª. Simone das Graças Domingues Prado
Orientadora
Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Profª. Drª. Márcia A Zanoli Meira e Silva
Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Profª. Me. Juliana da Costa Feitosa
Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Bauru, 16 de janeiro de 2023.

Dedico esta monografia à minha família e aos meus amigos por todo apoio que me permitiu chegar até aqui.

Agradecimentos

Agradeço primeiramente à minha família, por todos os sacrifícios e apoio que me permitiram trilhar esse caminho.

Agradeço também aos professores por todo o empenho que possuem e dedicação ao ensinar os alunos apesar de todos os desafios enfrentados para dar aula em uma universidade pública. Em especial a minha orientadora Profª. Drª. Simone das Graças Domingues Prado por não desistir de mim.

Agradeço aos ex-alunos Gustavo Rosa e João Renato Ribeiro Manesco pela construção do modelo de TCC em Latex.

E finalmente agradeço aos meus amigos, por todos os incríveis momentos que vivemos nesses anos e todo o apoio que me deram.

A computer once beat me at chess,
but it was no match for me at kick boxing.
Emo Philips

Resumo

A Agenda é uma aplicação *mobile* que permite aos usuários gerenciar e organizar seus eventos e tarefas de maneira dinâmica. Ela foi desenvolvida usando o Flutter, um popular *framework* de código aberto para a criação de aplicativos móveis *cross-platform*. Para garantir a qualidade e a manutenção da base de código, utilizou-se várias técnicas de engenharia de *software*, como boas práticas na nomenclatura de variáveis, métodos e classes, e a aplicação dos conceitos de *SOLID*. Essas práticas ajudaram a identificar e corrigir *bugs*, bem como evitar o surgimento de novos. O aplicativo de Agenda possui uma interface amigável e intuitiva ao usuário. Ele foi desenvolvido para ser usado em dispositivos Android. No geral, o desenvolvimento do aplicativo de agenda usando Flutter e técnicas de engenharia de software resultou em uma ferramenta confiável e eficaz para gerenciar tarefas, e que foi feita em um projeto escalável e de fácil manutenção.

Palavras-chave: Aplicativo *Mobile*. Engenharia de *Software*. Flutter. *Clean Code*. *Clean Architecture*.

Abstract

Agenda is a mobile application that allows users to dynamically manage and organize their events and tasks. It was developed using Flutter, a popular open source framework for creating cross-platform mobile apps. To ensure the quality and maintenance of the code base, this project was developed using various software engineering techniques, such as good practices naming variables, methods and classes, and the application of SOLID concepts. These practices helped to identify and correct bugs, as well as prevent the appearance of new ones. The Calendar application has a user-friendly and intuitive interface. It was designed to be used on Android devices. Overall, the development of the calendar app using Flutter and software engineering techniques resulted in a reliable and effective tool for managing tasks, that was made in a scalable and easy to maintain project.

Keywords: Mobile Application. Software Engineering. Flutter. Clean Code. Clean Architecture.

Lista de figuras

Lista de códigos

Lista de abreviaturas e siglas

API	*Application Programming Interface*
AVD	*Android Virtual Device Manager*
CRUD	*Create Read Update Delete*
DB	*Database*
HTTP	*Hypertext Transfer Protocol*
ID	*Identificator*
IDE	*Integrated Development Environment*
IOT	*Internet Of Things*
JSON	*JavaScript Object Notation*
JWT	*JSON Web Token*
MVC	*Model-view-controller*
MVP	*Model-view-presenter*
MVVM	*Model-view-viewmodel*
REST	*Representational State Transfer*
SQL	*Structured Query Language*
UI	*User Interface*
URL	*Uniform Resource Locator*
VS	*Visual Studio*
WEB	*World Wide Web*

Sumário

1 Introdução

Rotinas são um fator comum presente na vida das pessoas, elas facilitam a organização das tarefas diárias e otimizam a sua execução. Essa técnica de aplicação de rotinas é comumente usadas em empresas, buscando aumento da produtividade e eventualmente, dos lucros. Esse tema é discutido no artigo de (MILAGRES, 2013) onde o autor menciona o potencial da aplicação das rotinas em empresas, assim como a complexidade da aplicação das mesmas da forma mais efetiva.

De acordo com Milagres (2013), a aplicação de rotinas em empresas considera que o funcionário não é capaz de tomar as decisões ótimas, tendo em vista que possui racionalidade e recursos computacionais limitados para isso. Como forma de lidar com isso, são aplicadas regras, padrões de comportamento e rotinas geradas através de tentativa e erro para suprirem as limitações do funcionário de forma satisfatória.

1.1 Rotinas

De acordo com Milagres (2013), Rotinas podem ser definidas como a execução habitual e diária de atividades. A constante repetição dos mesmos afazeres resulta em um método padronizado, proporcionado através de uma busca consciente ou inconsciente do melhor método de fazê-las. O cotidiano das pessoas é rodeado por rotinas, tanto individuais como o ato de levantar pela manhã, tomar banho e depois comer, podendo variar de pessoa a pessoa, e também rotinas comuns, como por exemplo dirigir pela via da direita e dar seta ao virar.

Esses hábitos ajudam no convívio em sociedade, permitindo a interação entre pessoas de maneira previsível e organizada, dessa forma as atividades podem ser executadas com maior velocidade e chance de sucesso. Por exemplo, no metrô é comum a utilização de escadas rolantes da seguinte maneira, pessoas paradas ao lado direito, deixando o lado esquerdo livre pra circulação rápida. Uma pessoa que não possui o hábito de utilizar o metrô poderia ficar parada ao lado esquerdo, bloqueando a passagem e causando transtorno aos outros usuários. Nesse caso apenas um inconveniente é causado pela falta de sincronismo na execução da tarefa de utilizar a escada rolante, mas se considerar uma situação semelhante com automóveis, um acidente poderia ser causado.

1.2 Problema

Rotinas são comuns na vida das pessoas e possuem grande impacto no dia a dia, a eficiência e praticidade dessas rotinas definem se esse impacto será positivo ou negativo, por

isso empresas adotam a utilização das mesmas. Quando se considera aplicação da rotina nas empresas, existem funcionários responsáveis pelo estudo e desenvolvimento das rotinas mais adequadas, entretanto essa não é a realidade para uma pessoa que executa as rotinas por conta própria, a mesma aprendeu esses hábitos através de observação, tentativa e erro. No entanto o resultado desses fatores nem sempre é ideal.

Dessa forma, o desenvolvimento de um aplicativo que ajude na organização das rotinas facilitaria o processo e adicionaria à vida do indivíduo um maior grau de conforto e melhora de desempenho.

1.3 Justificativa

A busca por otimização das atividades é algo cada vez mais desejado pelas pessoas. Em uma sociedade onde é comum que se tenha muitas atividades para realizar em um período limitado de tempo, é desejável que a realização dessas tarefas seja a mais eficiente possível.

O desenvolvimento na forma de um aplicativo *mobile* também considera o alto índice de brasileiros que possuem acesso a um smartfone. Como mostrado em uma pesquisa divulgada pelo IBGE em 2019, 81% dos brasileiros acima de 10 anos possuem um smartfone.(IBGE, 2019)

1.4 Objetivos

1.4.1 Objetivo Geral

Desenvolvimento de um aplicativo *mobile* capaz de distribuir dinamicamente as tarefas do usuário da forma mais adequada, para que ele possa ter acesso a uma rotina organizada e efetiva sem precisar ter o conhecimento necessário para a organização da mesma. Esse aplicativo será desenvolvido utilizando flutter e técnicas de *clean code* e *clean architecture* para construção de um produto com qualidade de código e escalável. Ademais também tem como objetivo a construção de um banco SQL para armazenamento dos dados e uma API seguindo os padrões e boas práticas REST para gerenciar esses dados e disponibilizâ-los ao aplicativo.

1.4.2 Objetivos Específicos

a) Estudar desenvolvimento de aplicações *mobile*, organização de tarefas e técnicas de implementação avançadas;

b) Implementar o banco de dados e a API utilizando conceitos REST, e os algoritmos capazes de distribuir as tarefas de maneira otimizada para as necessidades do usuário; e

c) Desenvolvimento de um aplicativo *mobile*, utilizando o conhecimento adquirido, para utilização e validação da eficiência dos métodos.

1.5 Organização do trabalho

O trabalho está organizado da seguinte maneira:

- Capítulo 2: Explicação teórica sobre métodos, tecnologias e técnicas de programação utilizadas;
- Capítulo 3: Prototipação do aplicativo, API e banco de dados, e materiais específicos utilizados;
- Capítulo 4: Implementação do trabalho demonstrando a utilização das tecnologias, conceitos e materiais explicados nos Capítulos 2 e 3; e
- Capítulo 5: Conclusão do trabalho e trabalhos futuros.

2 Fundamentação Teórica

A execução deste trabalho exige conhecimento de conceitos relacionados a rotinas, otimização de tempo, automação de tarefas e Flutter, portanto, será apresentado neste capítulo estudos de rotinas em âmbito pessoal, profissional e educacional, assim como aspectos do Flutter.

2.1 O que são rotinas?

Rotinas podem ser definidas como recorrentes padrões de ações ou o equivalente a regras, estão presentes no cotidiano de todas as pessoas seja nas ações mais simples como a ordem da louça a ser lavada, até as versões mais complexas e otimizadas de rotinas de trabalho em grandes empresas como explica Milagres (2013).

As rotinas buscam de forma consciente ou inconsciente facilitar e otimizar a realização de tarefas cotidianas. O tipo de rotina que será o foco desse trabalho serão as rotinas criadas de forma consciente, as quais exigem empenho e esforço para buscar a melhor opção.

Essa visão de que as rotinas não são alcançadas sem esforço, pelo contrário, exigem um estudo e reflexão sobre o assunto, foi tratado por Giddens (1984) , no qual ele diz:

> "A rotina é baseada na tradição, costume ou hábito, mas é um grande erro supor que esses fenômenos não precisam de explicação, que são simplesmente formas repetitivas de comportamento realizadas 'sem pensar'. Pelo contrário, como Goffman (junto com a etnometodologia) ajudou a demonstrar, o caráter rotinizado da maioria das atividades sociais é algo que deve ser 'trabalhado' continuamente por aqueles que o sustentam em sua conduta diária"(GIDDENS, 1984, tradução nossa)

No trecho citado, Giddens (1984) infere que executar rotinas não é uma atividade que se realiza "sem pensar", e sim algo que exige esforço por parte daqueles que a realizam, até os problemas e situações mais cotidianos do dia a dia necessitam de análise e adaptação.

2.1.1 Características das rotinas

As rotinas podem ser separadas em diferentes características para serem explicadas, primeiramente vem os padrões. Historicamente o conceito de rotinas revela que, a noção de padrões sempre esteve no centro dessa discussão, Winter (1986) define rotina como um padrão de comportamento que é seguido repetidamente, porém pode ser alterado se as condições para essas ações mudarem. Padrões são formados por uma junção de diferentes conceitos, tal como comportamento, ação, interação e atividade. Recorrência, é uma característica chave

para as rotinas pois uma rotina é por definição algo que acontece repetidamente no cotidiano do indivíduo.

2.2 Engenharia de *software*

A engenharia de *software* é a aplicação de princípios e práticas de engenharia ao projeto, desenvolvimento, testes e manutenção de sistemas de *software*. É um campo de estudo que se concentra na criação de *software* de alta qualidade que seja confiável, de fácil manutenção e eficiente. Também envolve o uso de uma variedade de ferramentas e técnicas, tais como linguagens de programação, sistemas de controle de versão, estruturas de teste e ferramentas de gerenciamento de projetos, como explica Martin (2008). Todas essas ferramentas se unem para exterminar aquilo que é conhecido como "Código ruim".

Código ruim é um código mal escrito e difícil de entender, manter ou modificar. Pode ser difícil de ler e entender devido à falta de comentários, nomes de variáveis obscuros ou falta de formatação e indentação consistentes. Pode ser difícil de manter e modificar devido a uma lógica complexa ou não generalizada, falta de abstração e modularidade insuficiente. Em busca de sempre escrever o melhor código possível, uma leitura muito recomendada como a bíblia dos programadores é o *Clean Code* (MARTIN, 2008).

2.2.1 Código Limpo

Código limpo é o contrário do código ruim, ele é fácil de ler, entender e dar manutenção. Tem uma estrutura limpa e bem organizada e segue as melhores práticas durante a sua escrita. Um código limpo é muito mais confiável e eficiente, é fácil de entender e modificar, e também é fácil de testar e depurar.

Uma excelente definição de código limpo é a seguinte fala do criador do C++, Stroustrup (1985):

> "Eu gosto que meu código seja elegante e eficiente. A lógica deve ser simples para dificultar a ocultação de erros, as dependências mínimas para facilitar a manutenção, o tratamento de erros completo de acordo com uma estratégia articulada e o desempenho próximo do ótimo para não tentar as pessoas a fazer o código confuso com otimizações sem princípios. Código limpo faz uma coisa bem"(STROUSTRUP, 1985, tradução nossa).

Stroustrup (1985) enfatiza que o código limpo deve ter um propósito único e bem definido e não tentar fazer muito. Muitos princípios de projeto podem ser resumidos por esta idéia, e muitos escritores têm tentado transmitir esta mensagem. O mau código, por outro lado, tende a ter objetivos pouco claros ou conflitantes e é muitas vezes desordenado com detalhes desnecessários. Em contraste, o código limpo é claro e focalizado, com cada função, classe e módulo tendo um propósito específico e não-distraído. Em seguida nessa seção será

abordado algum dos pontos principais do *Clean Code* (MARTIN, 2008) quem foram utilizados no projeto Agenda.

2.2.1.1 Nomes

No desenvolvimento de *software*, os nomes são usados para uma variedade de propósitos, tais como variáveis de nomenclatura, funções, classes e diretórios. É importante escolher bons nomes, pois eles podem ter um impacto significativo na legibilidade e compreensibilidade do código. Algumas diretrizes para a criação de bons nomes incluem:

- Escolher nomes que descrevam com precisão e clareza a finalidade do item nomeado;
- Usar uma convenção de nomenclatura consistente em toda a base de código, como o uso de *camelCase* para variáveis e *PascalCase* para classes; e
- Evitar o uso de abreviações nos nomes, pois elas podem ser confusas e tornar o código mais difícil de ler.

2.2.1.1.1 Classes

É geralmente uma boa prática escolher nomes para as classes que revelam a intenção ou a finalidade da mesma, como explica Martin (2008). Isto pode tornar o código mais fácil de entender e manter, pois comunica claramente o papel da classe e seu uso pretendido. Boas praticas na escolha de nomes de classes são:

- Escolher nomes que descrevam com precisão e clareza o objetivo da aula. Por exemplo, uma classe denominada *"Client"* seria mais descritiva do que uma classe denominada *"Peoplo"*;
- Use verbos como parte do nome da classe. Por exemplo, uma classe chamada *"FileWriter"* indicaria claramente que a classe é responsável por gravar em um arquivo; e
- Use substantivos como parte do nome da classe. Por exemplo, uma classe denominada *"Employee"* indicaria claramente que a classe representa uma entidade de funcionário.

É comum dar nomes genéricos a classes semelhantes, mas isso pode causar confusão se uma das classes receber um nome mais específico enquanto a outra retiver o nome genérico original. Isso pode fazer com que a classe genérica seja erroneamente interpretada como a mais específica. É importante escolher nomes para classes que sejam claros e descritivos e ser consistente nas convenções de nomenclatura usadas em toda a base de código para evitar confusão.

2.2.1.1.2 Interfaces e implementações

Ao projetar um sistema para criar formas usando um padrão de fábrica abstrato, Martin (2008) explica que é uma boa prática dar à interface um nome simples e descritivo, como *"ShapeFactory"* e deixar de fora qualquer codificação ou prefixos como *"I"*. A implementação concreta da interface pode receber um nome mais específico, como *"ShapeFactoryImp"* ou *"CShapeFactory"*, para indicar que é a implementação concreta da interface. Codificar a interface com um prefixo ou outro identificador pode ser confuso para os usuários e deve ser evitado se possível. Em vez disso, é melhor focar na escolha de nomes claros e descritivos que reflitam com precisão o propósito e a função de cada classe no sistema.

2.2.1.1.3 Métodos

No desenvolvimento de *software*, é uma boa prática escolher nomes claros e descritivos para métodos que reflitam com precisão o propósito e o comportamento do método. Os métodos que executam uma ação devem ser nomeados usando um verbo ou frase verbal, como *"postPayment"*, *"deletePage"* ou *"save"*.

2.2.1.2 *Prefer Exceptions to Returning Error*

O uso de códigos de erro como valor de retorno de uma função de comando pode violar o princípio de separação *command-query* porque incentiva o uso de comandos como expressões nos predicados de instruções *if*. Isso pode levar a estruturas profundamente aninhadas e tornar o código mais difícil de ler e entender.

Em vez de retornar códigos de erro, geralmente é melhor usar exceções para lidar com erros. Isso permite que o código de processamento de erro seja separado do código *"happy path"* e pode tornar o código mais simples e fácil de ler.

2.2.1.3 Explique-se no código

É verdade que o código nem sempre é o melhor meio de explicação, principalmente quando os conceitos discutidos são complexos ou abstratos. No entanto, isso não significa que o código nunca deva ser usado como meio de explicação. Na verdade, o código pode ser uma maneira extremamente eficaz de ilustrar um ponto ou transmitir uma ideia. No Código 1 é demonstrado um exemplo onde a explicação do código se encontra em um comentário. E no Código 2 em contraponto é mostrado um exemplo onde a nomenclatura do código é autoexplicativa.

Código 1 – Exemplo de código utilizando comentários

```
1    //Verificar se todos os campos do login foram preenchidos
2    //e nao sao nulos
3    if (username!=null && username!=''
```

```
4     && password!=null && password!='')
```

Código 2 – Exemplo de código com nomenclatura autoexplicativa

```
1     if (user.isFilled())
```

2.2.1.4 Prevendo mudanças

As necessidades de um sistema podem mudar ao longo do tempo, o que pode exigir alterações no código. Na programação orientada a objetos, é uma boa prática usar classes e interfaces abstratas para representar conceitos e isolar o impacto de detalhes concretos de implementação. Isso pode tornar o sistema mais flexível e mais fácil de modificar e manter.

As dependências de detalhes de implementação concretos também podem dificultar o teste de um sistema. Por exemplo, se uma classe *Events* depende de uma API *GoogleEvents* externa para obter seu valor, a volatilidade da API pode dificultar a gravação de casos de teste confiáveis. Para atenuar esse problema, é uma boa ideia usar abstrações e depender de interfaces em vez de detalhes concretos. Isso pode facilitar a gravação de casos de teste isolados de dependências externas e mais estáveis ao longo do tempo.

Em vez de projetar o *Events* diretamente para depender do *GoogleEvents*, podemos criar uma interface chamada *ExternalEvents* que inclui um único método, como demonstrado no Código 3. Isso permite que o *Events* dependa da interface em vez de uma implementação específica, o que pode aumentar a flexibilidade e tornar o *design* mais modular.

Código 3 – Interface ExternalEvents

```
1
2 public interface ExternalEvents {
3  EventModel getEvent(DateTime date);
4 }
```

O ideal seria projetar *GoogleEvents* para implementar a interface *ExternalEvents*. Isso permite que *GoogleEvents* seja usado como um *ExternalEvents* na classe *Events*, como demonstrado no Código 4. Também podemos modificar o construtor de *Events* para usar uma referência a um objeto *ExternalEvents* como um argumento, em vez de depender diretamente da *GoogleEvents*:

Código 4 – Classe Events

```
1
2 public Events {
3     private ExternalEvents externalEvents;
4     public Events(ExternalEvents externalEvents){
5        this.externalEvents = externalEvents;
6     }
7     // ...
8 }
```

2.3 Arquitetura Limpa

Um fator muito importante no desenvolvimento de um projeto é o planejamento da arquitetura, tudo deve ser pensado de forma a garantir fácil visualização, acesso e manutenção. Em busca de alcançar esses objetivos para este trabalho foi utilizado de referência o livro *Clean Architecture* (MARTIN, 2017).

"Clean Architecture" é uma filosofia de *design* de *software* e um conjunto de diretrizes para a criação de sistemas de *software* escaláveis e sustentáveis. Foi introduzido por Martin (2017), em seu livro de mesmo nome.

No centro da *Clean Architecture* está a ideia de separar os vários componentes de um sistema de *software* em camadas distintas, cada uma com uma finalidade específica. As camadas externas dependem das camadas internas, mas as camadas internas não têm conhecimento das camadas externas. Essa separação de preocupações permite maior flexibilidade e modificação mais fácil do sistema. O principal *Design* de sistema explicado no livro de Martin e utilizado nesse projeto é o *SOLID*.

2.3.1 S.O.L.I.D.

SOLID é um acrônimo para cinco princípios de *design* orientado a objetos que, quando aplicados à sua base de código, podem ajudá-lo a criar um *software* mais sustentável e escalável, como explica Martin (2017). Os princípios são:

- Princípio da Responsabilidade Única: Uma classe deve ter apenas um motivo para mudar;
- Princípio Aberto-Fechado: Entidades de *software* (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação;
- Princípio da Substituição de Liskov: Os subtipos devem ser substituíveis por seus tipos básicos;
- Princípio de segregação de interface: Os clientes não devem ser forçados a depender de interfaces que não usam; e
- Princípio de Inversão de Dependência: Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

2.4 Dart

Dart é a linguagem de desenvolvimento utilizada como base do Flutter. Ela foi desenvolvida pensando na otimização do desenvolvimento das interfaces de usuário, como explica FLUTTER Team (2022). Ela possui funções *async* bastante completas para a implementação de telas programadas pensando em eventos.

Também possui *features* bastante interessantes como o *spread operator*, que permite através de uma sintaxe simples a inserção de múltiplos valores em uma biblioteca, e o *sound null safety*, criado para evitar erros não intencionais causados pela inserção de valores nulos que podem quebrar a execução do aplicativo.

2.5 Flutter

De acordo com FLUTTER Team (2022), Flutter é uma estrutura de desenvolvimento de aplicativos móveis de código aberto criada pelo Google. Ele permite que os desenvolvedores criem aplicativos compilados nativamente para dispositivos móveis, web e desktop a partir de uma única base de código usando a linguagem de programação Dart. O Flutter ganhou popularidade entre os desenvolvedores devido ao seu rápido ciclo de desenvolvimento e capacidade de criar aplicativos visualmente atraentes e de alta qualidade.

Uma característica chave do Flutter é o *Hot Reload*, que permite aos programadores fazer alterações ao seu código e ver imediatamente os resultados na aplicação. Isso permite a rápida iteração e teste de novas funcionalidades e melhorias. Flutter tem também uma rica biblioteca de *widgets* e ferramentas personalizáveis para a construção de interfaces de usuário bonitas e intuitivas, tornando-a uma escolha forte para a criação de aplicações visualmente atrativas.

Além de seu rápido ciclo de desenvolvimento e impressionante IU, Flutter também ostenta um bom desempenho. Os aplicativos Flutter são compilados em código nativo para a plataforma em que estão sendo executados, o que significa que eles têm o mesmo desempenho que os aplicativos construídos com linguagens de desenvolvimento nativas.

O Flutter foi adotado por uma série de empresas conhecidas por suas necessidades de desenvolvimento de aplicativos, incluindo *Alibaba*, *Hamilton Musical* e *Reflectly*. Também tem sido utilizado para construir aplicativos para ambas plataformas *Android* e *iOS*, tornando-o uma escolha versátil para o desenvolvimento de aplicativos multi-plataforma.

Em geral, Flutter é uma estrutura poderosa e eficiente para construir aplicações compiladas nativamente para uma variedade de plataformas. Seu rápido ciclo de desenvolvimento, suas capacidades expressivas de IU e seu bom desempenho o tornam uma escolha atraente para os desenvolvedores.

2.5.1 *Widgets*

Os *widgets* Flutter são os blocos de construção de um aplicativo Flutter. Eles são elementos pré-desenhados que um desenvolvedor pode usar para compor uma interface de usuário. Flutter vem com um grande número de *widgets* embutidos, e também permite aos desenvolvedores criar seus próprios *widgets* personalizados.

Os *widgets* do Flutter podem ser classificados em dois tipos: *stateful* e *stateless*. Os *stateful widgets* mantêm o estado e podem mudar com o tempo, enquanto os *stateless widgets* são imutáveis e não mudam.

Alguns exemplos de *widgets* Flutter embutidos incluem:

- Text: Um *widget* para exibição de texto;
- Button: Um *widget* para criar um botão que o usuário pode apertar;
- Image: Um *widget* para exibir uma imagem; e
- Container: Um *widget* para segurar outros *widgets* e aplicar efeitos visuais.

Widgets são hierárquicos, o que significa que um *widget* pode conter outros *widgets*. Isto permite a criação de interfaces de usuário complexas através da composição conjunta de *widgets* mais simples.

Além de seu uso na construção da interface do usuário, os *widgets* Flutter também lidam com a entrada e os gestos do usuário, bem como fornecem suporte para recursos de acessibilidade, como leitores de tela.

Em geral, os *widgets* Flutter são uma parte central da estrutura Flutter e são cruciais para criar interfaces de usuário eficientes e visualmente atraentes.

2.5.2 Dot Net 6.0

Dot NET 6 é definido pela Microsoft (2021) como uma estrutura livre, de código aberto e multiplataforma para a construção de aplicações modernas. É desenvolvido e mantido pela Microsoft, e inclui uma ampla gama de ferramentas e bibliotecas para a construção de aplicações para a web, móvel, desktop, nuvem e Internet das Coisas (IoT).

De acordo com um estudo da Microsoft (2021), o .NET 6 foi projetado para ser flexível e adaptável, com suporte a uma ampla gama de linguagens e plataformas de programação. Ele inclui um conjunto comum de APIs que podem ser utilizadas em diferentes plataformas, facilitando a construção de aplicações multiplataforma. Além disso, .NET 6 inclui uma série de melhorias de desempenho e novos recursos, tais como suporte aprimorado para montagem web e integração nativa, que facilitam a construção de aplicações de alto desempenho.

2.5.3 C#

C# é uma linguagem de programação poderosa, orientada a objetos, desenvolvida pela Microsoft. É utilizada para construir uma ampla gama de aplicações, incluindo aplicações web, móveis, desktop e nuvem, assim como jogos e outros *softwares*. O C# faz parte da estrutura .NET, que fornece um ambiente de tempo de execução, um conjunto de bibliotecas e um

conjunto de padrões de interoperabilidade de linguagem para construir e executar aplicações. A estrutura .NET suporta vários idiomas, incluindo C#, F# e *Visual Basic*, e tem como alvo múltiplas plataformas, incluindo Windows, macOS, Linux, iOS e Android.

2.5.4 *Migrations*

As *migrations* podem ser usadas para gerar um banco de dados, ou para fazer alterações em um banco de dados existente. Normalmente consistem de uma série de instruções que especificam o esquema do banco de dados, tais como nomes e tipos de dados de tabelas e colunas, assim como quaisquer dados que devem ser pré-populados no banco de dados. As *migrations* também podem incluir instruções para modificar o esquema do banco de dados, tais como adicionar ou apagar tabelas ou colunas, ou alterar os tipos de dados das colunas existentes.

Para gerar um banco de dados usando *migrations*, os desenvolvedores podem escrever uma série de scripts de migração que definem a estrutura e o conteúdo do banco de dados, e então usar uma ferramenta de linha de comando ou outra ferramenta para executar as *migrations*. Isto pode ser feito como parte do processo de implantação da aplicação, ou em qualquer outro momento durante o desenvolvimento ou manutenção da aplicação.

De acordo com um estudo de Kaur, Kaur e Singh (2019), o uso de *migrations* para gerar um banco de dados pode ter uma série de benefícios. Primeiro, pode ajudar a garantir que o esquema do banco de dados seja consistente e atualizado em diferentes ambientes, tais como desenvolvimento, preparação e produção. Isto pode ajudar a reduzir o risco de erros e inconsistências causados por atualizações manuais do banco de dados.

2.5.5 SQL Server

O Microsoft SQL Server é um Sistema de Gerenciamento de Banco de Dados (SGBD) projetado para lidar com uma ampla gama de requisitos de armazenamento e processamento de dados. De acordo com um estudo de Liu, Jia e Li (2008), o *SQL Server* é construído sobre uma arquitetura de Distema de Derenciamento de Banco de Dados Relacional (RDBMS), o que significa que ele armazena dados em um formato estruturado e tabular e usa a *Structured Query Language* (SQL) para manipular e consultar esses dados. O *SQL Server* é amplamente utilizado em uma variedade de contextos, incluindo negócios, governo e educação.

O *SQL Server* oferece uma gama de recursos e capacidades para ajudar a gerenciar e otimizar o armazenamento e processamento de dados, incluindo suporte para grandes volumes de dados, alta disponibilidade e escalabilidade, e recursos robustos de segurança e conformidade. Além de suas principais capacidades de gerenciamento de banco de dados, o *SQL Server* inclui ferramentas e tecnologias para integração de dados, inteligência comercial e analítica, que podem ajudar os usuários a extrair insights de seus dados e construir aplicações personalizadas

orientadas a dados.

2.5.6 REST

REST (*Representational State Transfer*) é um estilo arquitetônico de *software* que define um conjunto de diretrizes para a criação de APIs da web. De acordo com Fielding (2000), um dos principais autores do estilo arquitetônico REST. Ele é definido por quatro restrições de interface: identificação de recursos; manipulação de recursos através de representações; mensagens auto-descritivas; e, hipermídia como o motor do estado da aplicação. Estas restrições ajudam a garantir que as APIs REST sejam escaláveis, flexíveis e fáceis de usar.

As APIs REST são construídas em torno de um conjunto de pontos finais, que são URLs que representam recursos específicos. Cada *endpoint* está associado a um verbo HTTP específico, como GET, POST, PUT ou DELETE, que especifica a ação desejada. Para fazer uma solicitação a uma API REST, uma aplicação cliente envia uma solicitação HTTP para um *endpoint* específico, com o verbo apropriado e quaisquer parâmetros necessários. O servidor então responde com uma resposta HTTP, que inclui um código de status e quaisquer dados relevantes.

REST é amplamente adotado porque é simples, flexível e fácil de usar. Ele é baseado em protocolos e tecnologias padrão, como HTTP e URLs, que são amplamente suportados e bem compreendidos. Além disso, as APIs REST são fáceis de testar e depurar, e são bem adequadas para uso em uma variedade de contextos, incluindo aplicações web, móveis e IoT.

3 Metodologia

Neste capítulo é apresentada a metodologia utilizada para o desenvolvimento do aplicativo Agenda, utilizando o *framework* Flutter em conjunto com técnicas de *Clean Code* e *Clean Architecture*.

3.1 Aplicativo

O aplicativo Agenda é um aplicativo móvel que ajuda os usuários a gerenciar seus horários e compromissos. Ele permite aos usuários criar eventos, definir lembretes e visualizar sua agenda em um formato de calendário ou lista. Ele foi planejado utilizando o Figma e desenvolvido utilizando o *framework* Flutter em conjunto com a IDE VScode, e o emulador de dispositivos móveis do *Android Studio*.

3.1.1 Figma

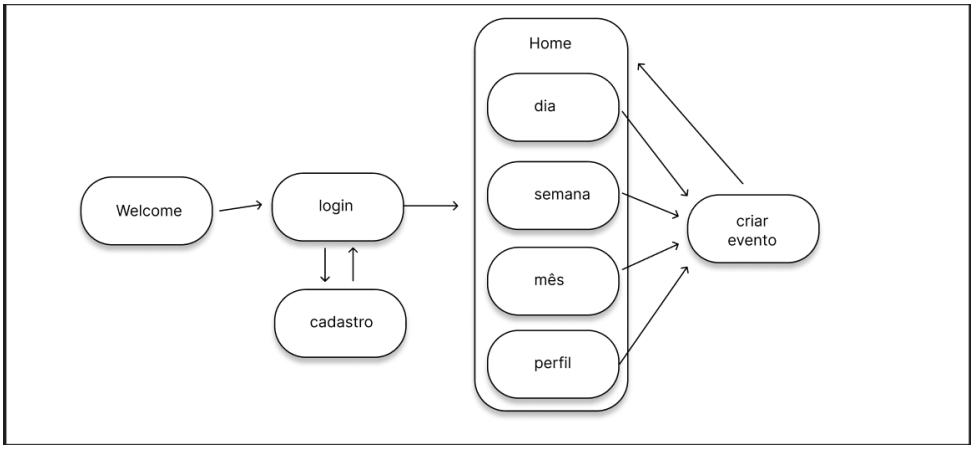
Figma é uma ferramenta em nuvem de desenho e prototipagem utilizada para criar interfaces de usuário. Ela permite que as equipes colaborem no processo de design em tempo real. Esta ferramenta é utilizada principalmente por designers e equipes de produtos para a criação e protótipo de interfaces de usuário para aplicações web e móveis. Está disponível para uso no desktop e aplicação web, tornando-o acessível a partir de qualquer dispositivo com conexão à internet.

Também em conjunto com o Figma foram utilizados dois *plugins* para fazer o design das telas, o *Material Icons Design*, capaz de gerar ícones utilizando a biblioteca de ícones *Material Icons*, e o *Calendar Generator*, utilizado para gerar o *mock* de um calendário.

3.1.1.1 Prototipação do fluxo

O fluxo foi pensando para ser o mais intuitivo possível ao usuário, começando pelo *Login* onde o usuário poderá acessar a sua conta ou criar uma conta nova caso ainda não possua. Ao realizar o login ele será introduzido a *home* onde através do conceito de *Tab*, uma barra que se apresenta com ícones dando acesso rápido a cada página, ele poderá acessar seu perfil e três formas diferentes de visualizar a sua agenda. Dentro de cada uma das páginas o usuário também terá acesso a um botão que o direcionará até a página de criar evento, onde ele poderá adicionar eventos ao seu calendário. Este fluxo pode ser visualizado na 1.

Figura 1 – Diagrama do fluxo do aplicativo Agenda.



Fonte: Elaborada pelo autor.

3.1.1.2 Prototipação do *design*

O design das páginas foi feito para que o usuário tenha uma visão limpa e organizada em cada tela, e para que consiga utilizar as *features* do aplicativo com a menor quantidade de *clicks* possíveis. Tudo foi baseado em padrões visuais tradicionalmente utilizados em aplicativos móveis.

A tela de *login* demonstrada na Figura 2 possui o nome do aplicativo centralizado e em destaque, dois campos de texto devidamente identificados como *login* e senha de usuário, um botão também em destaque para realizar a ação de *login* e um texto clicável abaixo do botão que redirecionará o usuário até o cadastro.

A *home*, vista na Figura 2, do *app* é a tela onde está listada os eventos do usuário naquele dia, através dessa tela o usuário também poderá acessar as outras telas com formas diferentes de visualizar os seus eventos, e também o seu perfil.

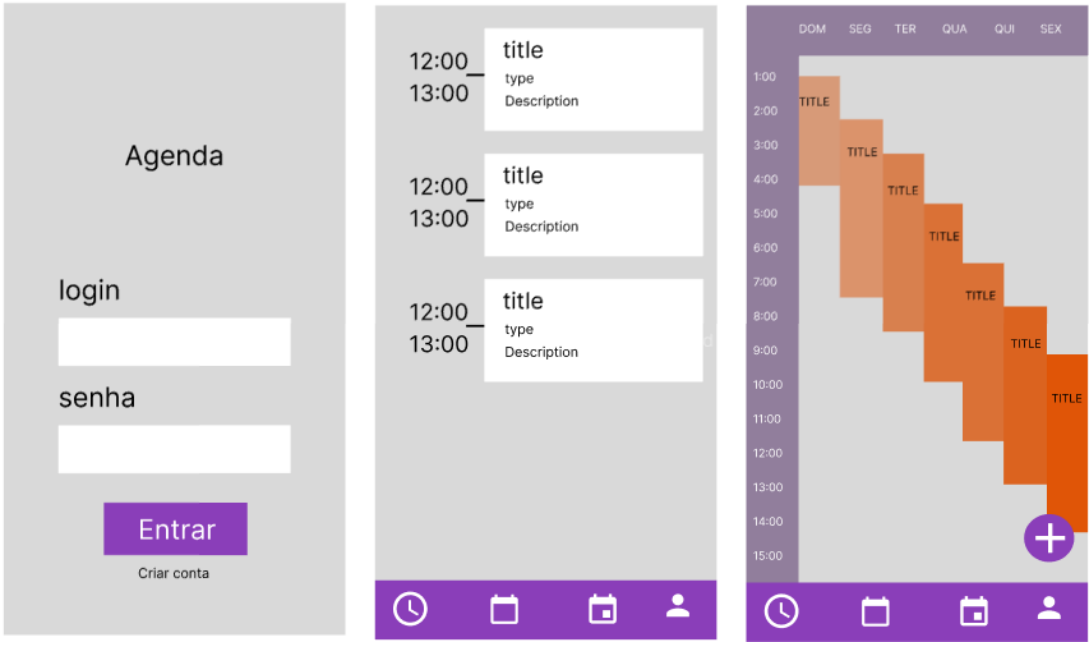
A tela semana, apresentada na Figura 2, é uma visualização em matriz de todos os eventos listados em cada dia da semana em que foi acessada, permitindo assim ao usuário uma visão bastante prática da sua rotina naquela semana.

A tela mês, apontada na Figura 3, possui um calendário onde o usuário pode selecionar uma data específica, e abaixo desse calendário será exibido uma lista referente aos eventos desse usuário no dia selecionado.

Na tela de perfil de usuário, demonstrada na Figura 3, está exibido um *placeholder* redondo e colorido com a letra inicial do nome do usuário, abaixo dele se encontra o nome do usuário, *email*, data de nascimento e um botão em destaque para realizar o *logout* do aplicativo.

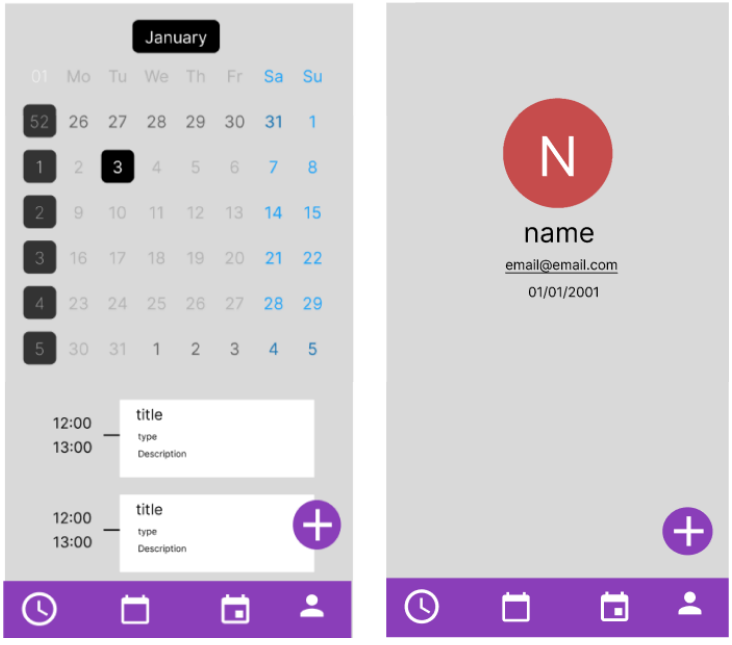
A tela de cadastro de usuário, exibida na Figura 4, é uma tela também simples, com cinco campos de entrada de texto devidamente nomeados para fácil compreensão do usuário,

Figura 2 – Tela de *login*, eventos do dia e semana, respectivamente.



Fonte: Elaborada pelo autor.

Figura 3 – Tela de eventos do mês e perfil do usuário, respectivamente.



Fonte: Elaborada pelo autor.

além de um botão em destaque para realizar a finalização do cadastro.

A tela de cadastro de evento, exibida na Figura 4, é parecida com a de cadastro de usuário, adaptando os campos a necessidade para a criação de um evento, também possui um botão em destaque para a finalizar a ação.

Figura 4 – Telas de cadastro de usuário e evento, respectivamente.

Fonte: Elaborada pelo autor.

3.1.2 *Widgets*

Os *widgets* Flutter são os blocos que compõem um aplicativo Flutter. Eles são elementos pré-desenhados que um desenvolvedor pode usar para fazer uma interface de usuário. Flutter vem com um grande número de *widgets* embutidos, e também permite aos desenvolvedores criar seus próprios *widgets* personalizados. Em seguida uma breve explicação de cada widget utilizado no aplicativo:

- *Text*: Exibição de texto;
- *Button*: Criação um botão que o usuário pode apertar;
- *Container*: Segurar outros *widgets* e aplicar efeitos visuais;
- *MaterialApp*: Envolve vários *widgets* normalmente necessários para uso do material design.
- *Scaffold*: Fornece uma estrutura base para um aplicativo. Ele inclui uma *AppBar*, uma *Drawer*, e uma *bottom navigation bar*;
- *AppBar*: Fornece uma barra de aplicação superior para um aplicativo; Normalmente inclui o título do aplicativo, ações e outros elementos, como uma *TabBar* ou uma *Drawer*;
- *Icon*: Exibe um gráfico. Ele pode ser usado para representar uma ação, um conceito ou uma peça de conteúdo;
- *Icons*: Exibe uma lista de ícones;

- *FloatingActionButton*: Botão circular que flutua acima do conteúdo de um aplicativo e é usado para realizar uma ação primária;
- *SizedBox*: Força o *widget* abaixo dele a ter um tamanho específico;
- *Padding*: Acrescenta uma margem ao *widget* abaixo dele;
- *Column*: Organiza seus filhos em uma linha vertical;
- *Row*: Organiza seus filhos em uma linha horizontal;
- *Expanded*: Ocupa o espaço livre restante em uma *Row* ou *Column*;
- *TextField*: Permite ao usuário inserir texto;
- *GestureDetector*: Oferece reconhecimento de gestos. Ele pode ser usado para detectar uma variedade de gestos, tais como uma torneira, uma prensa longa, um deslize, ou uma balança;
- *ListView*: Exibe uma lista de *widgets* em rolagem;
- *Center*: Centraliza seu filho dentro de si mesmo;
- *SingleChildScrollView*: Permite que um único filho seja rolável;
- *ListTile*: Representa um único ladrilho em uma lista;
- *Wrap*: Organiza seus filhos em um layout de embrulho;
- *CircularProgressIndicator*: Exibe o progresso de uma forma visualmente atraente;
- *SliverPersistentHeader*: Cria um cabeçalho que rola com o resto do conteúdo e permanece visível no topo do *viewport*;
- *SliverToBoxAdapter*: Integrar *widgets* sem fita em uma lista rolante de *widgets* de fita;
- *Stack*: Sobrepor vários *widgets* uns sobre os outros, com o último filho sendo desenhado em cima dos outros; e
- *Positioned*: Posicionar um *widget* filho dentro de um *Stack*.

3.1.3 Packages

Em Flutter, os *packages* são bibliotecas pré-construídas que fornecem funcionalidade adicional a um aplicativo Flutter. Eles são distribuídos através do repositório de pacotes pub.dev e podem ser facilmente adicionados a um projeto Flutter usando o gerenciador de pacotes pub.

Há centenas de pacotes disponíveis para o Flutter, desde funções utilitárias simples até kits completos de UI e integrações *backend*. Os pacotes utilizados nesse projeto foram:

- http: pacote para fazer *requests* HTTP em Flutter. Ele fornece uma API simples e fácil de usar para enviar *requests* HTTP e receber HTTP *responses*;

- *flutter_bloc*: pacote para gerenciar o estado e o fluxo de dados em um aplicativo Flutter usando o padrão de projeto BLoC (*Business Logic Component*). Ele fornece um conjunto de *widgets* e ferramentas para implementar o padrão BLoC em um aplicativo Flutter e ajuda a separar de forma limpa a camada de apresentação da lógica de negócios; e
- *shared_preferences*: pacote que fornece uma maneira simples e eficiente de salvar pequenas quantidades de dados localmente em um dispositivo. Ele permite armazenar pares de valores chave como simples *strings*, *booleans* ou *integers* e persistir entre os lançamentos de aplicativos.

3.2 Material Design

Material Design é um *design system* desenvolvido pelo Google que fornece diretrizes para o design e desenvolvimento de produtos digitais. Ele foi criado para permitir uma experiência de usuário consistente e intuitiva através de uma ampla gama de dispositivos e plataformas.

O *Material Design* é baseado na idéia de projetar cenários do mundo real e criar uma linguagem visual que seja baseada na realidade. Ele usa metáforas visuais familiares e princípios de movimento para criar um senso de profundidade e para orientar a atenção do usuário.

O Design de Materiais inclui diretrizes para layout, tipografia, cor, iconografia e outros elementos visuais, bem como diretrizes para design de interação e experiência do usuário. É utilizado pelo Google e muitas outras empresas para projetar seus produtos e interfaces de usuário.

Em Flutter, o Design de Materiais é implementado através da Biblioteca de Materiais, que fornece uma ampla gama de *widgets* de Design de Materiais e outras ferramentas para a construção de interfaces de usuário bonitas e funcionais. No projeto foi utilizado os padrões do material através da importação do mesmo em cada tela.

3.3 Android Studio

O Android Studio é um ambiente de desenvolvimento integrado (IDE) desenvolvido pelo Google para a construção de aplicativos Android. É o IDE oficial para o desenvolvimento do Android e é baseado no popular IDE IntelliJ IDEA Java.

O Android Studio fornece uma ampla gama de ferramentas e recursos para a construção, teste e depuração de aplicativos Android, incluindo:

- Uma estrutura de projeto flexível e um sistema de construção baseado em *Gradle*;
- Um rico conjunto de ferramentas de layout e design para a construção de belas interfaces de usuário;

- Um poderoso editor de código com completamento de código, refatoração e ferramentas de depuração;
- Uma gama de ferramentas para testes e aplicativos de depuração, incluindo um emulador integrado e suporte para testes em dispositivos físicos;
- Integração com sistemas de controle de versões, como Git; e
- Suporte para a construção e implantação de aplicativos na Loja do *Google Play*.

Dessas ferramentes foi utilizado exclusivamente o Emulador integrado. O emulador do Android Studio é um dispositivo virtual no computador e permite que teste os aplicativos Android sem um dispositivo físico. Ele está integrado à IDE e pode ser acessado através do *AVD Manager* (*Android Virtual Device Manager*). O emulador Android Studio é uma ferramenta conveniente para testar e depurar rapidamente os aplicativos Android durante o desenvolvimento.

3.4 Visual Studio Code

Visual Studio Code (VS Code) é um editor de código aberto e gratuito desenvolvido pela Microsoft para Windows, Linux e macOS. É um editor de código leve, projetado para desenvolvedores que precisam de uma ferramenta simples e rápida para escrever, editar e depurar código.

O VS Code inclui uma gama de recursos e ferramentas para escrever e depurar código, inclusive:

- Realce de sintaxe e completamento de código para múltiplas linguagens de programação;
- *IntelliSense*, um recurso que fornece preenchimento inteligente de código e sugestões contextuais;
- Terminal integrado e suporte para execução e depuração de código do editor;
- Depurador integrado para código de várias linguagens diferentes;
- Suporte para sistemas de controle de versão, como Git; e
- Interface personalizável e suporte para extensões para adicionar funcionalidades.

As extensões personalizadas são o diferencial para a escolha do VS Code como IDE padrão para este projeto. Ele possui extensões específicas para o desenvolvimento utilizando Flutter e Dart. Essas extensões fazem o preenchimento automático de funções em conjunto com uma explicação detalhada de cada classe disponível. São ideais para criação, edição, refatoração e recarregamento em tempo real de aplicações flutter. Além disso também foi utilizado a extensão *Dracula Official*, um tema que aplica diferentes cores às classes e funções no editor de texto facilitando a visualização do código.

3.5 Arquitetura do projeto

Nessa parte será explicado as escolhas de arquitetura e design do projeto.

3.5.1 *Design pattern*

As opções cogitadas para esse projeto foram MVC (*Model-View-Controller*), MVVM (*Model-View-ViewModel*) E MVP(*Model-View-Presenter*). A seguir uma breve explicação de cada uma delas:

***Model-View-Controller* (MVC):** é um padrão de projeto usado para separar a representação de dados da interação do usuário com ele. Ele divide uma aplicação em três componentes principais: a *model* (dados e lógica comercial), a *view* (interface do usuário) e o *controller* (link entre a *model* e a *view*);

***Model-View-ViewModel* (MVVM):** é utilizado no desenvolvimento de software para separar a camada de apresentação de uma aplicação da lógica comercial. Ele divide uma aplicação em três componentes principais: a *model* (dados e lógica comercial), a *View* (interface do usuário) e o *ViewModel* (link entre *model* e *View*). O *ViewModel* expõe dados da *model* de uma forma que a *View* possa utilizar e exibir, e lida com a lógica da IU e a entrada do usuário; e

***Model-View-Presenter* (MVP):** Ele divide uma aplicação em três componentes principais: o *model* (dados e lógica comercial), a View (interface do usuário) e o Presenter (link entre *model* e View). O Presenter é responsável por tratar as iterações do usuário e atualizar a View conforme necessário, assim como se comunicar com a *model* para recuperar e atualizar dados.

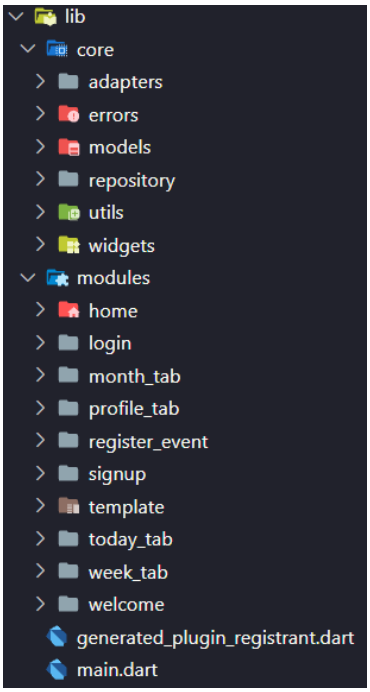
Todos os *design pattern* descritos anteriormente visam separar a camada de apresentação de uma aplicação da lógica comercial, melhorando a separação das preocupações e facilitando a manutenção e o teste de uma aplicação. O modelo escolhido foi o MVVM, pois ele se adequa muito bem ao método de controle de estado *Cubit* utilizado no projeto e que será explicado posteriormente.

3.5.2 Organização de pastas

A organização adequada de pastas é um aspecto importante do desenvolvimento de aplicativos móveis, pois ajuda a manter o projeto limpo, organizado e fácil de navegar. Um projeto bem organizado é mais fácil de trabalhar, especialmente para grandes projetos ou projetos com vários desenvolvedores, pois permite encontrar e acessar facilmente os arquivos necessários.

As pastas do projeto Agenda foram organizadas pensando na escolha do *design pattern* MVVM, explicado na seção 3.5.1. Primeiro o aplicativo foi dividido entre módulos, como por exemplo *login*, cadastro e perfil. Dessa forma qualquer ajuste necessário em área específica do aplicativo pode ser executada rapidamente pois as dependências estão concentradas na pasta do módulo. Além disso foi criada uma pasta *"core"* onde se localizam as dependências compartilhadas entre módulos. A arquitetura de pastas desse aplicativo pode ser visualizada na Figura 5.

Figura 5 – Arquitetura das pastas a nível do aplicativo.



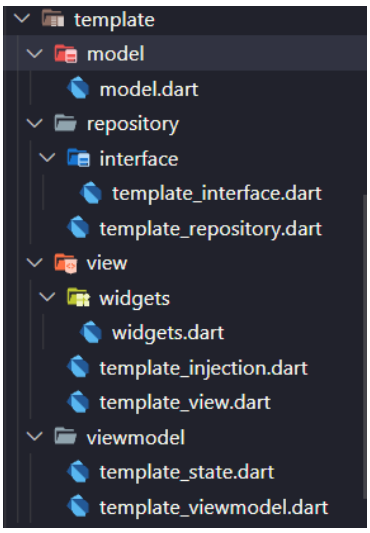
Fonte: Elaborada pelo autor.

Dentro dos módulos as pastas foram divididas pensando em isolar as responsabilidades o máximo possível e facilitar o acesso a cada componente. Foram criadas duas pastas *view* e *viewmodel* pensando no *design pattern* do projeto. Em complemento a elas foram criadas a *repository* para armazenar as *requests* à API que são solicitadas pela *viewmodel*, a *repository* também possui uma interface para esconder a implementação e apresentar para a *viewmodel* apenas os métodos. Ademais foi criada uma pasta *models* para o armazenamento das *models* utilizadas tanto pela *repository* como pela *view*. Por fim foi criada uma pasta *widgets* responsável por armazenar os *widgets* utilizados na construção da *view*, mantendo assim um código limpo e de fácil leitura, como pode ser visualizado na Figura 6.

3.6 Diagramando o Banco

De acordo com um artigo de Beaulieu (2003), há várias considerações importantes a serem consideradas ao criar um banco de dados SQL para armazenar informações sobre

Figura 6 – Arquitetura das pastas a nível de módulo.



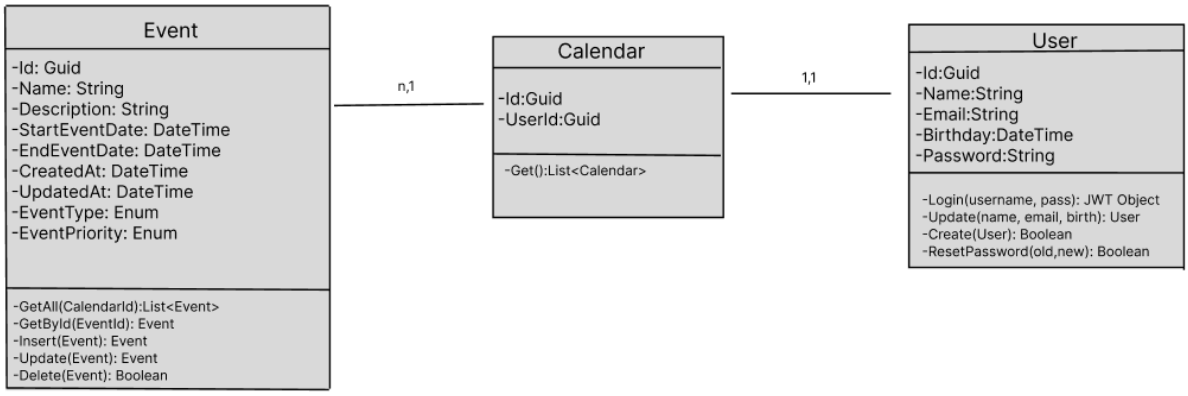
Fonte: Elaborada pelo autor.

usuários, eventos e calendários. Estas incluem modelagem de dados, tipos de dados, índices, normalização e segurança. Por exemplo, é preciso decidir como estruturar os dados para representar as relações entre usuários, eventos e calendários, e escolher os tipos de dados apropriados para cada atributo. Além disso, é importante considerar quais atributos devem ser indexados para melhorar o desempenho, e garantir que o banco de dados seja devidamente normalizado para minimizar a redundância.

Para a diagramação do banco, apresentado na Figura 7, primeiro foi considerado o que seria necessário ao usuário para acessar seu calendário e mandar as suas informações privadas. Dessa forma foi decidido que o usuário deveria ter um *Id* como *primary key*, uma *string Name* para identificação ao tratar com o usuário, uma *string Email* e Senha para a realização do login e um *Datetime Birthday* para recuperação de senha. Além disso, é necessário criar um modelo de evento que contenha todas as informações necessárias para gerenciamento do mesmo, para isso o evento deverá possuir Id como chave primaria, um nome, descrição, data, e dois tipos *enum*, um para prioridade e outro para Tipo.

Para correlacionar as entidades *user* e *event* foi necessário criar uma entidade *Calendar* com *Id* próprio e *Id* do usuário através de uma chave estrangeira. A entidade *Calendar* se relaciona com a entidade *user* em uma relação um para um, e se relaciona com a entidade *Event* em uma relação um para n, ou seja, cada calendário possui n eventos.

Figura 7 – Diagramação do banco.



Fonte: Elaborada pelo autor.

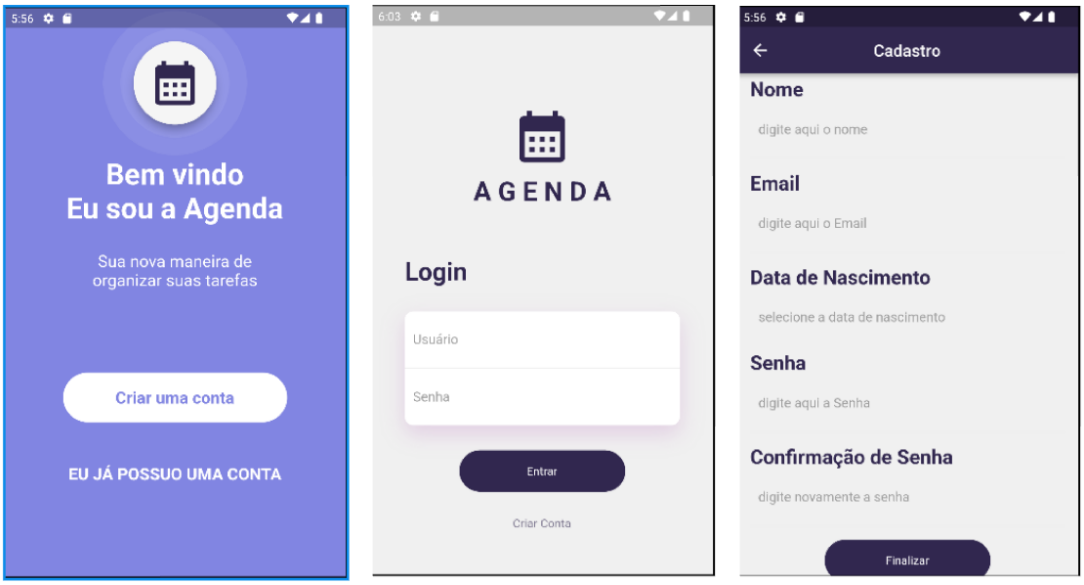
4 Desenvolvimento

O desenvolvimento deste trabalho foi divido em duas partes, primeiramento foi desenvolvido o fluxo e o *design* das telas. Na seguinte foi implementado o código em Flutter utilizando dos principais conceitos de engenharia de *software*.

4.1 Planejamento

Primeiramente foi pensado como deveria ser a parte visual do aplicativo e seu fluxo de telas. Foi considerado o fluxo mais comum dos aplicativos *mobiles* atuais, primeiro uma tela de *login*, apresentada na Figura 8, plana e simples com o nome do aplicativo, os campos para serem preenchidos e um botão em destaque para realizar o acesso, ademais um botão em forma de texto indicando um cadastro para novos usuários. Durante o desenvolvimento cogitou-se talvez uma interface mais amigável aos novos usuários, por isso foi criado uma tela de boas vindas, demonstrada na Figura 8, onde é feita uma breve apresentação sobre o aplicativo e ao final dela dois botões em destaque, um para novos usuários e outro para aqueles que já possuem uma conta cadastrada.

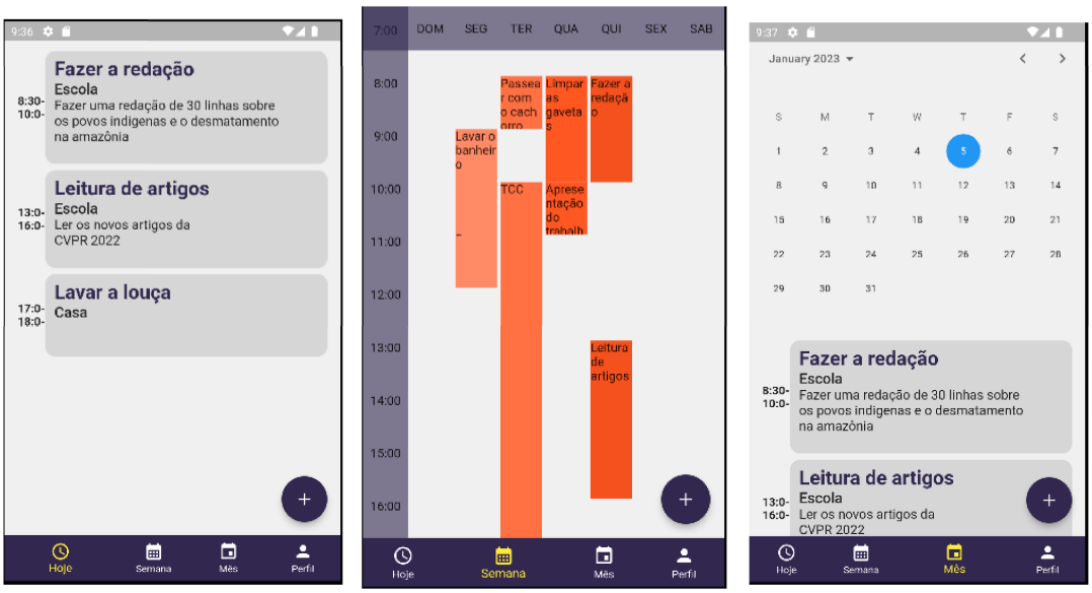
Figura 8 – Telas de Boas vindas, Login e Cadastro, respectivamente.



Fonte: Elaborada pelo autor.

Após o *login* o usuário será direcionado a tela principal do aplicativo. Para o projeto de uma Agenda a tela principal ideal é a apresentação dos eventos que o usuário possui para aquele dia. Na sequência é mais interessante ao usuário visualizar seus planos para essa semana e mês, como visto na Figura 9.

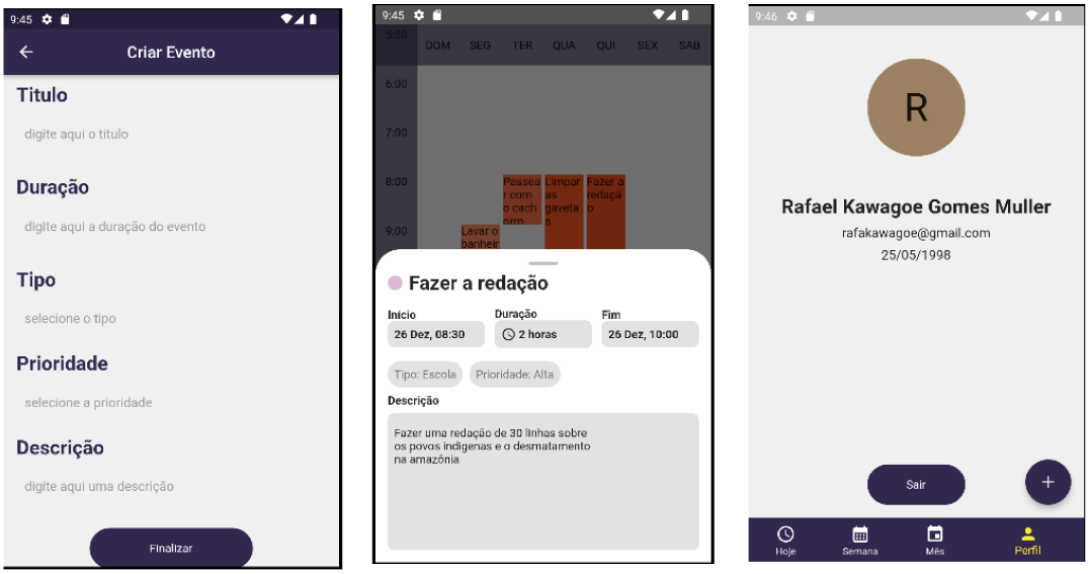
Figura 9 – Telas de eventos do dia, semana e mês, respectivamente.



Fonte: elaborada pelo autor

Para que o usuário possa adicionar novos eventos, as telas de visualização de eventos possuem um ícone no canto inferior direito. É uma técnica comum utilizada por aplicativos e bastante intuitiva para o usuário. A tela de criação de evento, demonstrada na Figura 10, é uma tela simples com os campos necessários para a inserção dos dados e um botão de finalizar em destaque.

Figura 10 – Telas de criar eventos, descrição do evento e perfil do usuário, respectivamente.



Fonte: Elaborada pelo autor.

Também é conveniente ao usuário visualizar os detalhes dos eventos que possui. Para isso basta o usuário selecionar o evento em qualquer tela que uma *BottomModal*, será exibida

com as informações, como mostra a Figura 10. Ademais para que o usuário possa ver as informações de quem está logado na sessão atual, e para que possa realizar o *logout* do aplicativo, foi criado uma tela de perfil, vista na Figura 10.

4.2 Implementando a API

A API foi feita utilizando .Net 6.0, na linguagem C#, utilizando conceitos REST, como apresentado, respectivamente, nas seções 2.5.2, 2.5.3, 2.5.6, para melhores práticas no desenvolvimento. A IDE de escolha para o desenvolvimento foi o VS Code, explicado na Seção 3.4, pois ele possui extensões que auxiliam no processo e foi utilizado no desenvolvimento do aplicativo móvel.

O padrão de design escolhido para a API foi o MVC, apresentado na Seção 3.5.1. É uma escolha popular para a construção de APIs REST porque proporciona uma separação limpa de preocupações e promove boas práticas de projeto de software. De acordo com Barry e McQuillan (2017), alguns dos benefícios de usar MVC para REST APIs incluem:

- Testabilidade: Facilitam a escrita de testes automatizados para a API, o que pode ajudar a garantir o funcionamento correto e reduzir o risco de bugs e regressões;
- Reusabilidade: Encorajam a construir componentes reutilizáveis, o que pode facilitar a construção e manutenção de APIs complexas;
- Flexibilidade: Permitem alterar a implementação da API sem afetar a interface, facilitado a adaptação às mudanças de requisitos; e
- Desempenho: Podem ajudar a melhorar o desempenho da API, permitindo otimizar componentes individuais de forma independente.

Com tudo definido o primeiro passo para começar a escrever a *API* é pela *model*. As *models* são utilizadas em todos as outras partes do desenvolvimento e também são fatores cruciais na execução das *migrations* para geração do banco. A seguir, alguns exemplos de *model*; a primeira *model* é a *User*, demonstrada no Código 5, utilizada no *login* e nas consultas, e em seguida a *model* do *Event*, visualizado no Código 6.

Código 5 – Model User feito em C#

```
1 public class User
2     {
3         [Key]
4         [Required]
5         public int Id { get; set; }
6         [Required]
7         public string Name { get; set; }
8         [Required]
```

```
9         public string Email { get; set; }
10        [Required]
11        public string Password { get; set; }
12    }
```

Código 6 – Model Event feito em C#

```
1 public class Event
2   {
3         public int ID { get; set; }
4         public string Nome { get; set; }
5         public string Description { get; set; }
6         public DateTime StartEventDate { get; set; }
7         public DateTime EndEventDate { get; set; }
8         public DateTime CreatedAt { get; set; }
9         public DateTime UpdatedAt { get; set; }
10        public EventType EventType { get; set; }
11        public EventPriority EventPriority { get; set; }
12
13        //FK with User
14        public int UserId { get; set; }
15    }
```

Em seguida foram feitas as *controllers* responsáveis por lidar com as requisições http recebidas pela API e fazer a utilização das *actions* da *model* e devolver a resposta adequada ao usuário. O Código 7 apresenta um exemplo de implementação do método de uma *controller* responsável por devolver as informações específicas de um evento cujo *Id* foi fornecido pelo usuário.

Código 7 – Método getById da Controller Event

```
1 [HttpGet("event/{id}")]
2         public async Task<ActionResult<IEnumerable<Event>>> GetById([
    FromServices] EventRepository eventRepository, int id)
3         {
4             var accessToken = Request.Headers[HeaderNames.Authorization
    ].ToString().Replace("Bearer ", "");
5             var handler = new JwtSecurityTokenHandler();
6             var t = handler.ReadJwtToken(accessToken).Payload;
7
8             var userEvents = await eventRepository.GetEventsById(id);
9
10            return Ok(userEvents);
11        }
```

Na sequência são implementadas as *Repositories*, responsáveis por abstrair os detalhes do armazenamento e recuperação de dados, permitindo escrever um código independente do

armazenamento de dados subjacente. Por fim, um exemplo de implementação do método *getEventById*, da *repository EventRepository*, apresentado no Código 8.

Código 8 – Método GetEventsById da Repository Event

```
public async Task<Event> GetEventsById(int eventId)
        {
            try
            {
                using (var ctx = new CalendarDBContext())
                {
                    Event existEvent = ctx.Events.Where(w => w.ID ==
   eventId)?.FirstOrDefault();

                    return existEvent;
                }
            }
            catch (Exception ex)
            {
                return null;
            }
        }
```

A organização dos eventos no banco é feita no momento que o usuário adiciona um evento. Na *controller* o método *CreateEvent* irá tratar a *request* feita pelo usuário, como demonstrado no Código 9.

Código 9 – Método CreateEvent da Controller Event

```
[HttpPost("event")]
        public async Task<ActionResult<Event>> CreateEvent([FromServices
   ] EventRepository eventRepository, [FromBody] CreateEvent createEvent
   )
        {
            var accessToken = Request.Headers[HeaderNames.Authorization
   ].ToString().Replace("Bearer ", "");
            var handler = new JwtSecurityTokenHandler();
            var t = handler.ReadJwtToken(accessToken).Payload;

            var createRepo = await eventRepository.CreateEvent(
   createEvent, Convert.ToUInt16(t.GetValueOrDefault("userId")));
            await eventRepository.OrganizeEvents(createRepo.EndEventDate
   , Convert.ToUInt16(t.GetValueOrDefault("userId")));

            return Ok(createRepo);
        }
```

Depois um método é responsável por recuperar os eventos que o usuário já possui cadastrado, e chamar o método *OrganizeDateEvents*, vizualizado no Código 11, para avaliar o

tipo e sua prioridade e rearranjar-los de maneira a priorizar os eventos com datas de finalização mais próximas enquanto alterna os tipos de atividade. O Código 12 mostra um exemplo.

Código 10 – Método OrganizeEvents da Repository Event

```
public async Task<List<Event>> OrganizeEvents(DateTime date,int userId)
        {
            try
            {
                using (var ctx = new CalendarDBContext())
                {
                    List<Event> eventos = ctx.Events.Where(w => w.UserId
    == userId && w.EndEventDate.Date == date.Date)?.OrderBy(ob => ob.
   EndEventDate).ThenBy(tb => tb.EventType).ToList();

                    eventos = (await OrganizeDateEvents(eventos)).
   OrderBy(ob=>ob.EndEventDate).ToList();

                    return eventos;
                }
            }
            catch (Exception ex)
            {
                return null;
            }
        }
```

Código 11 – Método OrganizeDateEvents da Repository Event

```
public async Task<List<Event>> OrganizeDateEvents(List<Event> eventos)
        {
            try
            {
                List<Event> responseList = new List<Event>();
                for (int i = 1; i < eventos.Count(); i++)
                {
                    if (eventos[i].EventType == eventos[i - 1].EventType
    && eventos[i].EndEventDate.Day == eventos[i - 1].EndEventDate.Day)
                    {
                        DateTime auxEventTimeEnd = eventos[i + 1].
   EndEventDate;
                        DateTime auxEventTimeStart = eventos[i + 1].
   StartEventDate;

                        eventos[i + 1].EndEventDate = eventos[i].
   EndEventDate;
                        eventos[i + 1].StartEventDate = eventos[i].
   StartEventDate;
                        eventos[i].EndEventDate = auxEventTimeEnd;
```

```
                        eventos[i].StartEventDate = auxEventTimeStart;
                    }
                }

                return eventos;
            }
            catch (Exception ex)
            {
                return null;
            }
        }
```

Código 12 – Método de registro de usuário em C#

```
[AllowAnonymous]
        [HttpPost("register")]
        public async Task<IActionResult> RegisterAsync([FromServices]
    UserRepository userRepository,
                                        [FromBody] NewUser newuser)
        {
            if (!ModelState.IsValid)
                return BadRequest("Body invalido");

            try
            {
                User usu = new User()
                {
                    Id = 0,
                    Name = newuser.name,
                    Email = newuser.email,
                    Password = newuser.password
                };

                var insertedUsu = await userRepository.
    CadastroDeUserAsync(usu);
                if (usu.Id == 0)
                {
                    return UnprocessableEntity(
                       new
                       {
                           status = HttpStatusCode.UnprocessableEntity,
                           Error = "Nao foi possivel adicionar usuario"
                       }
                    );
                }
                else
                {
                    return Created($"New User:", insertedUsu);
```

```
33                  }
34              }
35              catch (Exception ex)
36              {
37                  return StatusCode(500, ex.Message);
38              }
39
40          }
```

O controle de acesso foi implementado utilizando o sistema de *Web tokens JWTs*, uma forma popular de tornar segura uma *API REST*. Os JWTs são objetos JSON auto-contidos que contêm uma carga útil assinada de reivindicações. As reivindicações são partes de informações sobre um assunto (normalmente um usuário) que são codificadas no *token*. Exemplos de código de registro de usuário gerando um Id, exemplificado no Código 12, e *login* devolvendo um *token* de acesso, pode ser visualizado no Código 13:

Código 13 – Método de login em C#

```
1  [AllowAnonymous]
2          [HttpPost("login")]
3          public async Task<IActionResult> LoginAsync([FromServices]
       TokenService _tokenService,
4                                      [FromServices] UserRepository
       userRepository,
5                                      [FromBody] LoginUser loginUser)
6          {
7
8              if (!ModelState.IsValid)
9                  return BadRequest("Body invalido");
10
11             try
12             {
13                 User usu = new User()
14                 {
15                     Id = 0,
16                     Name = null,
17                     Email = loginUser.Email,
18                     Password = loginUser.Password
19                 };
20
21                 var usuConsulta = await userRepository.
       VerificarUsuarioSenhaAsync(usu);
22
23                 if (usuConsulta != null)
24                 {
25                     if (usuConsulta.Id != 0)
26                     {
```

```
27                          var token = await _tokenService.GenerateToken(
    usuConsulta);
28                          return Ok(
29                              new
30                              {
31                                  status = HttpStatusCode.OK,
32                                  Token = token
33                              });
34                      }
35                      else
36                      {
37                          return Unauthorized(
38                             new
39                             {
40                                 status = HttpStatusCode.Unauthorized,
41                                 Error = "Nao autorizado"
42                             });
43                      }
44                  }
45                  else
46                  {
47                      return NotFound(
48                          new
49                          {
50                              status = HttpStatusCode.NotFound,
51                              Error = "Nao encontrado"
52                          });
53                  }
54              }
55              catch (Exception ex)
56              {
57                  return StatusCode(500, "Falha ao realizar login.");
58              }
59          }
```

Por fim a *migration*, apresentada na Seção 2.5.4, foi o método utilizado para gerar o banco através das *models* definidas na API. Para executar uma *migration* é necessário primeiro criar uma classe de contexto, apontado no Código 14, representa uma sessão com o banco de dados. Ela é responsável por gerenciar o ciclo de vida de suas entidades e realizar operações CRUD no banco de dados. Também é nela que é feita a configuração da conexão com o banco de dados, dentro do método *OnConfiguration*.

Código 14 – Classe CalendarDBContext em c#

```
1 public class CalendarDBContext : DbContext
2   {
3     public CalendarDBContext() : base ()
4     {
```

```
5     }
6
7         public DbSet<User> Users { get; set; }
8         public DbSet<Event> Events { get; set; }
9
10        protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder)
11        {
12            var configuration = new ConfigurationBuilder()
13                .SetBasePath(Directory.GetCurrentDirectory())
14                .AddJsonFile("appsettings.json")
15                .Build();
16
17            var connectionString = configuration.GetConnectionString("
    CalendarConnection");
18            optionsBuilder.UseSqlServer(connectionString);
19        }
20    }
```

Após a realização das configurações, para executar a *migration* e gerar o banco dois comandos são necessários. Primeiro o *InitialCreate*, desmontrado no Código 15, que criará uma migração que representa o estado inicial do seu banco de dados. Em seguida será executado um *update* para aplicar a migração e criar o banco de dados, demonstrado no Código 16.

Código 15 – Código de execução da migration InitialCreate

```
1 Add-Migration InitialCreate
```

Código 16 – Código de execução da migration Database

```
1 Update-Database
```

4.3 Implementação dos *Widgets*

O *widget Text* é o *widget* padrão para exibição de texto no flutter. Devido ao alto índice de customização que este *widget* permite, e também em busca de padronizar os textos do aplicativo, foi criado um *widget CustomText* para encapsular o *Text*, como demonstrado no Código 17, assim os textos exibidos terão os estilos aplicados.

Código 17 – Exemplo de código do CustomText em Dart

```
1 class CustomText extends StatelessWidget {
2   const CustomText(this.text,
3       {Key? key, this.size, this.color, this.fontWeight})
4       : super(key: key);
5   final String text;
6   final double? size;
7   final Color? color;
```

```
  final FontWeight? fontWeight;

  @override
  Widget build(BuildContext context) {
    return Text(
      text,
      style: TextStyle(fontSize: size, color: color, fontWeight:
   fontWeight),
    );
  }
}
```

O *widget Button* foi a opção escolhida para concluir o fluxo das telas de maneira que seja intuitiva aos usuários. Assim como o *Text* ele também foi encapsulado dentro de outro *widget*, o *CustomButton*, apresentado no Código 18, para que seja utilizado com os estilos padrão do aplicativo.

Código 18 – Exemplo de código do CustomButton em Dart

```
class CustomButton extends StatelessWidget {
  const CustomButton(
      {Key? key, required this.text, required this.onTap, this.color})
      : super(key: key);
  final String text;
  final VoidCallback onTap;
  final Color? color;
  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: onTap,
      child: Container(
        height: 50,
        width: 200,
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(50),
          color: color ?? UiColors.primary,
        ),
        child: Center(
          child: Text(
            text,
            style: const TextStyle(color: Colors.white),
          ),
        ),
      ),
    );
  }
}
```

O *Container* foi utilizado de muitas formas diferentes devido a sua flexibilidade, um exemplo pode ser encontrado no Código 18 onde ele é utilizado para definir tamanho, largura, cor e curvatura da borda.

MaterialApp é utilizado como raiz do projeto, sendo o primeiro *widget* da árvore de *widgets*. Um exemplo de *Material App* é mostrado no Código 19. Nele é removido o *banner* de modo *debug*, é feita um configuração de tema e é definido o caminho para abertura do App, nesse caso é a *WelcomeView*, página de boas vindas.

Código 19 – Exemplo de código do MaterialApp em Dart

```
return MaterialApp(
      home: const WelcomeView(),
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        canvasColor: Colors.transparent,
      ),
    );
```

Icon é um *widget* para exibição de ícones e Icons é uma lista de ícones. Os dois foram utilizados juntos para inserir ícones nas telas e diminuir o excesso de textos, trazendo uma experiência mais agradável ao usuário. *FloatingActionButton*, como apontado no Código 20, é um botão flutuante acima do conteúdo do aplicativo, utilizado em conjunto com o *Icon* para indicar o caminho para o usuário adicionar novos eventos.

Código 20 – Exemplo de código de FloatingActionButton

```
FloatingActionButton(
            onPressed: () {
              Navigator.of(context).push(MaterialPageRoute(
                  builder: (context) => const RegisterInjection()));
            },
            backgroundColor: UiColors.primary,
            child: const Icon(Icons.add),
          ),
```

SizedBox, como exibido no Código 21, é um *widget* capaz de definir o tamanho de seus filhos. Ademais uma utilização muito comum dele é para distanciar dois *widgets* em uma única direção, criando assim uma margem sem a necessidade de utilizar o *widget Padding* que torna-se muito complexo para algo tão simples. O *Padding*, demonstrado no Código 22, é uma melhor opção quando é necessário adicionar margem em mais de uma direção.

Código 21 – Exemplo de código de SizedBox em Dart

```
const SizedBox(
              width: 8,
            ),
```

Código 22 – Exemplo de código de Padding em Dart

```
child: Padding(
                padding: const EdgeInsets.only(left: 16, right: 16),
                child: ListView.builder(
                  itemCount: 3,
                  itemBuilder: (context, index) {
                    return CalendarItem(
                      eventModel: EventList.listfriday[index],
                    );
                  },
                ),
              ),
```

Column e *Row*, apresentados especificamentes nos Códigos 23 e 24, são utilizados para organizar *widgets* em coluna ou linha, sendo muito úteis para ajudar a criar *designs* de tela mais complexos onde o conteúdo não está definido em uma sequência simples. *Expanded* existe para que um *widget* ocupe o espaço livre em uma *row* ou *column*.

Código 23 – Exemplo de código de Column e Expanded em Dart

```
Column(
      children: <Widget>[
        Container(
          padding: const EdgeInsets.all(10),
          decoration: BoxDecoration(
              border: Border(
                  bottom: BorderSide(
                      color: Colors.grey[200] ?? Colors.grey))),
          child: const TextField(
            decoration: InputDecoration(
                border: InputBorder.none,
                hintText: "Usu rio",
                hintStyle: TextStyle(color: Colors.grey)),
          ),
        ),
        Container(
          padding: const EdgeInsets.all(10),
          child: const TextField(
            decoration: InputDecoration(
              border: InputBorder.none,
              hintText: "Senha",
              hintStyle: TextStyle(color: Colors.grey),
            ),
            obscureText: true,
          ),
        )
      ],
    ),
```

Código 24 – Exemplo de código de Row em Dart

```
Row(
  children: [
    const Icon(
      Icons.schedule,
      size: 18,
    ),
    CustomText(
      ' ${(eventModel.finishDate.hour - eventModel.initialDate.hour).
   toString()} horas',
      fontWeight: FontWeight.bold,
    ),
  ],
),
```

TextField é utilizado para que o usuário possa inserir texto em um campo. É possível verificar o exemplo de uso do *TextFiel* no Código 23. O *widget Gesture Detector*, demonstrado no Código 25, é utilizado para reconhecer o *click* no evento e abrir a página de detalhes.

Código 25 – Exemplo de código de GestureDetector em Dart

```
GestureDetector(
              onTap: () => BottomModal.show(
                context,
                child: DescriptionBottomModal(
                  eventModel: list[index],
                ),
              ),
              child: Container(
                margin: const EdgeInsets.symmetric(horizontal: 2.0),
                color: Colors.deepOrange[100 + d * 100],
                child: CustomText(list[index].name),
              ),
            ),
```

ListView, como apontado no Código 26, foi utilizada no projeto para listar os eventos nas páginas e permitir rolagem da mesma.

Código 26 – Exemplo de código de ListView em Dart

```
ListView.builder(
            itemCount: todayViewmodel.eventList.length,
            itemBuilder: (context, index) {
              return CalendarItem(
                eventModel: todayViewmodel.eventList[index],
              );
            },
          );
```

Wrap, como atestado no Código 27, foi utilizado para listar as opções de escolha dentro de uma *bottomModal*. *ListTile* é um *widget* que representa um único ladrilho em lista e foi utilizado para desenhar os tipos na modal de seleção.

Código 27 – Exemplo de código de Wrap e ListTile em Dart

```
Wrap(
    children: <Widget>[
      ListTile(
        title: Text('Escola'),
        onTap: () => {signupViewmodel.typeSelected(EventType.school);},
      ),
      ListTile(
        title: Text('Domesticos'),
        onTap: () => {signupViewmodel.typeSelected(EventType.house);},
      ),
      ListTile(
        title: Text('Lazer'),
        onTap: () => {signupViewmodel.typeSelected(EventType.recreation)
   ;},
      ),
    ],
  ),
```

SingleChildScrollView, demonstrado no Código 28, é um *widget* padrão para garantir quem um único filho seja rolável.

Código 28 – Exemplo de código de SingleChildScrollView em Dart

```
SingleChildScrollView(
        child: child,
)
```

CircularProgressIndicator, apontado no Código 29, é uma forma simples de exibir para o usuário que algo está carregando.

Código 29 – Exemplo de código de CircularProgressIndicator em Dart

```
if (state is LoadingState) {
            return const CircularProgressIndicator();
          }
```

SilverToBoxAdapter é utilizado em conjunto com *SliverPersistentHeader*. Como demonstrado no Código 30, foi utilizado na tela de eventos da semana para criar um *header* que permanece fixo sobre um conteúdo.

Código 30 – Exemplo de código de SliverPersistentHeader em Dart

```

CustomScrollView(
```

```
        slivers: <Widget>[
          SliverPersistentHeader(
            delegate: WeekViewHeaderDelegate(),
            pinned: true,
          ),
          SliverToBoxAdapter(
            child: child,
          )
      ]
)
```

Stack e *Positioned*, apontados no Código 31, foram utilizados na página de exibição de eventos semanais para criar um visualização em matriz dos eventos, permitindo maior clareza ao usuário.

Código 31 – Exemplo de código de Positioned e Stack em Dart

```
    Stack(
        children: List.generate(
          list.length,
          (index) => Positioned(
            left: 0.0,
            top: list[index].initialDate.hour * 60,
            right: 0.0,
            height: 60.0 *
                (list[index].finishDate.hour - list[index].initialDate.
   hour),
            child: GestureDetector(
              onTap: () => BottomModal.show(
                context,
                child: DescriptionBottomModal(
                  eventModel: list[index],
                ),
              ),
              child: Container(
                margin: const EdgeInsets.symmetric(horizontal: 2.0),
                color: Colors.deepOrange[100 + d * 100],
                child: CustomText(list[index].name),
              ),
            ),
          ),
        ),
      ),
```

4.4 Implementação dos Packages

Nesta seção será discorrido o uso dos *packages* explicados na seção 3.1.3. O *package* http foi utilizado para fazer as requisições a API que retornam os dados necessários para o funcionamento do aplicativo, assim como faz os registros dos eventos e o cadastro do usuário. Para a utilização do *package* http foi criado um encapsulamento para diminuir a dependência do aplicativo a pacotes externos e facilitar o uso dos métodos. Para realizar esse encapsulamento foi criado um *adapter*, demonstrado no Código 32, e um *client* interface para o *adapter*, apontado no Código 33. Dessa forma qualquer inconsistência que ocorra no *package* e surja a necessidade de substituí-lo pode ser feita com facilidade e sem maiores danos ao código, como apresentado nas seções 2.2.1.4 e 2.2.1.1.2.

Código 32 – Exemplo de implementação de um HttpAdapter em Dart utilizando o package http

```
1
2 class HttpAdapter implements HttpClient {
3   Client client;
4
5   HttpAdapter({required this.client});
6
7   @override
8   Future<Response> request(
9       {required String url,
10      String? body,
11      Map<String, String>? headers,
12      required String method}) async {
13    if (method == 'GET') {
14      var response = await client.get(
15        Uri.parse(url),
16        headers: headers,
17      );
18      return response;
19    }
20    if (method == 'POST') {
21      var response = await client.post(
22        Uri.parse(url),
23        body: body,
24        headers: headers,
25      );
26      return response;
27    }
28    throw 'not a valid method';
29  }
30 }
```

Código 33 – Exemplo de implementação de um HttpClient em Dart utilizando o package http

```
abstract class HttpClient {
  Future<Response> request(
      {required String url,
      String? body,
      Map<String, String>? headers,
      required String method});
}
```

Para a implementação das regras de negócio ao realizar as *requests*, assim como aplicar a eles a *url* padrão da *API*, foi criada uma classe *HttpRequest*, apresentada no Código 34, que serve como *service* deste *adapter*.

Código 34 – Exemplo de implementação de um HttpRequest em Dart utilizando o package http

```

class HttpRequest {
  HttpRequest(this.httpClient, this.sharedPreferencesService);
  final HttpClient httpClient;
  final SharedPreferencesService sharedPreferencesService;

  Future<Response> request(
      {required String url,
      String? body,
      Map<String, String>? headers,
      required String method}) async {
    var response = await httpClient.request(
      url: 'url aqui $url',
      body: body,
      method: method,
      headers: headers,
    );
    return response;
  }
}
```

O Código 35 mostra um exemplo de como é feita a utilização desse *HttpRequest* dentro de uma *Repository*.

Código 35 – Exemplo de código de uso da classe HttpRequest em Dart

```

class EventRepository implements EventRepositoryInterface {
      EventRepository({
        required this.httpRequest,
        required this.sharedPreferences,
      });
      final HttpRequest httpRequest;
```

```
8       final SharedPreferencesService sharedPreferences;
9
10      @override
11      Future<List<EventModel>> getEventsByDay(DateTime dateTime) async {
12        Response response = await httpRequest.request(
13            url: "${sharedPreferences.get('url')}/API/events/day",
14            method: 'GET');
15        dynamic json = jsonDecode(response.body);
16        List<EventModel> listEvents = [];
17        for (var element in json) {
18          listEvents.add(EventModel.fromJson(element));
19        }
20        return listEvents;
21      }
22  }
```

O exemplo do Código 35 mostra também o uso do *package shared_preferences*. Neste exemplo ele está sendo utilizado para acessar a *url* da API salva localmente no dispositivo do usuário. Não é um uso muito comum para uma agenda, porém se o usuário tiver interesse no estudo e desenvolvimento de APIs poderá utilizar este aplicativo como *interface* para acessar a sua própria API, apenas alterando o endereço da *request* salvo localmente.

O *package* flutter *bloc* foi o padrão utilizado neste projeto para controle de estado em conjunto com o *design pattern* MVVM. Conforme a seção 3.5.1, ele funciona emitindo estados específicos através da *viewmodel* para interagir com a *view* e exibir o que for determinado para o usuário. Para o uso do *bloc* é necessário criar uma Classe *State* que definirá os estados que a tela poderá assumir, por exemplo *LoadingState* para representar o carregamento de dados ou *ErrorState* para representar um erro no carregamento. Para este exemplo utilizou-se o módulo de *login*, primeiro foi implementado a LoginState, demonstrada no Código 36.

Código 36 – Exemplo de código de LoginState em Dart

```
1  abstract class LoginState {}
2
3  class InitialState extends LoginState {}
4
5  class LoadingState extends LoginState {}
6
7  class LoadedState extends LoginState {}
8
9  class ErrorState extends LoginState {
10   ErrorState(this.message);
11   final String message;
12 }
```

Após a criação dos estados será escrita a *Viemodel*, demonstrada no Código 37, esta *Viewmodel* estende uma *Cubit* e tem em seu construtor o estado inicial definido anteriormente

para que ele seja emitido assim que a *view* for construída na tela. Nos métodos dessa *viewmodel* será emitido constantemente estados para controlar a *view*. Por padrão é recomendado emitir um estado de carregamento quando um método for chamado, um estado de conclusão quando o método for finalizado, ou um estado de erro caso haja algum erro na execução dos métodos.

Código 37 – Exemplo de código de uso da classe LoginViewmodel em Dart

```
class LoginViewmodel extends Cubit<LoginState> {
  LoginViewmodel(this.loginRepository) : super(InitialState());
  late UserModel userModel;
  final LoginRepository loginRepository;

  Future<void> login() async {
    try {
      emit(LoadingState());
      loginRepository.login(userModel);
      emit(LoadedState());
    } catch (e) {
      emit(ErrorState(
          'Houve um erro ao realizar o login, por favor tente novamente
    '));
    }
  }
}
```

Para que as *views* tenham acessos aos estados emitidos pela *viewmodel* é necessário que a *viewmodel* seja injetada na *view*. Para isso foi criada a *LoginInjection*, apontada no Código 38, cuja função é apenas injetar as dependências necessárias.

Código 38 – Exemplo de código de uso da classe LoginInjection em Dart

```
class LoginInjection extends StatelessWidget {
  const LoginInjection({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (_) => LoginViewmodel(LoginRepository(getDependencies())),
      child: const LoginView(),
    );
  }
}
```

Por fim, para que os estados emitidos sejam reconhecidos pela *viewmodel* e que as atualizações necessárias sejam feitas, é utilizado um *BlocBuilder*, exemplificado no código 39, responsável por redesenhar os seus filhos sempre que um estado diferente do atual for emitido. Neste caso sempre que o usuário clicar no botão e tiver que esperar pelo *login*, uma *CircularProgressIndicator*, demonstrada no Código 29, será exibida.

Código 39 – Exemplo de código de uso do BlocBuilder em Dart

```
BlocBuilder<LoginViewmodel, LoginState>(
                        builder: (context, state) {
                          if (state is LoadingState) {
                            return CircularProgressIndicator();
                          }
                          return CustomButton(
                            text: 'Entrar',
                            onTap: () async {
                              await viewmodel.login();
                              Navigator.of(context).pushAndRemoveUntil(
                                  MaterialPageRoute(
                                      builder: (context) =>
                                          const HomeInjection()),
                                  (Route<dynamic> route) => false);
                            },
                          );
                        },
                      ),
```

5 Conclusão

Com o constante surgimento de novos *frameworks*, em conjunto com o constante avanços das técnicas de desenvolvimento de *software*, tanto a nível de código como a nomenclatura de classes e métodos, como a nível de projeto com a organização de pastas, e inúmeras novas boas práticas que surgem para complementar ou substituir as antigas, este trabalho apresenta um pouco desse grande universo de conhecimento que é o desenvolvimento de aplicativos de qualidade, utilizando o *framework* Flutter e sua versatilidade, em uma linguagem muito completa e de qualidade que é o *Dart*. Todo este conhecimento acumulado está aplicado e documentado neste trabalho de muito valor para aqueles que buscam aprender, em nível intermediário, a desenvolver uma aplicação *mobile* com código e arquitetura limpa, em uma estrutura escalável e flexível utilizando o *Framework* Flutter. O resultado da utilização deste conhecimento é uma aplicação mobile de uma agenda, capaz de organizar dinâmicamente as tarefas do usuário, buscando otimiza-las e oferecendo uma melhor experiência com organização de rotinas.

5.1 Trabalhos Futuros

Visto que o mundo do desenvolvimento de *software* de qualidade é grande demais para um único trabalho, muitos pontos podem ser revisados e melhorados, pois no desenvolvimento de *software* não existe código perfeito e livre de problemas. Existe apenas um programador preocupado em fazer um código que pode sobreviver a tudo e continuar crescendo livremente. Ademais muitas *features* do aplicativo Agenda poderiam ser adicionadas para melhorar a experiência do usuário, como a implementação de um "recuperar senha", maior flexibilidade na criação e gerenciamento dos seus eventos e a capacidade de integrar esta agenda com outras agendas comumente utilizadas como o *Google Calendar*.

Referências

BARRY, J.; MCQUILLAN, J. Building rest apis with the asp.net mvc framework. *MSDN Magazine*, v. 42, n. 6, p. 34–39, 2017.

BEAULIEU, A. Designing a sql server 2000 database. *SQL Server Magazine*, v. 5, n. 4, p. 56–60, 2003.

FIELDING, R. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. Acesso em: 22 de Novembro de 2022. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

FLUTTER Team. *Flutter documentation*. 2022. Acesso em: 26 de Janeiro de 2023. Disponível em: https://flutter.dev/docs.

GIDDENS, A. *The constitution of society. Berkeley*. [S.l.]: CA: University of California Press, 1984.

IBGE. *Pesquisa Nacional por Amostra de Domicílios Contínua anual*. 2019. Disponível em: https://sidra.ibge.gov.br/tabela/7356#/n1/all/v/10648/p/last%201/c1/6795/c2/6794/c427/11327/d/v10648%201/l/,v+p+c1+c2,t+c427/resultado. Acesso em: 06 mai. 2021.

KAUR, S.; KAUR, S.; SINGH, G. Migrations: A comprehensive review. *International Journal of Computer Science and Information Security*, v. 17, n. 2, p. 107–111, 2019.

LIU, L.; JIA, L.; LI, S. Microsoft sql server: An overview. *Journal of Computer Science*, v. 4, n. 6, p. 381–384, 2008.

MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. [S.l.]: Pearson, 2008.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. [S.l.]: Pearson Education, 2017.

MICROSOFT. *DOT NET 6.0*. 2021. Acesso em: 18 de Outubro de 2022. Disponível em: https://dotnet.microsoft.com/download/dotnet/6.0.

MILAGRES, R. *Rotinas – Uma Revisão Teórica*. 2013. 161–196 p. Disponível em: https://periodicos.sbu.unicamp.br/ojs/index.php/rbi/article/view/8649013. Acesso em: 06 mai. 2021.

STROUSTRUP, B. *The C++ Programming Language*. [S.l.]: Addison-Wesley, 1985.

WINTER, S. G. The research program of the behavioral theory of the firm: Orthodox critique and evolutionary perspective. *Handbook of behavioral economics*, Greenwich, Conn.: PAI Press, v. 1, p. 151–188, 1986.