Projeto de Engenharia Informática em Contexto Empresarial

Alexandre Marques Machado

Ispg2020100636@ispgaya.pt

Escola Superior de Ciência e Tecnologia

Licenciatura de Engenharia Informática

Orientadores:

Jorge Simões

Thiago Porto

Agradecimentos

Quero agradecer a Queenslab pelo oportunidade e apoio fornecidos durante o desenvolvimento deste projeto de estágio. Agradeço também ao ISPGaya pelo apoio dado ao longo da duração da licenciatura.

Sumário Executivo

Este projeto teve como objetivo melhorar a facilidade com que se produzem aplicações móveis na empresa através da criação de um Boilerplate.

Contextualizando o projeto, o seu desenvolvimento baseou-se fundamentalmente na linguagem de programação Flutter, tendo também a presença de Node.js para criação de uma base de dados.

Neste documento encontram-se detalhadas as informações referentes ao local de estágio/empresa, escopo do projeto, tecnologias e normas do Flutter. Todas as características e funcionalidades implementadas serão explicadas neste documento, bem como opções que não foram implementadas e o porquê de não terem sido selecionadas.

Relativamente a conclusões, estas serão detalhadas no final do documento de forma crítica sobre todo o processo realizado para chegar ao resultado.

Abstract

This project objective was to facilitate the production of mobile applications of the company through the creation of a Boilerplate.

Contextualizing the project, its development was fundamentally based on the programming language of Flutter with the addition of Node.js for the implementation of a data base.

This document details the information portraying the company, the scope of the project, technologies and standards of Flutter. All the characteristics e functionalities implemented will be explained in this document, as well as the options that weren't implemented and the why they weren't selected.

Relatively to the conclusions, those will be detailed at the end of the document with a critical view on the whole process to reach the end goal.

Lista de abreviaturas/siglas

API - Application Programming Interface

JSON - JavaScript Object Notation

UI - User Interface

Índice

Introdução

Este relatório final de estágio apresenta os resultados e aprendizados adquiridos durante o desenvolvimento de um boilerplate em Flutter. O estágio foi realizado na empresa Queenslab, com o objetivo de aprimorar as minhas habilidades de desenvolvimento de aplicações móveis e adquirir experiência prática na criação de uma estrutura sólida e reutilizável para projetos Flutter.

Um boilerplate é uma estrutura básica que contém os componentes essenciais de uma aplicação, como gestão de estado, rotas de navegação, autenticação, chamadas de API e muito mais. Desenvolver um boilerplate personalizado em Flutter pode economizar tempo e esforço no início de cada novo projeto, permitindo que os desenvolvedores se concentrem na lógica de negócio específica da aplicação.

Durante estágio, o principal objetivo foi criar um boilerplate em Flutter que pudesse ser facilmente configurado e adaptado para atender às necessidades de diferentes projetos. Para isso, foram consideradas boas práticas de arquitetura, padrões de projeto e reutilização de código. Além disso, foi necessário incorporar recursos avançados, como gestão de estado eficiente.

Estado da Arte

Desde a introdução do telemóvel moderno na sociedade como um objeto de uso quotidiano que a necessidade de desenvolver a tecnologia móvel cada vez melhor tem aumentado. Resultado, atualmente qualquer solução para uma necessidade social ou pessoal pode ser traduzida para uma aplicação móvel.

Portanto, é importante que as empresas para serem competitivas invistam na rapidez e qualidade da criação de novas aplicações móveis bem como se mantenham a par de todas as inovações na sua área.

“In my opinion, the future of mobile is the future of everything”.

(Galligan, M., 2016).

Atualmente, a utilização de um Smartphone está tão enraizada na maioria das funções diárias de uma pessoa que é difícil imaginar a nossa vida sem eles. As suas utilizações são vastas tais como as suas vantagens, sendo que chega a ser necessária à sua utilização em funções que necessitem de contacto permanente com outras pessoas ou em funções que necessitam de informações atualizadas ao segundo.

"Mobile apps have revolutionized the way we work, play, and interact, becoming an integral part of our daily lives." (Cook, T., 2018)

Consequentemente, tem existido um grande fluxo de novas linguagens móveis. Cada uma com as suas características, mas que pretendem todas responder a necessidade exponencial de uma linguagem de programação eficiente em relação as necessidades e recursos atualmente disponíveis. Como tal, existem enumeras linguagens que podem ser abordados neste assunto, mas deste momento para a frente apenas Flutter vai ser referenciado pois é a base deste projeto.

É importante notar algumas importantes vantagens do Flutter:

- Desenvolvimento multiplataforma – é capaz de criar aplicações para iOS e Android usando um único código-base.
- Comunidade ativa e suporte - o Flutter possui uma comunidade de desenvolvedores ativa e em crescimento, o que significa que se pode

encontrar uma grande quantidade de recursos, bibliotecas e soluções para ajudar no desenvolvimento de aplicações. Além disso, o Flutter é mantido pelo Google, o que garante suporte e atualizações regulares.

- Hot Reload - O recurso de Hot Reload do Flutter permite que os desenvolvedores vejam as alterações em tempo real, sem a necessidade de reiniciar a aplicação.
- Alta performance - o Flutter, através da "Skia", renderiza diretamente os elementos da interface de usuário.
- Desenvolvimento rápido de interfaces de usuário - O Flutter utiliza um conceito chamado "widget" para construir interfaces de usuário. Esses widgets são altamente personalizáveis e permitem que se criem interfaces forma rápida e eficiente.

Claramente que a linguagem Flutter tem as suas desvantagens, mas no contexto da empresa e como linguagem emergente no desenvolvimento de aplicações móveis, esta foi a linguagem adotada para este projeto

Caracterização da entidade de estágio

Este projeto foi desenvolvido na empresa Queenslab como parte da unidade curricular Projeto de Engenharia Informática em Contexto Empresarial. Sediada em Gotemburgo, a Queenslab é uma empresa sueca de consultadoria, que presta os seus serviços nas áreas de tecnologia e design. Tem também outros dois escritórios, um em Estocolmo e outro em Lisboa.

Projetos semelhantes

Como referência para este projeto foi utilizado exemplos de Boilerplates. Destaco um exemplo que utilizei como referência, criado por Nguyễn Thành Minh. Este exemplo permitiu-me ter uma noção das algumas tecnologias que desconhecia e também uma noção da estrutura de um boilerplate.

Outra referência que utilizei mais extensivamente foi as duas aplicações previamente desenvolvidas pela Queenslab em Flutter. Estas serviram para me direcionar nas bibliotecas que a empresa já tinha experiência a utilizar bem como a estrutura das aplicações adotada pela empresa.

Como tal, o resultado teve uma influência grande das referências anteriormente mencionadas, tendo introduzido também algumas bibliotecas desconhecidos pela empresa com o pretexto que, na minha opinião, são melhores e mais fáceis de utilizar.

Requisitos

Requisitos funcionais:

Funcionalidade	Prioridade
Login/Register	Essencial
Autenticação	Essencial
Componentes reutilizáveis	Essencial
Padrão de design	Essencial
Integração API	Essencial
Suporte GraphQL	Essencial
Respostas e Erros	Essencial
Persistência de dados no dispositivo	Essencial
Caching(Modo offline)	Desejável
Testes	Desejável

Requisitos não funcionais:

Este projeto apenas necessita de ter uma facilidade de utilização, para que todos os funcionários da empresa tenham facilidade a utilizar o Boilerplate.

Casos de uso:

Dado a natureza deste projeto, não existem casos de uso visto que um Boilerplate se destina a ser uma ferramenta para facilitar e agilizar a criação de aplicações. Como tal, não tem qualquer função como produto final. No entanto, para efeitos meramente demonstrativos, apresento o seguinte exemplo:

Login

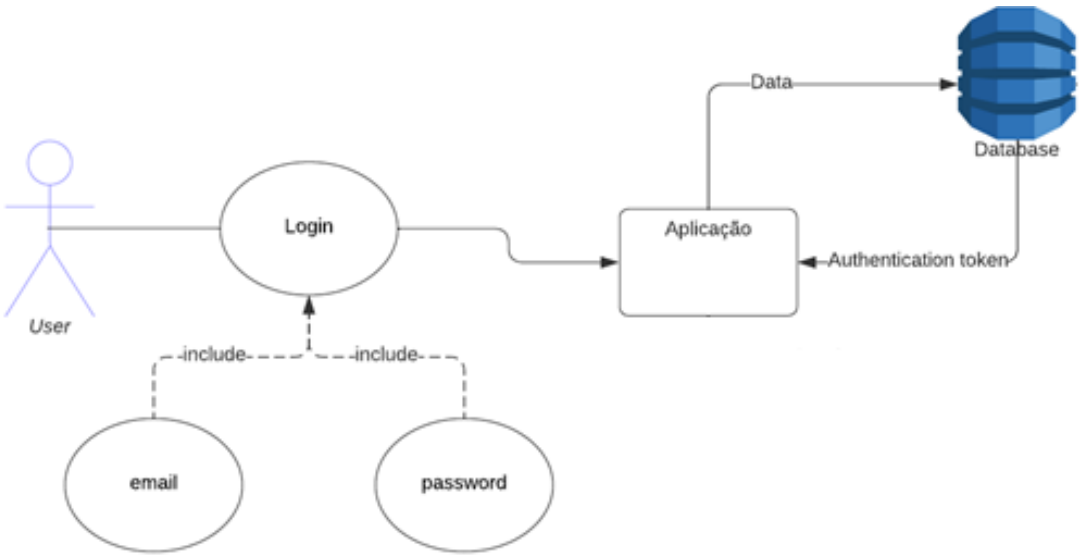

Figura 1 Login Use case

No caso uso descrito em cima, o utilizador processa o seu Login ao fornecer o seu email e password. A aplicação recebe estas informações e envia para o servidor (base de dados), caso estejam corretas, o servidor envia um token de autenticação a confirmar o utilizador.

Descrição de trabalho/atividades realizadas

Como mencionado anteriormente, este projeto é uma resposta a necessidade de criar aplicações em Flutter de uma maneira mais eficiente e ágil. Para tal, foi criado uma guideline através da ferramenta "Planner" que permitiu ter uma estrutura sobre conteúdos a desenvolver e as suas timelines.

O processo inicial foi estudar Flutter e as suas funções básicas:

- Dart Programming Language: Flutter utiliza a linguagem de programação Dart. É essencial ter um entendimento sólido dos conceitos e recursos da linguagem Dart, como variáveis, funções, classes, herança, mixins e async/await.
- Widgets: Widgets são os blocos de construção fundamentais em Flutter. É importante entender os diferentes tipos de widgets disponíveis e como combiná-los para construir a interface do usuário. Conhecimentos sobre os principais conceitos, como Stateless Widgets e Stateful Widgets, são necessários para criar interfaces interativas.
- Layouts e Estrutura de UI: Familiaridade com os diferentes tipos de layouts (como Row, Column, Stack, etc.) e como organizar e posicionar os widgets é crucial para criar interfaces responsivas e visualmente agradáveis. Além disso, entender a estrutura de widgets e como criar hierarquias de widgets aninhados é importante para criar uma interface bem estruturada.

Juntamente com este estudo inicial, foi também feito um estudo dos dois aplicações Flutter já produzidos pela empresa. Isto permitiu ter também uma noção da estruturação e utilização das ferramentas anteriores feitas pela empresa.

Após este estudo inicial, iniciou-se a fase de estudo e criação do design de estrutura/gestão de estado a utilizar no Boilerplate.

Flutter oferece várias opções para gestão do estado da aplicação, como setState, Provider, BLoC, MobX, Redux, entre outros. Ter conhecimento sobre diferentes padrões

e bibliotecas de gestão de estado é importante para criar aplicações escaláveis e fáceis de manter.

O design de estrutura que decidi adotar para este projeto foi o BLoC e passo a citar porque:

- Separação de preocupações: O BLoC promove uma clara separação entre a lógica de negócio e a interface do utilizador. Isso permite que se mantenha o código organizado e facilmente compreensível.
- Reatividade: O BLoC utiliza Streams para propagar alterações de estado para os componentes da interface do utilizador. Isso possibilita uma abordagem reativa, onde os componentes são atualizados automaticamente sempre que o estado no BLoC muda.
- Flexibilidade: O BLoC oferece flexibilidade para gerir o estado da aplicação. Pode-se escolher entre diferentes abordagens para a comunicação entre o BLoC e os componentes da interface do usuário, como o uso de Streams, ChangeNotifier ou Provider. Esta flexibilidade permite que se adote uma abordagem mais adequada para cada projeto.

Obviamente, o BLoC também tem as suas desvantagens. No entanto, após a minha análise, achei que seria a tecnologia mais adequada, maioritariamente pela sua flexidade e capacidade de trabalhar com outras tecnologias de gestão de estado que possam ser melhores em situações especificas.

A fase seguinte consistiu em analisar os componentes UI do Flutter. Foram estudados todos os componentes classificados como essenciais para o desenvolvimento básico de uma App.

Iniciou-se com o Widget AppBar, que é um componente de interface de usuário pré-definido que representa a barra da aplicação. É constituído, geralmente, por estes elementos:

- Título - Um texto que descreve o conteúdo ou o propósito do ecrã atual.
- Ícones de ação - Ícones que representam ações comuns relacionadas ao ecrã atual.

- Menu “Hamburger” - Um ícone (três linhas horizontais) que, quando clicado, pode abrir um menu de navegação lateral ou um menu suspenso com mais opções.

Adicionalmente, a AppBar é altamente personalizável e oferece várias propriedades para ajustar seu estilo, como cor de fundo, cor do texto, altura, posição do título, entre outros, sendo também possível adicionar funcionalidades extras, como animações, interações de toque e transições suaves.

O próximo componente UI foi o Bottom Navigation Bars. Essencialmente, este Widget é uma barra de navegação na parte inferior do ecrã que é utilizada para navegação entre seções ou páginas/ecrã da aplicação. O utilizador clica num ícone da barra de navegação e é direcionado para a seção ou página correspondente a esse ícone. É constituído, geralmente, por estes elementos:

- Ícones de Navegação: São ícones que representam as diferentes seções ou páginas/ecrã da aplicação. Cada ícone é associado a uma ação específica, como abrir uma página ou executar uma determinada função.
- Rótulos: Rótulos de texto opcionais podem ser exibidos abaixo dos ícones para identificar as seções ou páginas correspondentes. Os rótulos podem ser úteis para fornecer informações adicionais sobre cada item de navegação.

O próximo componente UI foram os Botões (Buttons), Widget utilizado para criar fromas interativas de realizar ações ou acionar eventos numa aplicação. A seu uso torna a aplicação mais interativa para o utilizador e consequentemente mais acessível. Existem diferentes tipos de Buttons como, por exemplo:

- ElevatedButton: Também conhecido como "RaisedButton" em versões anteriores, é um botão que possui uma elevação sutil quando pressionado.
- TextButton: É um botão que se parece com um texto simples, sem uma cor de fundo visível.
- OutlinedButton: É um botão com um contorno visível e sem cor de fundo.
- IconButton: É um botão que contém um ícone em vez de um rótulo de texto.
- FloatingActionButton: É um botão circular flutuante que fica acima do conteúdo principal.

- DropdownButton: É um botão que exibe uma lista suspensa de opções quando pressionado.

Apesar de existirem vários diferentes tipos de Buttons, todos eles partilham algumas características em comum:

- onPressed: É um callback (função) que é acionado quando o botão é clicado.
- child: É o conteúdo do botão, como um texto ou um widget filho.
- style: É um objeto TextStyle que define o estilo do texto dentro do botão, como cor, fonte, tamanho, entre outros.
- disabled: É uma propriedade booleana que define se o botão está ativo ou desativo. Quando desativado, o botão não responderá a eventos de click.
- color: Define a cor de fundo do botão. Depende do tipo de botão e pode variar de acordo com o tema da aplicação.
- textColor: Define a cor do texto dentro do botão.
- padding: Define o preenchimento interno do botão, controlando o espaço entre o conteúdo do botão e os limites.

O próximo componente UI foi o Form. Este é o Widget utilizado para criar formulários, fornecendo uma maneira conveniente de validar e processar os dados inseridos pelo utilizador. Os seus componentes principais são:

- Form: É o widget principal que envolve os campos de entrada e é responsável por gerenciar seu estado. Ele mantém uma lista de GlobalKey para cada campo, permitindo que o Form rastreie e valide os campos individualmente.
- TextFormField: Cria inputs que faz parte do Form. O TextFormField possui recursos como validação de entrada, manipulação de erros, máscaras de texto e muito mais.
- GlobalKey: É uma chave global que identifica exclusivamente um input no Form. Cada input dentro do Form deve ter uma GlobalKey associada a ele para que o Form possa aceder, validar e manipular seu estado.

O próximo componente UI foram os Dialogs que são Widgets utilizados para exibir informações ou interagir com o utilizador através de uma janela modal temporária. São invocados através do método showDialog(), sendo necessário fornecer o contexto atual e o widget de diálogo que desejamos aprensetar. Existem diferentes tipos de Dialogs, como:

- AlertDialog: É um diálogo que exibe uma mensagem de alerta ou aviso para o utilizador. Pode incluir um título, um corpo de texto e botões de ação, como "OK" ou "Cancelar".
- SimpleDialog: É um diálogo que exibe uma lista de opções para o utilizador escolher. Cada opção é representada por um ListTile e pode ser selecionada pelo usuário.

O próximo componente UI foram as Lists, Widgets utilizados para exibir vários dados em formato de uma lista. Permitem visualizar dados de forma organizada e scrollable. Os dados de uma lista são geralmente acedidos através de um índice. Alguns exemplos de Lists são:

- ListView: É um widget que exibe uma lista de dados scrollable verticalmente ou horizontalmente. O ListView é útil quando a lista de itens não se encaixa totalmente no ecrã e requer scrolling para visualizar todos os dados.
- GridView: É um widget que exibe uma grid de dados, onde cada dado é organizado em células. O GridView é útil para exibir dados em formato de grid, como uma galeria de imagens ou uma lista de produtos.
- SingleChildScrollView: É um widget que permite scrolling para exibir conteúdo que normalmente não cabe no ecrã. O SingleChildScrollView é útil quando existe uma lista curta ou uma única seção de conteúdo que precisa ser scrollable.

E, por fim, o último componente UI analisado foram os Cards. São Widgets que criam um container retangular, geralmente usados para exibir informação ou conteúdo de uma maneira visualmente agradável, destacando e organizando a informação de uma maneira clara. Tem características distintas como:

- Elevation: Uma propriedade de elevação que lhes confere uma sombra subtil, dando uma aparência de sobreposição no ecrã. A elevação cria uma sensação de profundidade e destaque em relação ao restante conteúdo.
- Content: Pode conter vários tipos de Widgets, como texto, imagens, botões, ícones e até mesmo outros Cards. Eles oferecem flexibilidade para exibir informações relevantes e interações dentro de seu espaço.
- Style: Sendo altamente personalizáveis, pode ser definido a cor de fundo, a cor da sombra, a borda, o espaçamento interno e outros estilos para adaptar o visual do Card de acordo com o design da aplicação.

Após a análise e implementação de um exemplo de cada um destes Widgets anteriormente referidos, passou-se para a seguinte fase do projeto API integration. Foram feitos estudos de bibliotecas como Dio, Chopper, http e foi selecionada a biblioteca http.

A biblioteca http é bastante versátil, permitindo ao utilizador fazer solicitações HTTP (como GET, POST, PUT, DELETE, etc) a servidores. Outras funcionalidades da biblioteca são:

- Upload e download de arquivos
- Tratamento de erros e códigos de status HTTP
- Autenticação com servidores (juntamente com outras bibliotecas)

Outra biblioteca que também foi implementada foi a biblioteca GraphQL que executa funções semelhantes a biblioteca http mas apenas comunica com servidores GraphQL e fá-lo de maneira mais eficiente e flexível. Outras funcionalidades da biblioteca são:

- Realizar consultas e mutações GraphQL
- Gerir cache de dados
- Permite que o servidor envie atualizações em tempo real para o cliente

Por fim, a última fase foi implementar um sistema de autenticação e de persistência de dados. Foram utilizados os packages JWT (JSON Web Tokens) e Flutter Secure Storage, respectivamente.

A biblioteca JWT permite criar, descodificar e validar tokens JWT com a funcionalidade de autenticar utilizadores. No entanto, só está biblioteca não chega para

construir um sistema de autenticação, sendo, portanto, necessário implementá-la com o recurso a outras bibliotecas como a biblioteca http.

A biblioteca Flutter Secure Storage foi utilizada para guardar dados no dispositivo. Estes dados são depois acedidos pela aplicação e verificam se o utilizador já executou funções como registo ou login previamente na aplicação. Se for o caso, a aplicação em vez de pedir ao utilizador para voltar a realizar as funções, redireciona o utilizador para a página Users da aplicação.

Simultaneamente na implementação do JWT foi criado um servidor local (base de dados) em Node.js, como demonstração de como ambos funcionavam.

Teste de Validação

Teste de implementação do BLoC/Cubit:

Anexo 1 – Verificação se o sistema de gestão de estado adotado está a funcionar como desejado, altera corretamente os estados da aplicação e navega para diferentes páginas.

Teste do Widget AppBar:

Anexo 2 – Verificação se o Widget AppBar está implementado de maneira correta e de forma reutilizável.

Teste do Widget Bottom Navigation Bar:

Anexo 3 – Verificação se o Widget Bottom Navigation Bar está implementado de maneira correta e de forma reutilizável.

Teste do Widget Buttons:

Anexo 4 - Verificação se o Widget Buttons está implementado de maneira correta e de forma reutilizável.

Teste do Widget Form:

Anexo 5- Verificação se o Widget Bottom Navigation Bar está implementado de maneira correta e de forma reutilizável.

Teste do Widget Dialogs:

Anexo 6 - Verificação se o Widget Bottom Navigation Bar está implementado de maneira correta e de forma reutilizável.

Teste do Widget Lists:

Anexo 7 - Verificação se o Widget Bottom Navigation Bar está implementado de maneira correta e de forma reutilizável.

Teste do Widget Cards:

Anexo 8 - Verificação se o Widget Bottom Navigation Bar está implementado de maneira correta e de forma reutilizável.

Teste da implementação da biblioteca http:

Anexo 9 – Verificação da implementação da biblioteca http no tratamento de RestAPIs e processamento da informação obtida.

Teste de implentação da biblioteca GraphQL:

Anexo 10 – Verificação da implementação da biblioteca GraphQL e dos dados recebidos. Verificação da integração da biblioteca com o BLoC e da atualização de estado.

Teste da implementação da biblioteca JWT:

Anexo 11 - Verificação da implementação da biblioteca JWT e da integração com o server criado em Node.js. Comparação de tokens e integração com a aplicação em Flutter.

Teste da implementação da biblioteca Flutter Secure Storage:

Anexo 12 - Verificação da implementação da biblioteca Flutter Secure Storage e se os dados são efetivamente guardados no dispositivo móvel onde se encontra instalada a aplicação.

Cronograma

Tasks	Descrição	Tempo
Flutter Basics	Estudo e revisão da linguagem Flutter	08/05/2023-12/05/2023
Design Patterns	Estudo e implementação da estrutura e método de gestão de estado	10/05/2023-16/05/2023
UI Components	Estudo e implementação dos diversos componentes necessários para uma aplicação	17/05/2023-26/05/2023
API Integration	Estudo e implementação de diversas bibliotecas para lidar com API's e outros serviços	29/5/2023-09/6/2023
Manual Testing	Teste de todas as funcionalidades acima descritas e resolução de conflitos	22/6/2023-26/6/2023
Costumizability	Implementar mais customizações nas Widgets reutilizáveis	26/6/2023-30/6/2023
Documentação	Analisar a documentação consultada e redigir o relatório final de projeto	3/7/2023-10/7/2023

Tabela 1 Cronograma

Para este projeto foi alocado uma duração de 2 meses, tendo sido concluído quase dentro do tempo estabelecido. Existiu um período que não foi realizada nenhuma task e tal aconteceu devido a minha necessidade de dedicar tempo as projetos e estudo de outras cadeiras (final do semestre). Tirando o motivo anterior, a razão pelo ligeiro atraso foi alguma falta de conhecimento na área de API Integration. Apesar disso, o resto das Tasks foram desenvolvidas no tempo alocado para elas.

Meios previstos e meios necessários

Primeiramente, para ser capaz de desenvolver este projeto foi necessária a previsão de uma revisão de conceitos sobre Flutter. Foi também prevista uma continua aprendizagem de todas as funcionalidades a implementar neste projeto.

Na realidade, os meios necessários foram agilizados na situação de revisão de conceitos base de Flutter dado estes já terem abordados na UC de Computação Móvel, adestrada no 1º semestre do 3º ano da licenciatura em Engenharia Informática no ISPGaya. No entanto, foi necessária uma aprendizagem relativamente a API's integration, visto que não tinha conhecimento anterior de como implementar em Flutter. Como tal, foi necessário fazer alguns ajustes em relação a fase de API's integration sendo, no entanto, comprido o que estava planeado. Houve também uma adição extra nas funcionalidades, tendo sido adicionado ao projeto um servido em node.js para utilizar como servidor de autenticação.

Problemas e decisões

Durante o desenvolvimento do projeto ocorreram vários problemas de implementação de funcionalidades sendo, como já referido anteriormente no documento, que maioria ocorreram na fase de API integration. Sendo tudo ferramentas novas, houve períodos de aprendizagem para cada uma das tecnologias.

Destaco a seguintes ferramentas em que houve mais dificuldade de implementação:

- BLoC – Como já referido anteriormente, o BLoC é uma biblioteca de gestão de estado da aplicação. Foi necessário um período de aprendizagem considerável visto que não existia experiência a trabalhar com ferramentas de gestão de estado em Flutter.

- GraphQL – Mais uma vez, a falta de experiência causou alguma demora na implementação desta biblioteca. No entanto, experiência anterior com ferramentas semelhantes de tratamento de queries, auxiliou a sua compreensão.

- JWT – Apesar de ter trabalho com tokens anteriormente, foi em linguagens diferentes. A dificuldade aqui baseou-se na integração do JWT com Flutter.

Dentro das outras ferramentas houve também dificuldades sendo que a experiência de tratamento com as funcionalidades definidas na fase API integration era nula nesta situação. No entanto destaco as bibliotecas anteriores por terem sido as mais diferentes.

Análise de resultado

Nesta seção do documento irei abordar as funcionalidades implementadas, bem como o código utilizado. Uma breve explicação de cada funcionalidade vai ser apresentada para contextualizar o leitor, bem como algumas notas de possíveis escolhas ou decisões efetuadas.

BLoC:

```dart
import 'package:bloc/bloc.dart';
import 'package:boilerplat/Bloc/events.dart';
import 'package:boilerplat/Bloc/states.dart';
import 'package:boilerplat/Repositories/item_repo.dart';
import 'package:boilerplat/Repositories/http_repo.dart';
import 'package:boilerplat/Services/graphql.dart';

class ItemBloc extends Bloc<ItemEvent, ItemState> {
  final _itemRepo = ItemRepository();

  ItemBloc() : super(ItemInitialState()) {
    on<LoadItemEvent>(
      (event, emit) => emit(ItemSuccessState(items: _itemRepo.loadItem())),
    );

    on<AddItemEvent>(
      (event, emit) =>
          emit(ItemSuccessState(items: _itemRepo.addItem(event.item))),
    );

    on<RemoveItemEvent>(
      (event, emit) =>
          emit(ItemSuccessState(items: _itemRepo.removeItem(event.item))),
    );
  }
}
```

Figura 2 SubClasse BLoC ItemBloc e subclasses

```dart
import '../Models/item.dart';
import 'package:equatable/equatable.dart';
import 'package:flutter/material.dart';

@immutable
abstract class ItemEvent {}

class LoadItemEvent extends ItemEvent {}

class AddItemEvent extends ItemEvent {
  Item item;

  AddItemEvent({
    required this.item,
  });
}

class RemoveItemEvent extends ItemEvent {
  Item item;

  RemoveItemEvent({
    required this.item,
  });
}
```

Figura 3 Class ItemEvent e subclasses

```dart
import 'package:boilerplat/Models/item.dart';
import 'package:boilerplat/Models/user.dart';
import 'package:equatable/equatable.dart';
import 'package:flutter/material.dart';

@immutable
abstract class ItemState {
  List<Item> items;

  ItemState({
    required this.items,
  });
}

class ItemInitialState extends ItemState {
  ItemInitialState() : super(items: []); // estado inicial vázio
}

class ItemSuccessState extends ItemState {
  ItemSuccessState({required List<Item> items}) : super(items: items);
}

class ItemErrorState extends ItemState {
  final Error error;

  ItemErrorState({required this.error, required super.items});
}
```

Figura 4Classe ItemState e subclasses

```dart
import 'package:boilerplat/Models/item.dart';

class ItemRepository {
  final List<Item> _items = [];

  List<Item> loadItem() {
    _items.addAll([
      Item(name: 'a'),
      Item(name: 'b'),
      Item(name: 'c'),
    ]);
    return _items;
  }

  List<Item> addItem(Item item) {
    _items.add(item);
    return _items;
  }

  List<Item> removeItem(Item item) {
    _items.remove(item);
    return _items;
  }
}
```

Figura 5 Class ItemRepository

Nas figuras 2, 3, 4 e 5 esta representado o código implementado para uma abordagem inicial a estrutura de gestão de estados (BLoC) do Boilerplate.

Em primeiro lugar, é necessário instalar a biblioteca flutter_bloc e para tal basta apenas adicionar a linha flutter_bloc: ^8.0.1 (ou versão mais atual) na linha das dependências do ficheiro pubspec.yaml.

Após este passo, podemos importar a biblioteca no ficheiro e utilizá-la. Criamos uma class 'ItemBloc' que herda as variáveis e métodos da superclass 'Bloc'. Os eventos vão ser do tipo 'ItemEvent' e os estados emitidos do tipo 'ItemState'.

É necessário criar um repositório que vai conter os dados, como demonstrados na Figura 5.

Depois cria-se as classes para lidar com os eventos e estados como demonstrado na figura 3 e 4.

Por fim, no ficheiro representado na figura 2 criamos uma variável '_itemRepo' '(variável privada) que é uma instância da class 'ItemRepository'. São definidas as maneiras de tratar os eventos e executadas as funções que emitem um novo estado quando os eventos são chamados. Por exemplo, no 'on<LoadItemEvent>' o event handler recebe o evento e emite um novo estado ('ItemSuccessState') com os dados carregados do repositório.

AppBar:

```dart
import 'package:flutter/material.dart';

class MyAppBar extends StatelessWidget implements PreferredSizeWidget {
  final String text;
  final Color? backgroundColor;
  final List<Widget>? widgets;
  final IconThemeData? iconTheme;

  const MyAppBar({
    super.key,
    required this.text,
    this.backgroundColor,
    this.widgets,
    this.iconTheme,
  });

  @override
  Size get preferredSize => const Size.fromHeight(60.0);

  @override
  Widget build(BuildContext context) {
    return AppBar(
      backgroundColor: backgroundColor,
      title: Text(text,
          style: const TextStyle(
            fontWeight: FontWeight.bold,
          )),
      centerTitle: true,
      actions: widgets,
    );
  }
}
```

Figura 6 Class MyAppBar

```dart
@override
Widget build(BuildContext context) {
  return MultiBlocProvider(
    providers: [
      BlocProvider<UserBloc>(
        create: (BuildContext context) => UserBloc(UserRepository()),
      ),
    ],
    child: Scaffold(
      appBar: const MyAppBar(
        text: 'Users',
        backgroundColor: Colors.green,
        widgets: [],
      ),
```

Figura 7 Utilização do Widget MyAppBar

Na figura 6 esta representado a implementação do widget AppBar de forma reutilizável.

Criei a classe 'myAppBar' que é uma subclasse da class 'StatelessWidget' que implementa a interface 'PreferredSizeWidget'. Esta implementação é útil neste caso porque permite definir widgets com tamanhos constantes. Posteriormente, cria-se as variáveis de classe e os construtores. Por fim, cria-se o design do Widget em si com as variáveis definidas anteriormente.

Na figura 7 esta representado a utilização do Widget 'AppBar' no ficheiro 'users_page.dart'. A únicos requisitos para o utilizar são importar o ficheiro 'app_bar.dart' para o ficheiro onde se pretende utilizar. Depois apenas é necessário evocar a classe através do método 'MyAppBar()' e preencher os requisitos obrigatórios com informação.

Bottom Navigation Bar:

```dart
import 'package:flutter/material.dart';
import '../Bloc/Navigation/nav_cubit.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

class MyBottomNavigationBar extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<NavigationCubit, NavigationState>(
        builder: (context, state) {
          return BottomNavigationBar(
            currentIndex: state.index,
            showUnselectedLabels: false,
            items: const [
              BottomNavigationBarItem(
                icon: Icon(
                  Icons.home,
                ),
                label: 'Home',
              ),
              BottomNavigationBarItem(
                icon: Icon(
                  Icons.settings,
                ),
                label: 'Settings',
              ),
              BottomNavigationBarItem(
                icon: Icon(
                  Icons.person,
                ),
                label: 'Profile',
              ),
            ],
```

Figura 8 Class MyBottomNavigationBar

```
            onTap: (index) {
              if (index == 0) {
                BlocProvider.of<NavigationCubit>(context)
                    .getNavBarItem(NavbarItem.home);
              } else if (index == 1) {
                BlocProvider.of<NavigationCubit>(context)
                    .getNavBarItem(NavbarItem.settings);
              } else if (index == 2) {
                BlocProvider.of<NavigationCubit>(context)
                    .getNavBarItem(NavbarItem.profile);
              }
            },
          );
        },
      ),
      }
}
```

Figura 9 Continuação da Classe MyBottomNavigationBar

Na figura 8 e 9 está representada a implementação do Widget Bottom Navigation Bar de maneira reutilizável.

Cria-se a class 'MyBottomNavigationBar' que é uma subclasse da class 'StatelessWidget'. Depois criamos a estrutura do Widget, utilizando o BLoC de navegação que já tínhamos desenvolvido. Por fim, cria-se uma lista de items que vão representar os icons na barra de navegação e o processo quando o utilizador pressionada cada um dos icons (onTap()).

```
      bottomNavigationBar:  MyBottomNavigationBar(),
      body: BlocBuilder<NavigationCubit, NavigationState>(
          builder: (context, state) {
        if (state.navbarItem == NavbarItem.home) {
          return const ItemPage();
        } else if (state.navbarItem == NavbarItem.settings) {
          return const GraphQlPage();
        } else if (state.navbarItem == NavbarItem.profile) {
          //return UsersPage();
        }
        return Container();
```

Figura 10 Utilização do Widget MyBottomNavigationBar

Na figura 10 está representada a utilização do Widget 'MyBottomNavigationBar' bem como as páginas de navegação a que cada item na navbar corresponde.

Buttons:

```dart
import 'package:flutter/material.dart';

class MyButton extends StatelessWidget {
  final String? text;
  final Function? callbackFunction;
  const MyButton({
    super.key,
    required this.text,
    this.callbackFunction,
  });

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () => callbackFunction,
      child: Text(text!),
    );
  }
}
```

Figura 11 Classe MyButton

Na figura 11 esta representada a implementação do Widget Button de maneira a ser reutilizável.

Tal como em Widgets anteriores, cria-se a classe 'MyButtons' que é subclasse de 'StatelessWidget'. Posteriormente, cria-se as variáveis de classe e o construtor.

Por fim, cria-se a arquitetura do Widget de acordo com as variáveis definidas anteriormente.

```
),
MyButton(
  text: 'Save',
  callbackFunction: callback(),
```

Figura 12 Utilização do Widget MyButton

Na figura 12 esta representada a utilização do Widget 'MyButton'. Neste caso, o Widget ira apresentar um texto "Save" no botão e ira executar a função 'callback()' quando pressionado.

Forms:

```dart
import 'package:boilerplat/Widgets/button.dart';
import 'package:boilerplat/Widgets/input_field.dart';
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

import '../Bloc/Navigation/nav_cubit.dart';

class MyForm extends StatelessWidget {
  final _formkey = GlobalKey<FormState>();
  MyForm({super.key});
  callback() {
    if (_formKey.currentState!.validate()) {
      setState() {}
    }
  }
```

Figura 13 Classe MyForm

```
  @override
  Widget build(BuildContext context) {
    return MultiBlocProvider(
        providers: [
          BlocProvider<NavigationCubit>(
            create: (context) => NavigationCubit(),
          )
        ],
        child: Form(
          key: _formkey,
          child: Padding(
              padding: const EdgeInsets.symmetric(horizontal: 20),
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  const MyInputfield(
                    obscureText: false,
                    hintText: 'User',
                    icon: Icons.person,
                  ),
                  const MyInputfield(
                    obscureText: true,
                    hintText: 'password',
                    icon: Icons.lock,
                  ),
                  MyButton(
                    text: 'Save',
                    callbackFunction: callback(),
                  ),
                ],
              )),
        ));
  }
}
```

Figura 14 Continuação da Classe MyForm

Na figura 13 e 14 esta representado a implementação do Widget Form de maneira reutilizável.

Tal como em Widgets anteriores, cria-se a classe ‘MyForm’ que é subclasse de ‘StatelessWidget’. Posteriormente, cria-se as variáveis de classe e o construto. No caso dos Widgets ‘Forms’ pode-se definir também uma ‘GlobalKey’ que é utilizada para verificar os dados do Widget ‘Form’.

Por fim, implementa-se a arquitetura do Widget ‘MyForm’. Neste caso, foi também adicionado um BLoC de navegação caso a função do ‘MyButton’ necessite de redirecionar o utilizar após a submissão.

Dialogs:

```dart
import 'package:flutter/material.dart';

enum DialogAction { yes, abort }

class Dialogs {
  static Future<DialogAction> yesAbortDialog(
    BuildContext context,
    String title,
    String body,
  ) async {
    final action = await showDialog(
      context: context,
      barrierDismissible: false,
      builder: (BuildContext context) {
        return AlertDialog(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(10),
          ),
          title: Text(title),
          content: Text(body),
          actions: <Widget>[
            ElevatedButton(
              onPressed: () => Navigator.of(context).pop(DialogAction.abort),
              child: const Text('No'),
            ),
            ElevatedButton(
              onPressed: () => Navigator.of(context).pop(DialogAction.yes),
              child: const Text(
                'Yes',
                style: TextStyle(color: Colors.white),
              ),
            )
          ],
        );
      },
    );
    return (action != null) ? action : DialogAction.abort;
  }
}
```

Figura 15 Classe Dialogs

```dart
IconButton(
  icon: const Icon(Icons.person_add),
  onPressed: () async {
    final action = await Dialogs.yesAbortDialog(
      context,
      'Add Event',
      'Do you want to add d?',
    );
    if (action == DialogAction.yes) {
      bloc.add(AddItemEvent(item: Item(name: 'd')));
```

Figura 16 Utilização do Widget Dialogs

Na figura 15 esta representada a implementação do Widget 'Dialogs' de forma reutilizável.

Neste caso é criado uma enumeração 'DialogAction' que representa as ações possíveis de dialog. Estas vão ser escolhas fornecidas ao utilizador. Cria-se também a class 'Dialogs' onde vão ser definidos os métodos para mostrar os diálogos.

Neste caso só foi implementado um método, 'yesAbortDialog'. Este método cria uma caixa de diálogo com o contexto, título e conteúdo que quisermos atribuir. Juntamente com estas características a caixa também vai apresentar dois botões com os valores da enumeração criada no início, sendo cada valor atribuído a uma caixa exclusivamente.

Por fim, é feita uma verificação da resposta dada pelo utilizador ao diálogo e dependendo da resposta, fecha-se o caixa de diálogo.

Na figura 16 esta representada a utilização do Widget 'Dialogs'. Neste caso o dialog esta a ser utilizado num botão. Se o utilizador carregar no botão yes do diálogo, é adicionado ao repositório de items o item "d". Se carregar não, o diálogo fecha e não é adicionado o item "d".

Lists:

```dart
import 'package:flutter/material.dart';

class MyList extends StatelessWidget {
  final List<String> items;
  final void Function(int) onItemClick;

  const MyList({
    Key? key,
    required this.items,
    required this.onItemClick,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: items.length,
      itemBuilder: (context, index) {
        return ListTile(
          title: Text(items[index]),
          onTap: () {
            onItemClick(index);
          },
        );
      },
    );
  }
}
```

Figura 17 Classe MyList

```
if (state is UserLoadedState) {
  List<UserModel> userList = state.users;
  return ListView.builder(
    itemCount: userList.length,
    itemBuilder: (_, index) {
      return Padding(
          padding: const EdgeInsets.symmetric(
              vertical: 4, horizontal: 8),
          child: MyCard(
            title:
                '${userList[index].firstName} ${userList[index].lastName}',
            subtitle: '${userList[index].email}',
            leading: CircleAvatar(
              backgroundImage: NetworkImage(
                userList[index].avatar.toString(),
```

Figura 18 Utilização do Widget MyList

Na figura 17 está representada a implementação do Widget 'Lists' de forma reutilizável.

Neste caso foi criado uma classe 'MyList' que é subclasse da class 'StatelessWidget'. Posteriormente foram criados as variáveis de classe e o construto. Por fim, criou-se a arquitetura do widget utilizando as variáveis criadas anteriormente.

Na figura 18 está representada a utilização do Widget 'MyList'. Neste caso não foi utilizado a lista definida anteriormente por ter sido necessário implementar argumentos específicos e como não é uma lista complexa foi tratada como exceção. O objetivo permanece o mesmo no entanto, esta lista é constituída por vários cards em que cada 1 corresponde a um index da lista.

Cards:

```dart
import 'package:flutter/material.dart';

class MyCard extends StatelessWidget {
  final String title;
  final String subtitle;
  final Widget? leading;
  final Widget? trailing;
  final void Function()? onTap;

  const MyCard({
    Key? key,
    required this.title,
    required this.subtitle,
    this.leading,
    this.trailing,
    this.onTap,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      child: ListTile(
        leading: leading,
        title: Text(title),
        subtitle: Text(subtitle),
        trailing: trailing,
        onTap: onTap,
      ),
    );
  }
}
```

Figura 19 Classe MyCard

```
child: MyCard(
  title:
      '${userList[index].firstName} ${userList[index].lastName}',
  subtitle: '${userList[index].email}',
  leading: CircleAvatar(
    backgroundImage: NetworkImage(
      userList[index].avatar.toString(),
    )
```

Figura 20 Utilização do Widget MyCard

Na figura 19 está representada a implementação do Widget 'Card' de forma reutilizável.

Neste caso foi criada a class 'MyCard' que pertence a subclasse 'StatelessWidget'. Posteriormente, cria-se as variáveis de classe e o construtor. Por fim, cria-se a arquitetura do Widget com as variáveis de classe.

Na figura 20 está representada a utilização do Widget 'MyCard'. Nesta situação o card gerado tem um título, subtítulo e leading definidos pelas informações dos users guardada na lista userList.

RestAPI's (Http):

```dart
class UserModel {
  int? id;
  String? email;
  String? firstName;
  String? lastName;
  String? avatar;

  UserModel({this.id, this.email, this.firstName, this.lastName, this.avatar});

  UserModel.fromJson(Map<String, dynamic> json) {
    id = json['id'];
    email = json['email'];
    firstName = json['first_name'];
    lastName = json['last_name'];
    avatar = json['avatar'];
  }
}
```

Figura 21Classe UserModel

```dart
import 'dart:convert';
import 'package:http/http.dart';
import '../Models/user.dart';

class UserRepository {
  String userUrl = 'https://reqres.in/api/users?page=2';

  Future<List<UserModel>> getUsers() async {
    Response response = await get(Uri.parse(userUrl));

    if (response.statusCode == 200) {
      final List result = jsonDecode(response.body)['data'];
      return result.map((e) => UserModel.fromJson(e)).toList();
    } else {
      throw Exception(response.reasonPhrase);
    }
  }
}
```

Figura 22 Classe UserRepository

Nas figuras 21 e 22 está representada a implementação inicial para o tratamento de uma restAPI. A API escolhida neste caso foi apenas uma simples API que funciona como base de dados para utilizadores, com informações como id, nome, email, etc.

Inicialmente foi necessário criar um modelo, representado pela class' UserModel' (fig. 21), que irá guardar as informações recebidas da API e um repositório, 'UserRepository' (fig.22), que ira tratar os dados recebidos da API através do uso da biblioteca http, que facilita o processo de criar pedidos a API através das suas funções inerentes.

O 'UserRepository' é constituído pela URL do endpoint da API e pelo método 'getUsers()'. Este método faz um pedido a API, através da utilização da biblioteca http, e verificar o status code da resposta. Caso seja igual a 200, o método executa o decoding da informação para uma lista através da função 'JsonDecode()' e posteriormente vai mapear cada um do item da lista anterior para um objecto 'UserModel', através da função 'UserModel.fromJson()' sendo o produto final uma lista de objetos 'UserModel'.

```dart
abstract class UserEvent extends Equatable {
  const UserEvent();
}

class LoadUserEvent extends UserEvent {
  @override
  List<Object> get props => [];
}
```

Figura 23 Class UserEvent (BLoC)

```dart
abstract class UserState extends Equatable {}

class UserLoadingState extends UserState {
  @override
  List<Object> get props => [];
}

class UserLoadedState extends UserState {
  final List<UserModel> users;
  UserLoadedState(this.users);
  @override
  List<Object> get props => [users];
}

class UserErrorState extends UserState {
  final String error;
  UserErrorState(this.error);
  @override
  List<Object> get props => [error];
}
```

Figura 24 Classe UserState (BLoC)

```dart
class UserBloc extends Bloc<UserEvent, UserState> {
  final UserRepository _userRepository;

  UserBloc(this._userRepository) : super(UserLoadingState()) {
    on<LoadUserEvent>((event, emit) async {
      emit(UserLoadingState());
      try {
        final users = await _userRepository.getUsers();
        emit(UserLoadedState(users));
      } catch (e) {
        emit(UserErrorState(e.toString()));
      }
    });
  }
}
```

Figura 25 Classe UserBloc (BLoC)

Nas figuras 23, 24 e 25 está representado o próximo passo na implementação de tratamento de RestAPI's. Neste passo da implementação, foi necessário desenvolver uma nova estrutura BLoC que permita lidar com as mudanças de estados nas estruturas desenhadas para lidar com a RestAPI.

Na figura 23 e 24 podemos ver as classes necessários para a atualização do estado e dos eventos bem como o tratamento de erros caso aconteçam.

Na figura 25 podemos ver a implementação da classe UserBloc que ira gerir os eventos e estados. Por exemplo, 'on<LoadUserEvent> o event handler ('UserLoadingState()') aguarda ate receber uma resposta da função 'getUsers()' e, se não houver erros, carrega um novo estado 'UserLoadedState()' com os dados recebidos pela função 'getUsers()'.

```
child: BlocBuilder<UserBloc, UserState>(
  builder: (context, state) {
    if (state is UserLoadingState) {
      return const Center(
        child: CircularProgressIndicator(),
      );
    }
    if (state is UserErrorState) {
      return const Center(child: Text('Error'));
    }
    if (state is UserLoadedState) {
      List<UserModel> userList = state.users;
      return ListView.builder(
        itemCount: userList.length,
        itemBuilder: (_, index) {
          return Padding(
              padding: const EdgeInsets.symmetric(
                  vertical: 4, horizontal: 8),
              child: Card(
                color: Theme.of(context).primaryColor,
                child: ListTile(
                  title: Text(
                    '${userList[index].firstName} ${userList[index].lastName}',
                    style: const TextStyle(
                        color: Colors.white),
                  ),
                  subtitle: Text(
                    '${userList[index].email}',
                    style: const TextStyle(
                        color: Colors.white),
                  ),
                  leading: CircleAvatar(
                    backgroundImage: NetworkImage(
                        userList[index].avatar.toString()),
                  ),
                ),
              ));
```

Figura 26 Utilização do UserBloc

Na figura 26 esta representada a arquitetura da representação gráfica dos users para o utilizador.

Como podemos ver é feita uma verificação do state da página e dependedo do resultado é apresentado diferentes cenários. O estado 'UserLoadingState' apresenta um indicador circular, o estado 'UserErrorState' apresenta um erro e o estado 'UserLoadedState' apresenta um Cards com as informações de cada um dos utilizadores registados na API, agrupando estes cards numa List, em que cada card corresponde a um index da lista.

GraphQL:

```dart
import 'package:graphql/client.dart';

class GraphQlService {
  late GraphQLClient _client;

  GraphQlService() {
    HttpLink link = HttpLink(
        'https://api.mocki.io/playground?endpoint=https://api.mocki.io/v2/c4d7a195/graphql');

    _client =
        GraphQLClient(link: link, cache: GraphQLCache(store: HiveStore()));
  }

  Future<QueryResult> runQuery(String query,
      {required Map<String, dynamic> variables}) async {
    QueryOptions options =
        QueryOptions(document: gql(query), variables: variables);

    final result = await _client.query(options);

    return result;
  }

  Future<QueryResult> runMutation(String query,
      {required Map<String, dynamic> variables}) async {
    MutationOptions options =
        MutationOptions(document: gql(query), variables: variables);

    final result = await _client.mutate(options);

    return result;
  }
}
```

Figura 27Classe GraphQlService

Na figura 27 podemos ver a fase inicial da implementação do GraphQL.

O processo inicial envolver criar uma pasta de serviços e o ficheiro graphql.dart. Neste ficheiro vamos criar a classe e os métodos necessários para consultar e realizar mutações no servido GraphQL.

É necessário criar um construtor da classe 'GraphQlService' que cria uma instância de 'GraphQLClient' para se conectar ao servidor GraphQL. É usado um objeto 'HttpLink' para definir a URL do servidor GraphQL que será consultado. Posteriormente, é criado o método 'runQuery' para consultar o servidor GraphQL e o método 'runMutation' para realizar mutações no servidor.

O método 'runQuery' recebe uma string e um mapa de variáveis que são usadas para construir um objeto 'QueryOptions'. Por fim, é utilizado o método 'query()' do '_client' para consultar o GraphQL, sendo o resultado da query um 'QueryResult'.

No caso do método 'runMutation', o processo é semelhante ao 'runQuery', no entanto o método utilizado para consultar o GraphQL é o 'mutate'. Tirando este pormenor, o processo é o mesmo.

```dart
abstract class GraphQlEvents extends Equatable {
  GraphQlEvents();

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

class FetchData extends GraphQlEvents {
  final String query;
  final Map<String, dynamic> variables;

  FetchData(this.query, {required this.variables}) : super();

  @override
  List<Object> get props => [query, variables];
}
```

Figura 28 Classe GraphQlEvents (BLoC)

```dart
abstract class GraphQlStates extends Equatable {
  GraphQlStates();

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

class LoadingData extends GraphQlStates {
  LoadingData() : super();
}

class LoadDataSuccess extends GraphQlStates {
  final dynamic data;

  LoadDataSuccess(this.data) : super();

  @override
  List<Object> get props => data;
}

class LoadDataError extends GraphQlStates {
  final dynamic error;

  LoadDataError(this.error) : super();

  @override
  List<Object> get props => error;
}
```

Figura 29 Classe GraphQlStates (BLoC)

```dart
class GraphqlBloc extends Bloc<GraphQlEvents, GraphQlStates> {
  late GraphQlService service;

  GraphqlBloc(super.initialState) {
    service = GraphQlService();
  }

  @override
  GraphQlStates get initialState => LoadingData();

  Stream<GraphQlStates> _mapFetchDataToState(FetchData event) async* {
    final query = event.query;
    final variables = event.variables;

    try {
      final result = await service.runMutation(query, variables: variables);

      if (result.hasException) {
        print('graphQlErrors: ${result.exception!.graphqlErrors.toString()}');
        //print('clientErrors: ${result.exception?.clientException.toString()}');
        yield LoadDataError(result.exception!.graphqlErrors[0]);
      } else {
        yield LoadDataError(result.data);
      }
    } catch (e) {
      print(e);
      yield LoadDataError(e.toString());
    }
  }

  Stream<GraphQlStates> mapEventToState(GraphQlEvents event) async* {
    if (event is FetchData) {
      yield* _mapFetchDataToState(event);
    }
  }
}
```

Figura 30 Classe GraphqlBloc (BLoC)

Nas figuras 28,29 e 30 estão representadas o próximo passo na implementação do tratamento de servidores GraphQL.

Neste passo, é necessário criar um BLoC que seja capaz de tratar das alterações de estado e eventos dos serviços definidos no passo anterior.

Na figura 28 está representada a criação da classe 'GraphQlEvents' que ira lidar com os eventos. A class 'FetchData' é uma subclasse da 'GraphQlEvents' e serve para ir buscar informação ao servidor GraphQL.

Na figura 29 está representada a criação da classe 'GraphQlStates' que ira lidar com os estados. Dentro desta classe, estão subclasses que serão responsáveis por definir os estados:

- 'LoadingData' – indica que a informação ainda não foi recebida (download).
- 'LoadDataSuccess' – indica que a informação foi recebida e vai ser exibida na lista on foi guardada.
- 'LoadDataError' – indica que ocorreu um erro e apresenta-o.

Na figura 30 está representada o BLoC que vai gerir os estados utilizando as classes de eventos, estados e serviços definidas anteriormente.

Neste BLoC está definida a class 'GraphqlBloc' e o seu construtor. É também definido o estado inicial (LoadingData()). Posteriormente são definidos dois métodos:

- '_mapFetchDataToState' - função assíncrona que recebe um evento 'FetchData' e retorna um Stream de estados 'GraphQlStates'. Dentro do método, são extraídos a query e as variables do evento recebido. Em seguida, é feita uma chamada ao método 'runMutation' do serviço 'GraphQlService' para executar a mutação no servidor GraphQL. Se o resultado contiver uma exceção ('result.hasException' é verdadeiro), o código imprime as mensagens de erro e emite o estado 'LoadDataError' com a primeira mensagem de erro encontrada. Se não houver exceção, o estado 'LoadDataError' é emitido com os dados retornados ('result.data').
- O método' mapEventToState' é um método obrigatório que mapeia os eventos recebidos para os estados correspondentes. Neste caso, ele verifica se o evento é uma instância de 'FetchData' e, se for, chama o método '_mapFetchDataToState' para obter o fluxo de estados relacionados.

```
return BlocBuilder<GraphqlBloc, GraphQlStates>(
    builder: (BuildContext context, GraphQlStates state) {
  if (state is LoadingData) {
    return const Scaffold(
      appBar: MyAppBar(text: 'GraphQL Page'),
      body: LinearProgressIndicator(),
    );
  } else if (state is LoadDataError) {
    return Scaffold(
      appBar: const MyAppBar(text: 'GraphQl Page'),
      body: Center(child: Text(state.error)),
    );
  } else {
    data = (state as LoadDataSuccess).data['users']['user'];
    return Scaffold(
        appBar: const MyAppBar(text: 'GraphQl Page'),
        body: ListView.builder(
            itemCount: data.length,
            itemBuilder: (BuildContext context, int index) {
              var item = data[index];
              return Card(
                elevation: 4.0,
                margin: const EdgeInsets.all(8.0),
                child: ListTile(
                  leading: Text(item['id']),
                  title: Text(item['name']),
                  trailing: Container(
                    padding: EdgeInsets.all(8.0),
                    decoration: BoxDecoration(
                      color: item['email'] == null
                          ? Colors.red.withOpacity(0.3)
                          : Colors.green.withOpacity(0.3),
                      borderRadius: BorderRadius.circular(30.0),
                    ),
                    child: Text(
                      item['email'],
                      style: TextStyle(
                          color: item['email'] == null
                              ? Colors.red
                              : Colors.green),
                    ),
```

Figura 31Utilização do GraphqlBloc

Na figura 31 está representada a estrutura gráfica criada para mostrar os dados obtidos do servidor GraphQL.

Como podemos observar, é feita uma verificação do state da página e dependedo do resultado é apresentado diferentes cenários. O estado 'LoadingData' apresenta um indicador linear (horizontal) enquanto a informação é carregada, o estado 'LoadDataError' apresenta um erro ao mostrar os dados e o estado 'LoadDataSuccess' apresenta um Cards com as informações de cada um dos utilizadores registados no servidor GraphQL, agrupando estes cards numa List, em que cada card corresponde a um index da lista.

Autenticação:

```
var sqlite = require('sqlite3');

var db = new sqlite.Database("users.sqlite3");

db.run(`CREATE TABLE users(
  id INTEGER PRIMARY KEY,
  username TEXT NOT NULL,
  password TEXT NOT NULL
)`);

db.close();
```

Figura 32 Criação da base de dados

```
var express = require('express');
var jwt = require('jsonwebtoken');
var sqlite = require('sqlite3');
var crypto = require('crypto');

const KEY = "myincredibly(!!1!11!)<'SECRET>)Key'!";

var db = new sqlite.Database("users.sqlite3");

const app = express();

app.use(express.urlencoded({extended: true}));

app.post('/signup', express.urlencoded(), (req, res) => {
    var password = crypto.createHash('sha256').update(req.body.password).digest('hex');
    db.get("SELECT FROM users WHERE username = ?", [req.body.username], function(err, row) {
        if(row != undefined ) {
          console.error("can't create user " + req.body.username);
          res.status(409);
          res.send("An user with that username already exists");
        } else {
          console.log("Can create user " + req.body.username);
          db.run('INSERT INTO users(username, password) VALUES (?, ?)', [req.body.username, password]);
          res.status(201);
          res.send("Success");
        }
      });
});
```

Figura 33 Rotas de autenticação

```
app.post('/login', express.urlencoded(), function(req, res) {
    console.log(req.body.username + " attempted login");
    var password = crypto.createHash('sha256').update(req.body.password).digest('hex');
    db.get("SELECT * FROM users WHERE (username, password) = (?, ?)", [req.body.username, password], function(err, row)
      if(row != undefined ) {
        var payload = {
          username: req.body.username,
        };

        var token = jwt.sign(payload, KEY, {algorithm: 'HS256', expiresIn: "15d"});
        console.log("Success");
        res.send(token);
      } else {
        console.error("Failure");
        res.status(401)
        res.send("There's no user matching that");
      }
    });
  });

  app.get('/data', function(req, res) {
    var str = req.get('Authorization');
    try {
      jwt.verify(str, KEY, {algorithm: 'HS256'});
      res.send("Very Secret Data");
    } catch {
      res.status(401);
      res.send("Bad Token");
    }
});

let port = process.env.PORT || 3000;
app.listen(port, function () {
    return console.log("Started user authentication server listening on port " + port);
});
```

Figura 34 Continuação de rotas de autenticação

Nas figuras 32,33 e 34 está representado o passo inicial para o processo de implementação de autenticação de Login. Neste caso, por uma questão de inovação decidi criar e implementar um servidor local em node.js.

Antes de criar qualquer ficheiro foi necessário iniciar um novo projeto em node.js e instalar as bibliotecas que vamos utilizar. Isto foi feito através da linha de comandos e dentro do diretório onde pretendia criar o servidor. Só após este passo é que começamos a criar os ficheiros necessários.

Começando pelo ficheiro representado na figura 32, criamos uma base de dados chamada users.sqlite3. Depois fazemos ‘db.run’ para criar uma tabela com as categorias id, username e password e fechamos a db.

Nas figuras 33 e 34 está representada a criação do servidor em Node.js usando a framework Express para lidar com a autenticação do utilizador, SQlite para interagir com a base de dados e Crypto para manipular hashes. É também definida inicialmente a

'KEY' que vai ser utilizada para verificar os tokens JWT (autenticação entre app e servidor) e feita a conexão com a base de dados 'users.sqlite3'.

Passo seguinte é definir as rotas que a autenticação vai utilizar:

- A rota POST "/signup"- Regista novos utilizadores. Recebe os dados enviados pelo formulário de registo, realiza o hash da password, verifica se o utilizador já existe, se existir envia uma mensagem de erro ao utilizador, e insere os dados do utilizador na base de dados caso não exista o utilizador.
- A rota POST "/login" – Autentica os utilizadores. Recebe os dados do formulário de login, realiza o hash da password, verifica se o username e a password correspondem a um utilizador existente, se não existir envia um erro ao utilizador, e gera um token JWT para autenticar o utilizador.
- A rota GET "/data" – Devolve dados a utilizadores autenticados. Recebe um token JWT, verifica se este é valido e, se for, retorna os dados. Se for inválido, envia um erro ao utilizador.

Por fim, o último passo é a configuração do port a qual o servidor deve ouvir e exibe uma mensagem na consola quando o servidor é iniciado.

```dart
import 'package:http/http.dart' as http;

const SERVER_IP = 'http://192.168.1.167:5000';

Future<String?> attemptLogIn(String username, String password) async {
  var res = await http.post(
    "$SERVER_IP/login" as Uri,
    body: {
      "username": username,
      "password": password,
    },
  );
  if (res.statusCode == 200) return res.body;
  return null;
}

Future<int> attemptSignUp(String username, String password) async {
  var res = await http.post(
    '$SERVER_IP/signup' as Uri,
    body: {
      'username': username,
      "password": password,
    },
  );
  return res.statusCode;
}
```

Figura 35 Funções de solicitação http

Na figura 35 está representado as funções que se utilizou para fazer solicitações HTTP ao servidor Node.js que criamos anteriormente.

Inicialmente definimos uma constante chamada SERVER_IP para armazenar o IP do servidor Node.js que criamos. Depois são criadas duas funções:

- 'attemptLogIn' – Recebe o username e a password como parâmetros e executa o método http.post para fazer uma solicitação POST à rota '$SERVER_IP/login' do servidor com os valores passados nos parâmetros. Após a solicitação ser feita, a resposta é armazenada na variável 'res' e verificada pelo seu statusCode.
- 'attemptSignUp' – Recebe o username e a password como parâmetros e executa o método http.post para fazer uma solicitação POST à rota '$SERVER_IP/signup' do servidor com os valores passados nos parâmetros.

```dart
import 'package:boilerplat/Pages/users_page.dart';
import 'package:boilerplat/Widgets/app_bar.dart';
import 'package:flutter/material.dart';
import '../Controllers/auth.dart';
import '../Widgets/dialog_widget.dart';
import '../main.dart';

class LoginPage extends StatelessWidget {
  final TextEditingController _usernameController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();

  LoginPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: const MyAppBar(
        text: 'Log in',
      ),
      body: Padding(
        padding: const EdgeInsets.all(8.0),
        child: Column(
          children: <Widget>[
            TextField(
              controller: _usernameController,
              decoration: const InputDecoration(
                labelText: 'Username',
              ),
            ),
            TextField(
              controller: _passwordController,
              obscureText: true,
              decoration: const InputDecoration(labelText: 'Password'),
            ),
```

Figura 36 Utilização das funções http

```
ElevatedButton(
  onPressed: () async {
    var username = _usernameController.text;
    var password = _passwordController.text;
    var jwt = await attemptLogIn(username, password);
    if (jwt != null) {
      storage.write(key: "jwt", value: jwt);
      Navigator.push(
          context,
          MaterialPageRoute(
              builder: (context) => UsersPage.fromBase64(jwt)));
    } else {
      displayDialog(
        context,
        'An Error Ocurred',
        'No account was found matching that username or password',
      );
    }
  },
  child: const Text('Log in'),
),
```

Figura 37 Continuação da utilização das funções http

```
            ElevatedButton(
              onPressed: () async {
                var username = _usernameController.text;
                var password = _passwordController.text;

                if (username.length < 4) {
                  displayDialog(context, 'Invalid Username',
                      "The username needs to be at least 4 chars long");
                } else if (password.length < 8) {
                  displayDialog(context, 'Invalid Password',
                      'The password should be at least 8 chars long');
                } else {
                  var res = await attemptSignUp(username, password);
                  if (res == 201) {
                    displayDialog(context, 'Signed Up!',
                        'The user was created. Log in now');
                  } else if (res == 409) {
                    displayDialog(context, 'User already registered',
                        'Log in with your existing account!');
                  } else {
                    displayDialog(context, 'Error', 'Unknown error occured');
                  }
                }
              },
              child: const Text('Sign Up!'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Figura 38 Continuação da utilização das funções http

Por fim, nas figuras 36, 37 e 38 é representado o último passo na implementação de autenticação de Login. Neste passo é implementado a página que lida com o login na aplicação e o registo de um novo utilizador.

Inicialmente, é necessário definir a class 'LoginPage' que é subclasse da classe 'StatelessWidget'. Posteriormente, são definidos os controladores de texto que serão responsáveis por armazenar os dados inseridos para registo e o construtor da class 'LoginPage'.

A próxima fase é construir a arquitetura da página e organizar os elementos. São criados dois 'TextFields' para que o utilizador possa introduzir informação referente ao

seu username e a sua password, um botão que, quando pressionado, executa a função 'attemptLogin' e um botão que, quando pressionado, executa a função 'attemptSignUp'.

No caso do login, se não existirem erros e a autenticação (jwt) não for negativa, o utilizador é redirecionado para a página 'UsersPage' e o jwt é guardado na storage do dispositivo. Caso a autenticação dê erro então é exibido um dialog com o texto referente a este erro.

No caso do registo de um utilizador, se o utilizador não tiver já registado é registado com a informação que forneceu e é exibido um dialog a confirmação o registo. Caso exista um erro, ele é tratado de duas maneiras: se for um erro tipo 409(já existe o utilizador na base de dados) é exibido um dialog com o aviso que já existe o utilizador ou se for um erro diferente então é exibido um dialog que avisa para o erro se definir que tipo de erro é.

FlutterSecureStorage:

```
const storage = FlutterSecureStorage();
```

Figura 39 Variável da instância FlutterSecureStorage

```
Future<String> get jwtOrEmpty async {
  var jwt = await storage.read(key: "jwt");
  if (jwt == null) return "";
  return jwt;
}
```

Figura 40 Função de verificação do JWT

```
future: jwtOrEmpty,
builder: (context, snapshot) {
  if (!snapshot.hasData) return const CircularProgressIndicator();
  if (snapshot.data != '') {
    var str = snapshot.data;
    var jwt = str?.split('.');

    if (jwt?.length != 3) {
      return LoginPage();
    } else {
      var payload = json.decode(
          ascii.decode(base64.decode(base64.normalize(jwt![1]))));
      if (DateTime.fromMillisecondsSinceEpoch(payload["exp"] * 1000)
          .isAfter(DateTime.now())) {
        return UsersPage(str!, payload);
      } else {
        return LoginPage();
      }
    }
  } else {
    return LoginPage();
  }
```

Figura 41 Verificar JWT e apresentar páginas ao utilizador

Nas figuras 39,40 e 41 está representado a implementação da biblioteca FlutterSecureStorage.

Inicialmente é necessário criar uma variável 'storage' da instância FlutterSecureStorage() (fig. 39). Esta variável é a variável que é usada para guardar o token JWT como já vimos na secção analise de resultados do JWT.

O próximo passo é definir uma função que verifica o conteúdo armazenado no FlutterSecureStorage (fig. 40). Se o valor for nulo, significa que não há um token JWT armazenado no FlutterSecureStorage. Nesse caso, a função retorna uma string vazia "". Caso contrário, ou seja, se jwt não for nulo, a função retorna o valor do token JWT armazenado.

Por fim, o último passo é utilizado a função criada anteriormente para definir que página apresentar a utilizador (fig. 41). Através de diversas condições: verificar se o JWT é uma string vazia ou não, se não for dividir o JWT em 3 partes e verificar se a divisão foi feita corretamente. Depois descodifica, normaliza e guarda na variável 'payload' o JWT e compara o campo exp do payload com o momento atual para decidir que página apresentar ao utilizador. Caso o JWT seja vazio, então o utilizador é redirecionado para a página Login.

Conclusões

Para concluir, este projeto atingiu os objetivos inicialmente estabelecidos. Com bastante trabalho e suporte da empresa foi criado um Boilerplate capaz de agilizar o tempo de criação de uma aplicação em Flutter. Foi evidenciada a importância de ferramentas como Boilerplate para empresas que produzam constantemente aplicações.

As funcionalidades previstas foram implementadas de formas diferentes, dado também flexibilidade ao utilizar da ferramenta para ver as mais adequadas para situações específicas.

Foi um projeto interessante, do qual levo bastante conhecimento bem como conhecimento/dicas que apenas seriam possíveis adquirido num ambiente profissional.

Por entato, confiantemente, posso também acrescentar que as minhas expectativas do estágio foram cumpridas, levando desta experiência não só conhecimento como também memórias.

Glossário

iOS – Sistema operativo da Apple Inc, usado em dispositivos da marca (iPhone, iPod, iPad, etc).

Android - Sistema operativo baseado em Linux usado para em dispositivos móveis (smartphones e tablets).

Dart – Linguagem de programação multi-paradigma criada pela Google.

Flutter – Framework criada pela Google para desenvolver aplicações móveis.

Packages – Bibliotecas com código modular que pode ser utilizado por outros utilizadores.

Tasks – Tarefas a realizar/cumprir.

Referências bibliográficas

Termosa, S. (2008). An Introduction to Flutter: The Basics. https://www.freecodecamp.org/news/an-introduction-to-flutter-the-basics-9fe541fd39e2/

Minh, N. (2022). Flutter — Building a perfect Boilerplate Project from scratch. https://medium.com/@NALSengineering/flutter-building-a-perfect-boilerplate-project-from-scratch-8a0a92429614

flutter_bloc 8.1.3. (2023, 10 de maio). Recuperado em 10 de maio de 2023, de https://pub.dev/packages/flutter_bloc

AppBar class. (2023, 17 de maio). Recuperado em 17 de maio de 2023, de https://api.flutter.dev/flutter/material/AppBar-class.html

BottomNavigationBar class. (2023, 17 de maio). Recuperado em 17 de maio de 2023, de https://api.flutter.dev/flutter/material/BottomNavigationBar-class.html

New Buttons and Button Themes. (2023, 19 de maio). Recuperado em 19 de maio de 2023, de https://docs.flutter.dev/release/breaking-changes/buttons

Forms. (2023, 19 de maio). Recuperado em 19 de maio de 2023, de https://docs.flutter.dev/cookbook/forms

Dialog Class (2023, 21 de maio). Recuperado em 21 de maio de 2023, de https://api.flutter.dev/flutter/material/Dialog-class.html

Create lists with different types of items. (2023, 21 de maio). Recuperado em 21 de maio de 2023, de https://docs.flutter.dev/cookbook/lists/mixed-list

Card class. (2023, 22 de maio). Recuperado em 22 de maio de 2023, de https://api.flutter.dev/flutter/material/Card-class.html

http 1.1.0. (2023, 29 de maio). Recuperado em 29 de maio de 2023, de https://pub.dev/packages/http

graphql 5.1.3. (2023, 2 de junho). Recuperado em 2 de junho de 2023, de https://pub.dev/packages/graphql

Introduction to JSON Web Tokens. (2023, 6 de junho). Recuperado a 6 de junho de 2023 de https://jwt.io/introduction

flutter_secure_storage 8.0.0. (2023, 8 de junho). Recuperado a 8 de junho de 2023 de https://pub.dev/packages/flutter_secure_storage

Anexos

Anexo 1:

Identificador:	PROJETO-TESTE-BLoC/CUBIT
Objectivo:	Verificar a implementação do BLoC/Cubit e conferir se os estados das páginas estão a ser atualizados corretamente
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criação de subclasses BLoC • Criação de classes e subclasses de eventos BLoC • Criação de classes e subclasses de estados BLoC • Criação da subclasse Cubit • Criação da classe e subclasses de navegação Cubit	
Especificações de Saídas: • Update do state da página • Navegação para outra página • Apresentação de Erros em caso de não existir dados correspondentes.	

Tabela 2 Teste BLoC/CUBIT

Anexo 2:

Identificador:	PROJETO-TESTE-APPBAR
Objectivo:	Verificar a implementação do Widget AppBar de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar class MyAppBar • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget AppBar com diferentes características baseadas nas especificações	

Tabela 3 Teste AppBar

Anexo 3:

Identificador:	PROJETO-TESTE-BOTTOM-NAVIGATION-BAR
Objectivo:	Verificar a implementação do Widget BottomNavigationBar de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class MyBottomNavigationBar • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget BottomNavigationBars com diferentes características baseadas nas especificações	

Tabela 4 Teste BottomNavigationBar

Anexo 4:

Identificador:	PROJETO-TESTE-BUTTONS
Objectivo:	Verificar a implementação do Widget Button de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class MyButton • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget Button com diferentes características baseadas nas especificações	

Tabela 5 Teste Buttons

Anexo 5:

Identificador:	PROJETO-TESTE- FORM
Objectivo:	Verificar a implementação do Widget Form de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class MyForm • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget Form com diferentes características baseadas nas especificações.	

Tabela 6 Teste Form

Anexo 6:

Identificador:	PROJETO-TESTE- DIALOGS
Objectivo:	Verificar a implementação do Widget Dialogs de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class displayDialog e Dialogs • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget Dialogs com diferentes características baseadas nas especificações.	

Tabela 7 Teste Dialogs

Anexo 7:

Identificador:	PROJETO-TESTE- LISTS
Objectivo:	Verificar a implementação do Widget Lists de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class MyList • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget List com diferentes características baseadas nas especificações.	

Tabela 8 Teste Lists

Anexo 8:

Identificador:	PROJETO-TESTE- CARDS
Objectivo:	Verificar a implementação do Widget Cards de forma a ser reutilizável.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class MyCard • Variáveis de costumização • Arquitetura do Widget	
Especificações de Saídas: • Criação do Widget Card com diferentes características baseadas nas especificações.	

Tabela 9 Teste Cards

Anexo 9:

Identificador:	PROJETO-TESTE- HTTP
Objectivo:	Verificar a implementação da biblioteca HTTP e análise de dados obtidos pelos requests a API
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class UserRepository, UserModel, UserBloc, UserEvents e UserState • Definir métodos para fazer request a API, tratamento de Json e Login • Condições para tratamento de erros em requests • Criação da página users_page	
Especificações de Saídas: • Apresentação de um grupo de dados provenientes do request feito a API • Erro caso exista alguma inconformidade no request.	

Tabela 10 Teste Http

Anexo 10:

Identificador:	PROJETO-TESTE- GRAPHQL
Objectivo:	Verificar a implementação da biblioteca GraphQL e análise dos dados obtidos
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criar a class GraphQlService, GraphqlBloc, GraphQlEvents e GraphQlStates • Definir métodos para correr Queries ou Mutações • Condições para tratamento de erros de queries/mutações • Criação da página Graphql_page	
Especificações de Saídas: • Apresentação dos dados provenientes de uma query/mutação a base de dados • Erro em casa de inconformidades nas quaries/mutações	

Tabela 11 Teste GraphQL

Anexo 11:

Identificador:	PROJETO-TESTE- JWT
Objectivo:	Verificar a implementação da biblioteca JWT
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criação da variável token que implementa a biblioteca jwt • Implementação de condicionais para verificar a variável token • Respostas aos pedidos enviados ao server em node.js baseado em status	
Especificações de Saídas: • Apresentação da página Users após um login bem-sucedido. • Apresentação de erros conforme o status code definido para cada erro.	

Tabela 12 Teste JWT

Anexo 12:

Identificador:	PROJETO-TESTE- FLUTTER-SECURE-STORAGE
Objectivo:	Verificar a implementação da biblioteca FlutterSecureStorage.
Autor(es):	Alexandre Marques Machado
Especificações de Entradas: • Criação de uma variável storage que implementa FlutterSecureStorage • Dar conteúdo a variável storage após o primeiro Login bem-sucedido. • Criação de métodos para verificar o conteúdo da variável storage e navegar para diferentes páginas baseado em condições.	
Especificações de Saídas: • Quando o utilizador abre a aplicação, existe load automático da página Users caso o FlutterSecureStorage contenha o jwt do Login. • Se for a primeira vez que utiliza a aplicação, o utilizador é direcionado para a página de Login.	

Tabela 13 Teste FlutterStorageSecure