UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
SISTEMAS DE INFORMAÇÃO

Luiz Henrique Andrade Souza
João Victor Munhóz dos Santos

Desenvolvimento de Apps Mobile

CAMPO GRANDE

2023

Luiz Henrique Andrade Souza
João Victor Munhóz dos Santos

Desenvolvimento de Apps Mobile

Atividade Orientada de Ensino do Curso de Sistemas de Informação da Faculdade de Computação da Universidade Federal de Mato Grosso do Sul, como Componente Curricular Não Disciplinar optativo para obtenção de diploma de bacharel em Sistemas de Informação.

Orientadora: Professora Dra. Ana Karina Dourado Salina de Oliveira

CAMPO GRANDE

2023

RESUMO

O desenvolvimento de aplicativos móveis é algo cada vez mais comum no mercado de trabalho, com sua demanda crescendo exponencialmente. Para tanto, diversas linguagens de programação, *frameworks*, padrões e modelos foram desenvolvidos, visando fornecer métodos e ferramentas para facilitar esse desenvolvimento. Este artigo explora o uso da linguagem de programação Dart, o *framework* Flutter, a arquitetura *Model-View-Controller* (MVC) e o Modelo de Maturidade de Richardson, destacando sua importância, vantagens e desvantagens e principais pontos, trazendo uma compreensão detalhada que facilite o aprimoramento das práticas de desenvolvimento de aplicativos móveis. É feito também um estudo de caso com o aplicativo gEstoque.

Palavras-chave: Aplicativos Android, Aplicativos iOS, Desenvolvimento Híbrido, Desenvolvimento Nativo, Linguagem de Programação Dart, Framework Flutter, Padrão de arquitetura Model-View-Controller, Padrão MVC, Modelo de Maturidade de Richardson.

LISTA DE TABELAS

Tabela 1 - Descrição das requisições HTTP.....	17
--	----

SUMÁRIO

1. INTRODUÇÃO.....	6
2. OBJETIVOS.....	7
2.1 OBJETIVO GERAL.....	7
2.2 OBJETIVOS ESPECÍFICOS.....	7
3. REVISÃO BIBLIOGRÁFICA.....	8
3.1 PROGRAMAÇÃO ORIENTADA A OBJETOS.....	8
3.1.1 Herança.....	8
3.1.2 Abstração.....	8
3.1.3 Polimorfismo.....	9
3.1.4 Encapsulamento.....	9
3.2 API RESTFUL.....	9
3.2.1 APIs.....	9
3.2.2 REST.....	9
3.2.3 O que é API RESTful.....	10
3.3 DESENVOLVIMENTO NATIVO X HÍBRIDO.....	10
3.3.1 Desenvolvimento Nativo.....	10
3.3.2 Desenvolvimento Híbrido.....	10
4. ANÁLISE DOS COMPONENTES.....	12
4.1 LINGUAGEM DE PROGRAMAÇÃO DART.....	12
4.1.1 Origem.....	12
4.1.2 Características e Funcionalidades.....	12
4.2 FRAMEWORK FLUTTER.....	13
4.2.1 Origem.....	13
4.2.2 Características.....	13
4.2.3 Desenvolvimento Cross-Platform.....	14
4.2.4 Exemplos de Casos de Uso do Flutter.....	14
4.3 ARQUITETURA MODEL-VIEW-CONTROLLER.....	15

4.3.1 Características.....	15
4.3.2 Fluxo de Funcionamento.....	15
4.3.3 Vantagens.....	15
4.4 MODELO DE MATURIDADE DE RICHARDSON.....	15
4.4.1 Nível 0 - POX (Plain Old XML).....	16
4.4.2 Nível 1 - Recursos (Resources).....	16
4.4.3 Nível 2 - Verbos HTTP.....	16
4.4.4 Nível 3 - HATEOAS (Hypermedia as the Engine of Application State).....	16
5. INTEGRAÇÃO DOS COMPONENTES.....	17
5.1 CONSUMO DE API EM DART/FLUTTER.....	17
5.2 ARQUITETURA MVC EM DART/FLUTTER.....	18
6. ESTUDO DE CASO: APLICATIVO GESTOQUE.....	20
6.1 ANÁLISE DA APLICAÇÃO.....	20
6.2 APLICAÇÃO DO MODELO DE RICHARDSON.....	21
6.3 PORTABILIDADE PARA IOS.....	21
7. CONCLUSÃO.....	22
7.1 TRABALHOS FUTUROS.....	22
REFERÊNCIAS.....	24

1. INTRODUÇÃO

O avanço constante e exponencial da tecnologia vem trazendo transformações significativas no uso e interação com dispositivos móveis, buscando cada vez mais eficiência, usabilidade e inovação. Conseqüentemente, diversas ferramentas e metodologias vêm sendo desenvolvidas, buscando um espaço cada vez maior no mercado e na comunidade científica e acadêmica.

Uma das linguagens de programação mais utilizadas no desenvolvimento de aplicativos é o Dart, notável por sua agilidade, eficiência e facilidade de aprendizado. O uso em conjunto com o *framework* Flutter, torna a ferramenta altamente versátil, com forte desempenho, suporte e produtividade em diversas plataformas.

O uso do padrão de arquitetura *Model-View-Controller* (MVC) organiza o desenvolvimento de modo a facilitar a manutenção, reusabilidade e escalabilidade do código através de uma estrutura de programação flexível, separando as responsabilidades em três camadas (TREINAWEB. Acesso em: 11-08-2023).

O Modelo de Maturidade de Richardson trata-se de uma abordagem para o design de evolução em APIs RESTful, propondo quatro níveis de avaliação que classifiquem, entre outros elementos, a simplicidade e flexibilidade na interação entre cliente e servidor (JUNIOR. Acesso em: 29-11-2023).

Este trabalho analisa a combinação dos elementos descritos com uma abordagem coesa e eficiente para o desenvolvimento de aplicações robustas e satisfatórias, detalhando a linguagem base de programação, a manutenção, a organização, a estrutura e a flexibilidade.

É realizado também um estudo de caso com o aplicativo gEstoque, aplicativo de gerenciamento de estoque da empresa Soldamaq, disponível para sistemas operacionais Android. É analisada a estrutura do código e a implementação do mesmo de acordo com os padrões descritos neste trabalho, bem como avaliada a possibilidade de *port* para plataformas iOS, estudando e destacando os requisitos a serem implementados.

2. OBJETIVOS

2.1 OBJETIVO GERAL

Este trabalho tem como objetivo realizar uma análise teórica do uso da linguagem de programação Dart, o *framework* Flutter, o modelo arquitetural *Model-View-Controller* (MVC) e o Modelo de Maturidade de Richardson, analisando sua viabilidade, simplicidade, qualidade e eficiência no desenvolvimento de aplicativos móveis, bem como realizar um estudo de caso com o aplicativo gEstoque, avaliando a implementação dos padrões estruturais descritos e a viabilidade de *port* para sistemas iOS.

2.2 OBJETIVOS ESPECÍFICOS

- Definir os conceitos básicos necessários para a compreensão das ferramentas analisadas no trabalho.
- Analisar a linguagem de programação Dart, explorando suas características fundamentais, sintaxe, estrutura, vantagens e desvantagens.
- Analisar o *framework* Flutter, examinando seus recursos e funcionalidades e a eficiência em seu uso no desenvolvimento.
- Analisar o modelo arquitetural *Model-View-Controller* (MVC), explorando seus princípios, características, aplicabilidade e vantagens de organização e manutenção do código
- Analisar o Modelo de Maturidade de Richardson, examinando seus diferentes níveis de maturidade e o impacto de sua incorporação na flexibilidade e maturidade de APIs Restful.
- Avaliar a viabilidade, simplicidade, qualidade e eficiência do código no uso conjunto de Dart, Flutter, MVC e Modelo de Richardson, destacando as vantagens e desvantagens em desempenho e produtividade.
- Estudar um Caso de Uso que incorpore os elementos de Dart, Flutter e MVC, e checar a viabilidade de classificação pelo Modelo de Richardson.
- Analisar a possibilidade de disponibilizar o aplicativo estudado em sistemas iOS, definindo os requisitos necessários.

3. REVISÃO BIBLIOGRÁFICA

3.1 PROGRAMAÇÃO ORIENTADA A OBJETOS

O paradigma de programação de orientação a objetos (POO) é baseado na aproximação entre os conceitos do mundo real e a programação. Nesse modelo, são criadas classes que podem instanciar objetos, cada um possuindo características e comportamentos próprios, definidos por meio de métodos e atributos. A orientação a objetos surgiu como uma alternativa à programação estruturada, sendo amplamente adotada no mercado devido, entre outras vantagens, à sua organização e facilidade de manutenção de código (NOVAGEO. 2023).

A linguagem de programação Dart é considerada uma linguagem orientada a objetos, seguindo seus princípios e sua estrutura, e faz uso dos conceitos básicos desse paradigma que tornam o código modular, reutilizável e de fácil entendimento (TREINAWEB. 2023). Os quatro pilares da programação orientada a objetos são apresentados a seguir:

3.1.1 Herança

Trata-se da criação de novas classes no código as quais herdam o comportamento e características de uma classe já existente, denominada superclasse, especializando a classe herdada com características próprias, na forma de métodos, atributos e sua implementação (MARTIN, 2009). Um exemplo: a classe Veículo pode ser herdada pelas classes Caminhão ou Ônibus, ambas as quais compartilham as características apresentadas em Veículo, em adição às suas próprias.

3.1.2 Abstração

Refere-se a capacidade de criar classes e métodos de maneira abstrata, ou seja, não podem ser diretamente instanciados em objetos. Essas classes servem unicamente como superclasses, contendo características comuns que serão compartilhadas por todas as suas subclasses. Possuem métodos abstratos, os quais não possuem qualquer implementação, sendo caracterizados e implementados exclusivamente nas subclasses, permitindo a adaptação e especialização do comportamento conforme necessário. Essa prática contribui para a modularidade e flexibilidade do código em sistemas orientados a objetos (DEITEL, 2017).

3.1.3 Polimorfismo

É a capacidade de objetos pertencentes a subclasses diferentes, mas que compartilham uma superclasse comum, responderem às mesmas chamadas de métodos. Cada subclasse tem a liberdade de caracterizar e implementar o método de acordo com suas necessidades, assim, uma única chamada de método pode ser utilizada para invocar métodos distintos, definidos pelo compilador através dos parâmetros ou da subclasse instanciada no objeto. Essa característica proporciona flexibilidade e simplicidade ao código, tornando-o mais adaptável a diferentes situações (DEITEL, 2017).

3.1.4 Encapsulamento

Refere-se à capacidade de uma classe ocultar seus métodos e atributos, definindo-os como “*private*”. Assim, esses elementos só podem ser acessados ou modificados por dentro da própria classe, impedindo o acesso direto de outros objetos, que passam a depender de outros métodos públicos como *getters* (para retornar valor dos atributos) e *setters* (para modificar o valor dos atributos). Isso traz segurança e preserva a integridade do sistema, além de impedir modificações indevidas (DEITEL, 2017).

3.2 API RESTFUL

3.2.1 APIs

O termo API (Interface de Programação de Aplicações) refere-se ao conjunto de regras e protocolos responsáveis pela interação entre *softwares*. Elas definem as maneiras em que os sistemas podem se comunicar e trocar informações, especificando os métodos de requisições, formatos de dados e *endpoints* (URLs) que acessem os recursos e serviços dos sistemas (BHIEL, 2016).

3.2.2 REST

REST (*Representational State Transfer*) é um modelo de arquitetura desenvolvido por Roy Fielding em sua tese de doutorado, que define princípios de comunicação entre sistemas distribuídos *Web* de maneira flexível e escalável. Ele faz uso de serviços disponíveis no protocolo HTTP para realizar operações sobre os recursos utilizados. É amplamente utilizado devido a sua simplicidade, padronização, escalabilidade e facilidade de integração e comunicação entre os diferentes sistemas (ALURA, 2023).

3.2.3 O que é API RESTful

Uma API RESTful é uma API que aplica os princípios e conceitos do REST, identificando os dados e serviços utilizados através de URLs únicas, e realizando operações do padrão HTTP (GET, POST, PUT, DELETE) sobre esses recursos. Ela mantém a independência entre cliente e servidor e utiliza uma comunicação *stateless*, ou seja, cada solicitação de dados ou recursos do cliente contém todas as informações necessárias para que seja processada e atendida (HOSTINGER. 2023).

A manipulação dos recursos através de uma API RESTful pode se dar através de diferentes representações, tais como JSON, XML ou HTML, podendo o cliente escolher a que lhe seja mais adequada. Opera também em um sistema de camadas, com cada camada responsável por uma funcionalidade específica (HOSTINGER. 2023). A simplicidade, eficiência, escalabilidade e flexibilidade são alguns dos benefícios existentes no uso de uma API RESTful.

3.3 DESENVOLVIMENTO NATIVO X HÍBRIDO

Há diversos tipos de aplicativos *mobile*. Alguns exclusivos para sistemas Android, alguns para sistemas iOS, e outros que funcionam em ambos. Assim, o desenvolvimento de aplicações móveis pode ser classificado entre híbrido e nativo.

3.3.1 Desenvolvimento Nativo

No desenvolvimento nativo, o sistema é desenvolvido para um sistema operacional específico: Android ou iOS. Esse modo de desenvolvimento geralmente faz uso de linguagens e *frameworks* exclusivos e mais recomendados para a plataforma específica. Entre as vantagens há um melhor desempenho, maior acesso aos recursos disponíveis nas ferramentas utilizadas e melhor integração com o sistema operacional (ALURA. 2023). Entretanto, no caso de um *port* para outro sistema operacional, há um trabalho e dificuldades maiores para realizar a adaptação, necessitando muitas vezes realizar uma engenharia reversa do código.

3.3.2 Desenvolvimento Híbrido

No desenvolvimento híbrido, a aplicação criada pode ser implementada em diferentes sistemas operacionais, e assim, faz uso de linguagens, *frameworks* e outros recursos com suportes para ambos os sistemas: Android e iOS. O código pode ser compartilhado entre diferentes plataformas, poupando tempo e dinheiro do desenvolvedor, e utiliza contêineres

nativos para execução (ALURA. 2023). Porém, pode haver dificuldades em realizar implementações complexas e específicas para plataformas, dependendo de recursos e funcionalidades de ferramentas externas, como as oferecidas pelo *framework* Flutter

4. ANÁLISE DOS COMPONENTES

4.1 LINGUAGEM DE PROGRAMAÇÃO DART

4.1.1 Origem

Dart é uma linguagem de programação criada pela Google e lançada em 2011, surgindo como uma alternativa ao JavaScript que buscava trazer mais eficiência para o desenvolvimento de aplicações *Web*, trazendo incorporando a orientação a objetos e buscando superar limitações estruturais existentes (TREINAWEB. 2023).

Entretanto, Dart não foi completamente aceita pela comunidade como substituto do JavaScript, e foi recebida com relutância. Ao longo dos anos, com várias atualizações significativas e a incorporação do *framework* Flutter, passou a ser amplamente utilizada no desenvolvimento de aplicações móveis, o qual continua sendo seu maior nicho (TREINAWEB. 2023).

Dart possui muitas similaridades na sintaxe e estrutura com linguagens de programação já existentes e muito utilizadas até então, como Java e C#, logo, possui uma curva de aprendizagem muito eficiente para desenvolvedores já familiarizados com essas linguagens. Entre as vantagens do Dart em relação a elas, estão a sintaxe mais dinâmica, flexível e legível, além da capacidade de compilar em código nativo (TREINAWEB. 2023).

4.1.2 Características e Funcionalidades

O Dart traz dois modos de compilação distintos:

- JIT (*just-in-time*): o código é compilado e traduzido para linguagem de máquina com a aplicação em execução, trazendo um ciclo de desenvolvimento mais rápido e aplicando imediatamente qualquer alteração no código, porém tem custo de desempenho e inicialização mais lentos (TREINAWEB. 2023).
- AOT (*Ahead-Of-Time*): o código é compilado antes da execução, traduzindo para linguagem de máquina. Ele traz um desempenho melhor, gastando menos tempo na execução, já que a mesma não faz a tradução do código. Porém possui um tempo de compilação maior, e qualquer alteração no código exigiria uma recompilação completa (TREINAWEB. 2023).

O Dart usa o modelo de gerenciamento de memória, o qual utiliza um coletor de lixo (*garbage collector*), responsável por identificar e limpar objetos não utilizados na memória, automaticamente liberando o espaço alocado. Isso traz um desempenho maior para o aplicativo e reduz a necessidade de gerenciamento manual da memória por parte do desenvolvedor (MEDIUM. 2023).

O Dart utiliza programação assíncrona e concorrente, a qual utiliza um modelo baseado em *isolates*, unidades leves de processamento, que permitem a execução paralela do código fazendo uso de *threads* separadas. Assim, cada *isolate* opera uma pilha própria de execução, e interage com outros *isolates* através de mensagens, evitando problemas de concorrência como bloqueio de *threads* e permitindo que ações intensivas sejam realizadas sem prejuízo ao desempenho geral da aplicação. Essas características de gerenciamento de memória e programação concorrente são destaques do Dart no desenvolvimento eficiente de aplicativos (MEDIUM. 2023).

4.2 FRAMEWORK FLUTTER

4.2.1 Origem

Desenvolvido pela Google em 2017, Flutter é um *framework* (ferramenta) de código aberto para criação de interfaces de usuário de forma versátil e compatível com múltiplas plataformas. Foi criado buscando superar limitações existentes na época na linguagem de programação a qual é baseado, o Dart, traz várias ferramentas, vantagens e *widgets* personalizáveis, criando um ambiente de desenvolvimento ágil e eficiente que o tornaram uma ferramenta muito inovadora e popular no mercado (AWS AMAZON. 2023).

4.2.2 Características

O Flutter faz uso de diversos *widgets*, componentes reutilizáveis que permitem a construção de interfaces complexas de maneira fácil, eficiente e modular. Eles são representados por elementos como botões, caixas de textos e outros elementos mais complexos de design, com alta versatilidade e flexibilidade. São organizados através de uma árvore de *widgets*, com uma estrutura hierárquica. Assim, cada *widget* pode ser criado e testado independentemente, trazendo facilidade de construção, manutenção e atualização de interfaces.

O Flutter também se destaca por utilizar o modo de compilação AOT, traduzindo o código da linguagem Dart para a linguagem nativa antes da execução, porém ainda assim faz uso do *Hot Reload*, uma funcionalidade que se baseia no modo JIT. Assim, desenvolvedores podem implementar e testar alterações no código sem reiniciar a aplicação, visto que as atualizações no código serão refletidas instantaneamente no ambiente de execução. Durante o desenvolvimento, o *Hot Reload* é usado para implementar e desenvolver o código, e após a implementação final do aplicativo, o modo AOT é usado para melhorar o desempenho (FLUTTER. 2023).

4.2.3 Desenvolvimento Cross-Platform

O Flutter é uma ferramenta de desenvolvimento *cross-platform*, a capacidade de executar aplicações em diferentes plataformas utilizando o mesmo código fonte, e assim traz diversas funcionalidades que facilitem esse fim. Faz uso de uma única linguagem de programação, Dart, utilizável em Android, iOS, *Web* e *Desktop*. Assim, uma única base de código pode ser utilizada e aproveitada em diferentes plataformas (FLUTTER. 2023)..

O Flutter com APIs nativas, logo, embora forneça uma experiência unificada e funcional para múltiplas plataformas, ainda é possível acessar diretamente as APIs e integrar recursos e funcionalidades específicas da plataforma conforme necessário, diretamente na aplicação (FLUTTER. 2023).

O Flutter oferece suporte à Integração Contínua, trazendo facilidade de automação, testes automatizados, ambientes reproduzíveis e customizações avançadas (ALURA. 2023). Isso, juntamente com a funcionalidade *Hot Reload*, *Widgets* personalizáveis e uma comunidade ativa e em constante expansão, tornam o Flutter uma ferramenta diferenciada, comparada com outras soluções de desenvolvimento *cross-platform*.

4.2.4 Exemplos de Casos de Uso do Flutter

Diversas empresas e aplicativos muito conhecidos no mundo utilizam o Flutter, tais como o Nubank, a BMW, através de seu aplicativo ConnectedDrive, Alibaba, Google Ads, entre muitos outros.

4.3 ARQUITETURA MODEL-VIEW-CONTROLLER

4.3.1 Características

O modelo de arquitetura *model-view-controller* (MVC) é um dos padrões de estrutura de *software* mais utilizados no mercado. Ele faz a separação da lógica de negócio, da manipulação dos dados e da interface do usuário, através de camadas (TREINAWEB. 2023). As camadas são divididas em:

- **Model:** Camada responsável por gerenciar e manipular os dados de acordo com a lógica do negócio, executando operações e respondendo consultas do usuário repassadas pelas outras camadas.
- **View:** Representa a interface em que o usuário interage, exibindo as informações recebidas das outras camadas e enviando requisições conforme as ações do usuário.
- **Controller:** Serve como a camada de controle e interação entre as outras camadas, armazenando a lógica de manipulação de dados e interface do usuário. Ela recebe e processa os *inputs* da camada *View* e atualiza o *Model*.

4.3.2 Fluxo de Funcionamento

Inicialmente, o usuário interage com a camada *View*, colocando suas requisições e comandos de acordo com suas necessidades. A camada *Controller* recebe e processa essas entradas, atualizando a *Model* conforme necessário, que notifica as mudanças à camada *View*. Uma vez notificado, a *View* atualiza a interface do usuário de acordo com o novo estado do código (TREINAWEB. 2023).

4.3.3 Vantagens

A separação entre as três camadas do MVC traz facilidade na manutenção e escalabilidade ao sistema, permitindo a reusabilidade do código, realização de testes independentes em cada componente e divisão da equipe de desenvolvimento em cada camada.

4.4 MODELO DE MATURIDADE DE RICHARDSON

É uma abordagem proposta por Leonard Richardson que classifica o padrão API RESTful em quatro níveis de maturidade, analisando a eficiência e evolução da API no uso de seus recursos e sua estrutura (JUNIOR. 2023). Os níveis de maturidade estão listados a seguir:

4.4.1 Nível 0 - POX (Plain Old XML)

O nível 0, é o mais básico, não é considerado dentro do padrão REST, não possuindo padrão de identificação de recursos através de URLs. A falta de regras e padrões na comunicação e implementação dos métodos e protocolos HTTP reduz a eficiência da estrutura do *software*, e o torna de difícil compreensão (JUNIOR. 2023).

4.4.2 Nível 1 - Recursos (Resources)

Este nível traz a utilização de recursos do HTTP (GET, POST, PUT, DELETE) para estruturar a API e realizar operações. Os recursos são acessados por uma URL única, e cada recurso é identificado por substantivos no plural, utilizando verbos para diferenciar as operações (JUNIOR. 2023).

4.4.3 Nível 2 - Verbos HTTP

Neste nível, cada recurso é acessado por uma URL específica, e os métodos HTTP apropriados são utilizados para executar cada operação. Porém, ainda não é capaz de utilizar livremente todos os recursos do HTTP (JUNIOR. 2023).

4.4.4 Nível 3 - HATEOAS (Hypermedia as the Engine of Application State)

O nível 3 possui a maturidade mais avançada, porém menos utilizado. Introduz o recurso HTTP *hypermedia* (hipermídia), que orienta aos clientes a estrutura de navegação dos demais recursos. Assim, o consumidor da API, ao acessar a rota principal, terá acesso a *links* de navegação para as demais operações executadas (JUNIOR. 2023).

5. INTEGRAÇÃO DOS COMPONENTES

5.1 CONSUMO DE API EM DART/FLUTTER

A linguagem de programação Dart, faz uso dos protocolos HTTP através de uma biblioteca padrão '**dart:io**' a qual fornece as ferramentas necessárias para acessar e utilizar os métodos. Ela realiza a interação entre o sistema operacional e os protocolos de entrada e saída de dados.

A biblioteca traz diversas funcionalidades. Entre elas estão classes para criação e manipulação de diretórios e arquivos, objetos de entrada padrão (**stdin**), saída padrão (**stdout**) e erro (**stderr**), suporte à comunicação e operações de rede entre aplicativos, com a criação de clientes e servidores TCP/UDP e APIs, que permitem a interação com os processos de entrada e saída do sistema operacional, entre muitas outras funções.

A biblioteca faz uso dos pacotes externos '**http**' e '**dio**'. O pacote '**http**' permite a realização de requisições dos tipos GET, POST, PUT e DELETE, entre outras. O resultado da requisição enviada é encapsulado em um objeto '**response**'. Na Tabela 1 estão descritos os tipos de requisição mencionados:

Tabela 1 - Descrição das requisições HTTP

Requisição	Descrição
GET	Realiza a solicitação dos dados de um recurso do servidor.
POST	Envia dados a um servidor para criação de um novo recurso.
PUT	Atualiza um recurso existente.
DELETE	Exclui um recurso.

O pacote '**dio**' é uma alternativa muito popular, que mantém todos os métodos do pacote '**http**', e adiciona funcionalidades adicionais. Uma dessas funcionalidades são os *interceptors* (interceptadores), que permitem o acesso e modificação das respostas às requisições antes que elas sejam recebidas e processadas, permitindo entre outras coisas o tratamento de erros e adição de cabeçalhos. Permite também o cancelamento de requisições,

upload de dados e arquivos nas requisições do tipo POST e gerenciamento de cookies nas requisições.

Os pacotes mencionados são frequentemente utilizados em conjunto com o Flutter, devido a flexibilidade e facilidade de uso. No Flutter, sendo um tipo de operação assíncrona, o consumo de APIs é realizado através de *widgets* como **'FutureBuilder'** e **'StreamBuilder'** (FLUTTER. 2023).

A função **'fetchData'** faz uma requisição HTTP à uma API, que então tem seu estado monitorado por um dos *widgets* mencionados e constrói a interface do usuário de acordo com o resultado da requisição (FLUTTER. 2023). O uso apropriado dos recursos mencionados torna a aplicação eficiente e responsiva na requisição e busca de dados da API.

Ao retornar os resultados de uma requisição, a maioria das APIs o fazem no formato JSON, que são convertidas e traduzidas por ambos Dart e Flutter através da biblioteca **'dart:convert'** (DART. 2023).

5.2 ARQUITETURA MVC EM DART/FLUTTER

Para garantir a eficiência do código, é importante incorporar os métodos de consumo de API a um modelo de arquitetura eficiente, integrando os dados recebidos com a interface de usuário de maneira consistente. Este trabalho explora o modelo de arquitetura MVC.

O modelo MVC divide a arquitetura do *software* em três camadas, logo, as funções necessárias para integração da API ao sistema podem ser divididas de acordo com as funcionalidades de cada camada.

A camada *Model*, a qual representa a lógica do negócio e os dados, deve incluir as classes que definem a estrutura de dados no consumo da API, implementando métodos que realizem requisições HTTP e processem os resultados. A camada *View*, responsável por apresentar os dados ao usuário, deve fazer uso dos *widgets* disponíveis no Flutter para exibir os dados e resultados recuperados da API. Já a camada *controller*, que coordena a interação entre as outras duas camadas e gerencia as requisições, deve incorporar bibliotecas **'http'** e **'dio'** para implementar e aplicar solicitações HTTP, gerenciando e realizando as chamadas de métodos apropriadas nas demais classes conforme a necessidade das requisições do usuário.

A implementação do modelo MVC junto ao Flutter traz uma integração compatível, robusta e bem estruturada, que melhora a qualidade do código e a *User Experience* (UX),

trazendo uma visualização mais dinâmica da interface de usuário, precisão e rapidez na atualização dos dados. Assim, o aplicativo se torna mais modular e aumenta a escalabilidade, reusabilidade e eficiência do código. Assim, o código se torna mais simples, flexível e de fácil entendimento e manutenção.

6. ESTUDO DE CASO: APLICATIVO GESTOQUE

6.1 ANÁLISE DA APLICAÇÃO

O gEstoque é um sistema de gerenciamento de estoque desenvolvido para a empresa Soldamaq na linguagem de programação Dart, e foi selecionado para estudo de caso de uso neste trabalho. Trata-se de uma aplicação consumidora que integra-se ao sistema ERP Sankhya

O aplicativo faz uso do *framework* Flutter, utiliza widgets para componentizar alguns cards como os produtos, o carrinho, indicadores, entre outros. A *framework* implementa o Hot Reload, em conjunto com os modos de compilação JIT e AOT. Assim, em conjunto com ferramentas de desempenho e renderização, houve uma melhora no desenvolvimento, evitando erros de runtime. Ao utilizar o *framework* Flutter, o arquivo de configurações ‘**pubspec.yaml**’ foi utilizado para armazenar os dados e configurações do projeto (MEDIUM. 2023). Entre eles estão as dependências, nome, versão, descrição, ambiente e *assets* (imagens e fontes).

O *software* tem sua estrutura definida pelo modelo *Model-View-Controller*, e divide as funções principais da aplicação entre as três camadas. A camada *View* armazena os layouts de telas, incluindo os *widgets* em um diretório de widgets, a camada *Model* contém o mapeamento de JavaScript para retorno da API dos Sistema ERP Sankhya, e a camada *Controller* gerencia a interface do usuário, cores e configurações URL. Uma camada extra denominada ‘*repository*’ foi criada para armazenamento dos padrões.

O gEstoque também faz uso da ferramenta Gradle, uma ferramenta de automação de construção (build automation), utilizada para compilação e construção de projetos de *software*. Com um *script* flexível e baseado nas linguagens Groovy e Kotlin, o Gradle é amplamente utilizado em projetos desenvolvidos em Java, bem como outras linguagens de estrutura similar, como o Dart, linguagem de programação utilizada no desenvolvimento do aplicativo.

As aplicações desenvolvidas em Dart, principalmente quando o *framework* Flutter é utilizado, o arquivo de configurações denominado “pubspec.yaml” é essencial. Estão contidos neste arquivo: configurações do projeto, como as dependências do projeto, nome do projeto, versão, descrição, ambiente, dependências para o desenvolvimento e os assets do projeto. Assets do projeto são as imagens e fontes, onde elas estarão localizadas.

6.2 APLICAÇÃO DO MODELO DE RICHARDSON

O gEstoque é uma aplicação consumidora, que integra-se a um sistema ERP externo. Desta maneira, embora incorpore alguns dos princípios do padrão REST, utilizando a identificação de recursos do sistema através de URLs e incorporando os protocolos HTTP (GET/POST), não se enquadra dentro dos padrões analisados pelo Modelo de Maturidade de Richardson. Assim, não é possível propor melhorias no desenvolvimento que sigam esses padrões e melhorem a maturidade do *software*, estando restritos à codificação do sistema ERP.

6.3 PORTABILIDADE PARA IOS

O gEstoque é um aplicativo desenvolvido exclusivamente para Android. Entretanto, como mencionado anteriormente, a linguagem de programação Dart e o *framework* Flutter possuem suporte para os dois tipos de sistema operacional, Android e iOS, com ambos possuindo recursos e funcionalidades que permitem e facilitam a integração entre as diferentes plataformas. Assim, verifica-se que a aplicação está em plena capacidade de operar em sistemas iOS.

O *port* da aplicação para plataformas iOS, devido às exigências e restrições nas políticas da Apple exigiria a utilização de um computador Mac para realizar a compilação e publicação na App Store, além da taxa anual de inscrição de desenvolvedores da Apple.

Assim, o aplicativo em si já está em condições técnicas para ser implementado em sistemas iOS, exigindo apenas a disponibilidade tecnológica e financeira da empresa em obter um computador Mac e pagar a taxa de inscrição conforme as requisições da Apple.

7. CONCLUSÃO

Na confecção deste trabalho, a maior dificuldade foi a familiarização com os detalhes das ferramentas estudadas: a linguagem de programação Dart, o *framework* Flutter, o padrão de arquitetura MVC e o Modelo de Maturidade de Richardson. Assim, o projeto exigiu o estudo detalhado e aprofundado dos componentes citados, de modo a realizar uma análise coesa e sucinta que alcançasse os objetivos definidos.

Neste trabalho foram explorados os aspectos do desenvolvimento em aplicações móveis utilizando as tecnologias Dart e Flutter, integradas com o modelo de arquitetura MVC e como aplicar o Modelo de Richardson para aumentar a qualidade, organização e maturidade do código. Foi feita uma análise e exposição da melhor maneira de integrar e incorporar essas ferramentas e implementar o consumo de APIs. Foi destacado a eficiência e a estrutura do código e as vantagens na construção do *software* seguindo os padrões estudados.

Também foi avaliado a viabilidade da implementação do Modelo de Maturidade de Richardson em um cenário real, utilizando como estudo de caso o já implementado aplicativo gEstoque. O aplicativo faz uso das ferramentas Dart, Flutter e MVC, porém, sendo uma aplicação consumidora ligada a um sistema ERP externo, o Modelo de Richardson não é aplicável.

Além disso, foi verificado que o aplicativo analisado, implementado em Dart e Flutter, já cumpre todos os requisitos técnicos para sua portabilidade em sistemas iOS, dependendo apenas da disponibilidade de um computador Mac e do pagamento da taxa de inscrição do programa de desenvolvedores da Apple para disponibilizar o sistema em plataformas iOS.

Analisando as informações expostas, conclui-se que o uso de Dart, Flutter, incorporados à Arquitetura MVC e construídos de acordo com o Modelo de Maturidade de Richardson, através do padrão RESTful é uma forma eficiente, simples e organizada de desenvolvimento de *software*. Desta maneira, os sistemas implementados dessa maneira possuem vantagens no desempenho, flexibilidade, modularidade, escalabilidade, facilidade de manutenção e entendimento do código.

7.1 TRABALHOS FUTUROS

Este trabalho avaliou a viabilidade de analisar a aplicação do Modelo de Maturidade de Richardson no aplicativo gEstoque, e foi concluído que não é aplicável, pois as funções

avaliadas pelo modelo estão incluídas diretamente no sistema ERP Sankhya ao qual o aplicativo está integrado, e cujo acesso e análise é restrito. Em trabalhos futuros, com a devida permissão e interesse da empresa Soldamaq e da responsável pelo sistema ERP, pode ser feita esta análise diretamente no sistema, emitindo um relatório dos níveis de maturidade e sugerindo mudanças relevantes que melhorem a qualidade, estrutura, organização e desempenho do código.

Também é possível, mediante interesse e viabilidade da empresa, realizar e documentar a compilação do sistema em um computador Mac para disponibilizar o mesmo na App Store para sistemas iOS, mediante pagamento da taxa de inscrição.

O trabalho analisou exclusivamente o modelo de arquitetura MVC (*Model-View-Controller*). Podem ser feitos estudos similares com outros modelos de arquitetura de software, tais como BLoC (*Business Logic Component*), Provider, MVP (*Model-View-Presenter*) entre os muitos outros padrões existentes, analisando sua viabilidade junto ao Modelo de Maturidade de Richardson.

O Flutter, sendo muito utilizado e possuindo uma comunidade ativa, está em constante evolução e adaptação. Assim novas funções e ferramentas podem ser desenvolvidas que integrem-se aos modelos estudados no trabalho e incrementem a qualidade do *software*, podendo novos estudos serem realizados para analisar a viabilidade das novas funcionalidades.

REFERÊNCIAS

AWS AMAZON. **What is Flutter?** Disponível em: <<https://aws.amazon.com/pt/what-is/flutter/>>. Acesso em: 25-07-2023.

ALURA. **Desenvolvimento de Apps Mobile: por onde começar?**. Disponível em: <<https://www.alura.com.br/artigos/desenvolvimento-apps-mobile-por-onde-comecar?>> Acesso em: 28-11-2023.

ALURA. **Flutter: aplicando integração contínua (CI)**. Disponível em: <<https://www.alura.com.br/conteudo/flutter-aplicando-integracao-continua-ci>>. Acesso

ALURA. **REST: Conceito e fundamentos**. Disponível em: <<https://www.alura.com.br/artigos/rest-conceito-e-fundamentos>>. Acesso em: 28-11-2023.

BIEFL, Matthias. **RESTful API**. API-University Press, 2016.

DART. **Documentação do Dart**. Disponível em: <<https://dart.dev/language>>. Acesso em: 29-07-2023.

DEITEL, Paul J.. **Java: Como programar? - 10ª Edição**. Pearson Education, 2017.

FLUTTER. **Documentação do Flutter**. Disponível em: <<https://api.flutter.dev/>> . Acesso em: 29-07-2023.

GAMMA, Erich et al. **Elements of Reusable Object-Oriented Software**. Design Patterns, 1995.

HOSTINGER. **O Que é uma API RESTful e Porque Isso Importa**. Disponível em: <<https://www.hostinger.com.br/tutoriais/api-restful>>. Acesso em: 28-11-2023.

JUNIOR, Rivaildo. **Modelo de maturidade de Richardson para APIs REST**. Disponível em: <<https://rivaildojunior.medium.com/modelo-de-maturidade-de-richardson-para-apis-rest-8845f93b288>>. Acesso em: 29-11-2023.

MARTIN, Robert C. **Clean code: a handbook of agile software craftsmanship**. Pearson Education, 2009.

MEDIUM. **Isolates.** Disponível em: <https://medium.com/android-dev-moz/isolates-a2a6a5db3fac>>. Acesso em: 29-11-2023.

MEDIUM. **Flutter: Don't Fear the Garbage Collector.** Disponível em: <https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30>>. Acesso em: 29-11-2023.

MEDIUM. **Flutter: Don't Fear the Garbage Collector.** Disponível em: <https://xeladu.medium.com/the-flutter-pubspec-yaml-in-detail-eee5729d9df7>>. Acesso em: 29-11-2023.

NOVAGEO. **O que é Programação Orientada a Objetos (POO)?** Disponível em: https://www.novageo.pt/novageo/displayArticles?numero=38409&_que_programacao_orientada_objetos_poo>. Acesso em: 22-03-2023.

TREINAWEB. **O que é MVC (Model-View-Controller)?** Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-mvc%5C%3E>>. Acesso em: 11-08-2023.

TREINAWEB. **O que é Dart?** Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-dart>>. Acesso em: 23-07-2023.