GoIFSC - sistema de notícias para instituições educacionais

Gabriel J. S. Ramos[1], **Rodrigo A. Machado**[1], **Juliano L. Gonçalves**[1], **Vilson H. Junior**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina
88506-400 – Lages – SC – Brasil

Abstract. *The search for better communication between the institution and its students has become an increasingly necessary process, due to the growth in the number of courses and students. This work presents the development of a communication system that aims to centralize and facilitate the communication of Câmpus Lages with its students, through a set of applications. Being divided into a Web application used by teachers to send the news, a Mobile application where students received the news via notification and a set of microservices responsible for providing communication between client applications.*

Resumo. *A busca por uma melhor comunicação entre a instituição e seus estudantes tem se tornado um processo cada vez mais necessário, devido ao crescimento do número de cursos e estudantes. Este trabalho apresenta o desenvolvimento de um sistema de comunicação que visa centralizar e facilitar a comunicação do Câmpus Lages com seus estudantes, por meio de um conjunto de aplicações. Sendo dividida em uma aplicação Web utilizada pelos docentes para enviar as notícias, uma aplicação Mobile onde os estudantes receberam as notícias via notificação e um conjunto de microsserviços responsáveis por prover a comunicação entre as aplicações clientes.*

1. Introdução

A história do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina (IFSC) começou em 1909 em Florianópolis, com a criação da Escola de Aprendizes Artífices de Santa Catarina. Desde então, com muitas mudanças de nome e de identidade, a educação profissional foi se aperfeiçoando e se interiorizando até a chegada do IFSC em Lages, com a criação do Câmpus (IFSC, 2019).

O Câmpus Lages oferece cursos de qualificação profissional, idiomas, técnicos, graduações e pós-graduação nas áreas de Ambiente e Saúde, Informática, Cultura Geral e Processos Industriais. Também é polo de oferta de cursos de pós-graduação a distância. Assim, a instituição oferece cursos gratuitos e de qualidade, atendendo às demandas por formação profissional do arranjo produtivo local dos municípios da região da Amures (IFSC, 2019).

Desde a sua inauguração em 2010, o Câmpus Lages, tem presenciado o aumento do número de cursos e, consequentemente, o número de estudantes. Desde 2015 tem a segunda maior oferta de vagas dentre os 22 Câmpus do IFSC. De forma per capita, é aquele que atende a maior quantidade de estudantes em vulnerabilidade social (Paula, 2020).

Com o aumento das atividades a instituição utiliza-se de meios formais e informais como o Sistema Integrado de Gestão de Atividades (SIGAA), *e-mail*, páginas oficiais e

aplicativos de mensagens instantâneas para comunicar eventos, notícias, fatos importantes entre outros.

As formas de comunicação formais e informais citadas possuem alguns problemas:

- O SIGAA é vinculado a um *e-mail* educacional fornecido pelo Câmpus, toda as informações postadas na ferramenta são automaticamente enviadas para o *e-mail* cadastrado, mas estas informações acabam sendo perdidas pelo desuso do *e-mail* cadastrado;
- O *e-mail* fornecido pelo estudante muitas vezes é perdido ou trocado. Devido à pouca ou nenhuma utilização, pode ocorrer que as informações enviadas por esses canais não sejam visualizadas ou são recebidas com atraso;
- O mesmo problema ocorre quando feito o contato via telefone fornecido pelo estudante, o número de telefone pode ter sido trocado ou perdido
- Em razão da quantidade de canais de comunicações utilizadas pelo Câmpus, isso gera uma descentralização das informações, tanto para quem envia quanto para quem recebe.

O objetivo deste projeto é desenvolver um sistema de comunicação centralizado, utilizando-se uma arquitetura de microsserviços, juntamente com aplicações clientes que permitam maior facilidade na comunicação entre o Câmpus Lages e seus estudantes.

Para alcançar este objetivo, foram traçados os seguintes objetivos específicos:

- Definir uma arquitetura de microsserviços para o gerenciamento e envio das informações;
- Implementar uma aplicação *Web* responsável pelo envio das informações;
- Implementar uma aplicação *Mobile* responsável por informar os estudantes sobre assuntos relevantes à sua escolha;
- Integrar tecnologias *Mobile* e *Web* para suportar requisitos, como interoperabilidade e acessibilidade.

Sob o aspecto metodológico, este trabalho será dividido em cinco etapas macro. A primeira etapa corresponde à identificação e conhecimento dos meios de comunicação usados pela instituição, a definição dos conceitos, seleção de assuntos e escrita do referencial teórico tendo como base pesquisa sobre trabalhos relacionados. A segunda etapa será a construção do modelo relacional da arquitetura de microsserviços e a implementação do microsserviço de notícias, utilizando-se o *Diagram.net* para criação dos modelos e o *Framework NestJS* para criação do microsserviço. A terceira etapa será a implementação do microsserviço de *Gateway* e de envio de notificações, utilizando-se também o *Framework NestJS*. A quarta etapa será a prototipação das interfaces clientes e a implementação da aplicação *Web*, utilizando-se o *Framer* para prototipação da interfaces clientes e o *VueJS* para criação da aplicação *Web*. A última etapa será a implementação da aplicação *Mobile* utilizando-se o *Toolkit Flutter* para criação da aplicação *Cross Platform Mobile*.

Em relação à classificação metodológica, este trabalho, sob o ponto de vista da sua natureza, é aplicado uma vez que trata-se da construção de um sistema informativo centralizado. Do ponto de vista da forma de abordagem, caracteriza-se uma pesquisa qualitativa, pois visa melhorar e centralizar a comunicação da instituição com os estudantes. Do ponto de vista de seus objetivos, define-se como um trabalho exploratório, já que o tema

de pesquisa proporciona uma maior proximidade com o problema de comunicação da instituição. Do ponto de vista dos procedimentos técnicos, é do tipo pesquisa-ação, pois o desenvolvimento do sistema informativo irá centralizar as informações da instituição, tornando-se mais um meio de comunicação.

Além da primeira seção, que contém uma introdução ao tema e aos objetivos, o restante do trabalho foi organizado da seguinte forma. A seção dois apresenta o referencial teórico, ou seja, descreve os assuntos necessários para o entendimento do trabalho como um todo. A seção três registra em detalhes como será realizado o desenvolvimento do projeto. A seção quatro apresenta o resultado do projeto e a seção cinco finaliza o trabalho com as considerações finais.

2. Referencial Teórico

Esta seção apresenta conceitos e características importantes sobre o conteúdo a ser abordado, que servirá de base para o desenvolvimento do trabalho. O mesmo encontrasse dividido em três subseções. A primeira subseção aborda sobre a Tecnologia da Informação e Comunicação. A segunda subseção apresenta a Arquitetura de Microsserviços e seu Ecossistema. A terceira subseção apresenta as tecnologias utilizadas no desenvolvimento das aplicações.

2.1. Tecnologia da Informação e Comunicação

Miranda (2016) apresenta que o termo Tecnologias da Informação e Comunicação (TIC) refere-se à conjugação da tecnologia computacional ou informática com a tecnologia das telecomunicações e tem na *Internet* e mais particularmente na *World Wide Web* (WWW) a sua mais forte expressão. Quando estás tecnologias são usadas para fins educativos, nomeadamente para apoiar e melhorar a aprendizagem dos estudantes e desenvolver ambientes de aprendizagem, podemos considerar as TIC como um subdomínio da Tecnologia Educativa.

As TIC's são consideradas como sinônimo das Tecnologias da Informação (TI). Contudo, é um termo geral que frisa o papel da comunicação na moderna tecnologia da informação. Entende-se que TIC consistem de todos os meios técnicos usados para tratar a informação e auxiliar na comunicação. Em outras palavras, TIC consistem em TI bem como quaisquer formas de transmissão de informações e correspondem a todas as tecnologias que interferem e mediam os processos informacionais e comunicativos dos seres. Ainda, podem ser entendidas como um conjunto de recursos tecnológicos integrados entre si, que proporcionam por meio das funções de software e telecomunicações, a automação e comunicação dos processos de negócios, da pesquisa científica e de ensino e aprendizagem de Oliveira (2015).

As TIC's enriqueceram e aprimoraram os meios de comunicação, dentre elas, o correio eletrônico (*E-mail*) redes *extranet*, *intranet* e a videoconferência, diretamente ligadas sobre todas as formas, na vida da sociedade, das organizações e instituições. Esses avanços tecnológicos potencializaram as formas de comunicação, estudo e compartilhamento dos conhecimentos, experiências, tornando visível a ciência (Robbins, 2005).

2.1.1. Tecnologia da Informação e Comunicação nas Instituições

Para as instituições torna-se relevante comentar, sobre outros benefícios advindos das tecnologias, como por exemplo o ensino a distância, as pesquisas científicas com conteúdo completo disponível e de forma instantânea, os formulários eletrônicos para solicitação de serviços internos ou externos com maior rapidez e eficácia, tornando a comunicação sem fronteiras, bem como, os serviços de perguntas e respostas *Frequently Asked Questions* (FAQ), reduzindo o tempo do cliente ou usuário nas organizações.

As mensagens por meio eletrônico, ou seja, os *E-mail* revolucionaram as formas de comunicação para as organizações e instituições, possibilitando que a mensagem chegue a todos os envolvidos ao mesmo tempo, sem desperdício de papel, por um meio rápido e eficiente.

Ressalta-se outra forma de comunicação neste mesmo patamar que são as mensagens instantâneas através dos aparelhos eletrônicos, *SmartPhones*, *Tablets*, entre outros. Que de acordo com o estudo de Robbins (2005), o *Short Message Service* (SMS) e mensagens de texto são formas rápidas e baratas para contatos com os funcionários das organizações e instituições. Outros meios conhecidos são as redes sociais que também contribuem para a comunicação entre os funcionários, as organizações e toda a sociedade (Robbins et al., 2011).

2.2. Arquitetura de Microsserviços

O setor de tecnologia vem testemunhando uma rápida mudança na arquitetura aplicada e prática de sistemas distribuídos, o que tem levado as gigantes da indústria (como *Netflix, Twitter, Amazon, eBay e Uber*) a abandonarem a construção de aplicações monolíticas e adotar a arquitetura de microsserviços. Embora os conceitos fundamentais por trás dos microsserviços não sejam novos, a aplicação contemporânea da arquitetura de microsserviços é atual, e sua adoção tem sido motivada em partes pelos desafios de escalabilidade, falta de eficiência, velocidade lenta de desenvolvimento e dificuldades em adotar novas tecnologias que surgem quando sistemas complexos de software estão contidos em e são implementados como uma grande aplicação monolítica Fowler (2017).

Adotar arquitetura de microsserviços, seja a partir do zero ou dividindo uma aplicação monolítica existente em microsserviços desenvolvidos e implantados independentemente, resolve esses problemas. Com a arquitetura de microsserviços, uma aplicação pode ser facilmente escalada tanto horizontalmente quanto verticalmente, a produtividade e a velocidade do desenvolvedor aumentam dramaticamente e tecnologias antigas podem facilmente ser tocadas pelas mais recentes (Fowler, 2017). A arquitetura de microsserviços proposta permitirá a integração de novas funcionalidades no formato *Plug in Play* (Liga/Desliga), provisionando assim uma arquitetura modular.

A arquitetura de microsserviços é uma abordagem para o desenvolvimento de uma única aplicação formada por um conjunto de serviços em um determinado domínio limitado (microsserviços), cada um rodando em seu próprio processo distribuído e se comunicando através de trocas de mensagens.

Em sistemas distribuídos existe a troca de mensagens entre várias aplicações, podendo gerar uma sobrecarga nos canais de comunicação entre as mesmas (Hohpe e Woolf,

2004). Um exemplo de rede interconectada pode ser visto na figura 1 onde os retângulos azuis representam aplicações e as linhas sua devidas conexões.

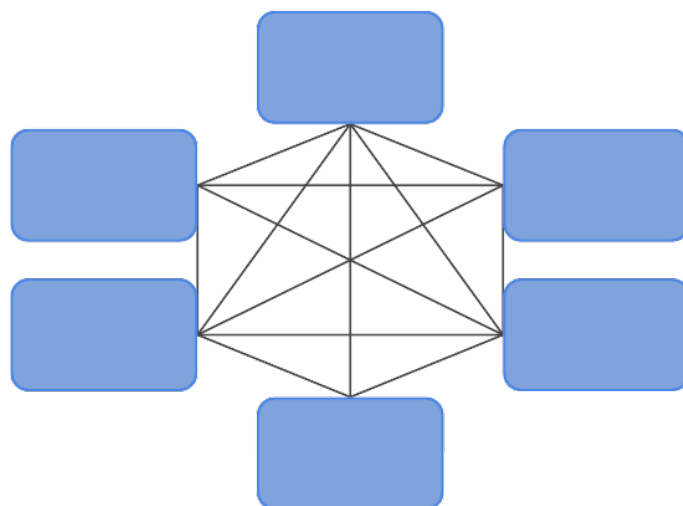

Figura 1. Integração múltipla ponto a ponto adaptado de (Hohpe e Woolf, 2004)

Para resolver este problema foi utilizado o *Message Broker*. Pode-se definir um *Message Broker* como uma aplicação responsável por receber mensagens de múltiplos remetentes, determinar o destino correto e enviar a mensagem ao canal de comunicação com o destino correto (Hohpe e Woolf, 2004). Na figura 2 está ilustrada a utilização de um *Message Broker* representado pelo retângulo central verde.

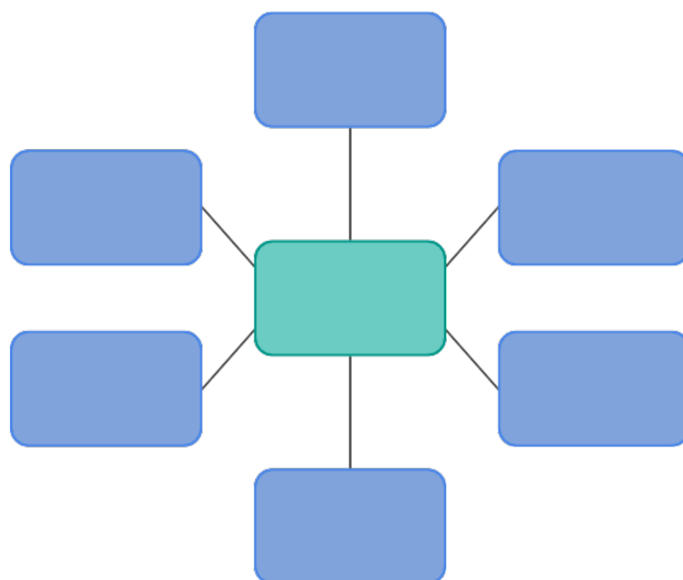

Figura 2. Integração com *Message Broker* adaptado de (Hohpe e Woolf, 2004)

Uma arquitetura compreende-se por um conjunto de classes agrupadas, originando componentes ou pacotes com funcionalidades definidas em um sistema, ressaltando a importância da sua escolha para uma determinada necessidade (Tolfo et al., 2011). Shaw e Garlan (1996) apresenta que uma arquitetura de software é responsável por definir a

organização do sistema e manter os relacionamentos entre os componentes. Portanto, a arquitetura comporta-se como o esqueleto do sistema, por isso é essencial todo projeto ter uma arquitetura bem implementada para que o mantenha organizado e objetivo, visando facilitar seu entendimento por qualquer *stakeholder* ao observá-lo.

Existem vários tipos de arquitetura de *sistemas*, e, dentre eles, podem ser destacadas duas:

- A arquitetura monolítica ou arquitetura de três camadas é uma das mais utilizada no desenvolvimento de softwares, onde todo o sistema é feito dentro de um único escopo, ou seja, se define por ser um grande bloco de código onde estão presentes todas as regras de negócio e funcionalidades da aplicação, podendo ser dividida em três partes: aplicações clientes, onde existe a interação com os usuários, aplicações *back-end*, onde é tratado toda a parte negocial bem como o processamento de requisições e uma base de dados. Geralmente uma aplicação que foi construída em cima da arquitetura monolítica costuma ter uma única base de dados utilizada é acessada por todo o sistema.
- Já a arquitetura de microsserviços é baseada na construção de uma aplicação dividida em vários serviços menores, bem específicos e objetivos sendo também independentes entre si, essa arquitetura torna-se mais sustentável, possibilitando que as soluções sejam reutilizáveis e aumentem a produtividade. Gamma et al. (1995) ainda sugerem que o projeto deve ser apropriado para o problema que visa solucionar, contudo, deve ser generalizável para resolver futuras especificações, minimizando a quantidade de re-projetos.

A arquitetura de microsserviços pode ser aplicada em diferentes contextos e aplicações, pois se trata de um padrão de desenvolvimento de software, que implica no desenvolvimento de um conjunto de serviços web com funções bem definidas que trabalham de forma colaborativa. Este tipo de arquitetura possibilita o desenvolvimento de sistemas mais flexíveis e escaláveis, logo o sistema possuirá uma capacidade de expansão e manutenção mais simples, tendo em vista que o nível de manutenibilidade, ou seja, o grau de facilidade de ser mantido, estará diretamente relacionado a cada um dos serviços.

2.2.1. Ecossistema microsserviços

Os microsserviços não vivem isolados. O ambiente no qual os microsserviços são construídos, executados é interagem é onde eles vivem. As complexidades do ambiente de microsserviços de grande escala são comparáveis às complexidades ecológicas de uma floresta tropical, um deserto ou um oceano, e considerar este ambiente como um ecossistema um ecossistema de microsserviços é benéfico quando se adota a arquitetura de microsserviços (Fowler, 2017). Em ecossistemas de microsserviços bem projetados e sustentáveis, os microsserviços são separados de toda a infraestrutura. Eles são separados dos *hardwares*, são separados das redes, separados dos *pipelines* de *build* e de *deployment*, separados das descobertas de serviço (*service discovery*) e do balanceamento de carga. Isso tudo faz parte da infraestrutura do ecossistema de microsserviços, e construir, padronizar e manter essa infraestrutura de maneira estável, escalável, tolerante a falhas e confiável são fatores essenciais para uma operação bem sucedida dos microsserviços.

2.2.2. *Gateway*

Uma API *Gateway* consiste em um único ponto de entrada (figura 3) para a aplicação, servindo como interface única de API para receber as requisições dos clientes e encaminhar as mesmas para o seu serviço correspondente (Neto Moreira et al., 2021). As requisições podem ser agregadas a fim de fornecer um serviço único composto por vários Microsserviços.

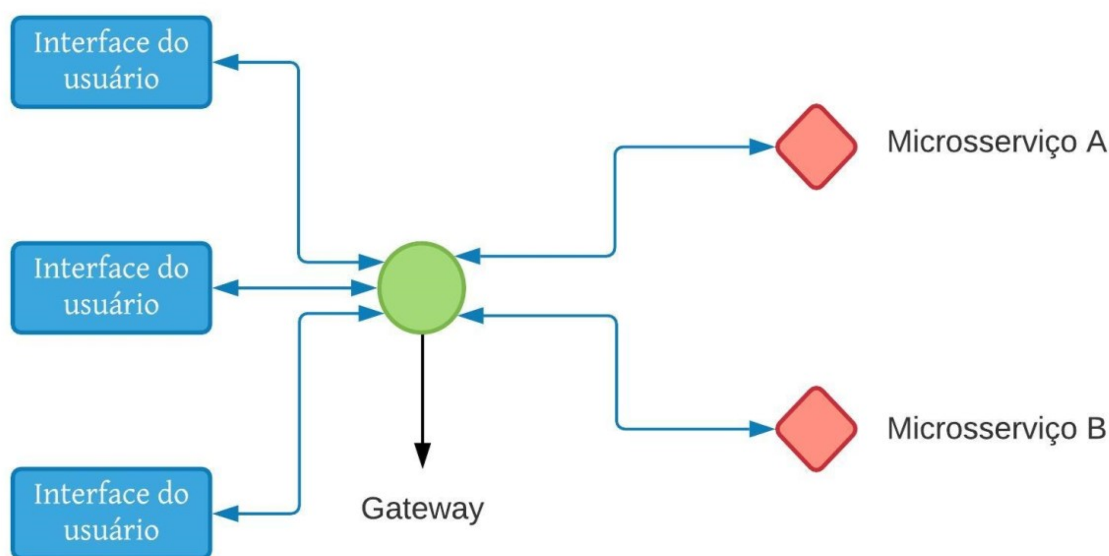

Figura 3. Exemplo de *Gateway* (Neto Moreira et al., 2021)

2.2.3. *Padrão REST*

A Transferência de Estado Representacional ou *Representational State Transfer* (REST) é um modelo de arquitetura desenvolvido no ano de 2000 por Roy Fielding, um dos autores do protocolo mais usado no mundo, o *Hypertext Transfer Protocol* (HTTP) (Fielding, 2000). Trata-se de uma coleção de princípios e restrições arquitetônicas para o desenvolvimento de aplicações distribuídas na *Web*. Ele adota o paradigma cliente servidor, onde as requisições partem inicialmente do cliente e são respondidas pelo servidor (Fielding, 2000). A formalização de um conjunto de melhores práticas denominadas restrições foi o intuito geral de REST. As restrições tinham como objetivo determinar a forma na qual padrões como *Hypertext Transfer Protocol* (HTTP) e *Uniform Resource Identifier* (URI) deveriam ser modelados, aproveitando de fato todos os recursos oferecidos pelos mesmos. O REST foi gerado numa perspectiva conhecida como *null style*. Basicamente, representa um conjunto simples e vazio de características. Os *web services* desenvolvidos com o padrão arquitetural REST são chamados de *Web* API ou simplesmente API REST. Se o *Web service* segue todos os princípios e restrições em sua implementação, este é considerado RESTful (Salvadori et al., 2015).

2.3. Tecnologias

Nesta subseção serão apresentadas as tecnologias que serão utilizadas para a implementação deste trabalho. As tecnologias descritas foram escolhidas com base no conhecimento dos autores nas mesmas e sua ampla documentação na *internet*.

2.3.1. *RabbitMQ*

RabbitMQ é definido como um *Message Broker* e gerenciador de filas que permite o envio e o recebimento de mensagens entre aplicações, sendo uma implementação do protocolo *Advanced Message Queuing Protocol* (AMQP). O *RabbitMQ* foi desenvolvido na linguagem de programação *Erlang*, tendo seu desenvolvimento iniciado em 2006, e a primeira versão liberada foi em 2007 através de uma licença *Mozilla Public License* (MPL) RabbitMQ (2021).

O *RabbitMQ* suporta o envio de mensagens através de fila e através de esquema de publicação e inscrição, sendo que o mesmo é composto basicamente por um servidor, bibliotecas cliente para a comunicação com o *Broker* que estão disponíveis em diversas tecnologias, incluindo *Go, Nodejs e Dart*, é uma plataforma para *plug-ins* adicionais para complementar a utilização do *RabbitMQ* através de recursos adicionais. Estes *plugins* são criados pela comunidade e são apenas publicados no site do RabbitMQ (RabbitMQ, 2021).

2.3.2. *Json Web Token*

Popularmente conhecido como *JSON Web Token* (JWT) é um padrão aberto de *tokens* de acesso para representação segura de afirmações *claims*. Por utilizar técnicas de criptografia assimétrica, é possível armazenar diversas informações dentro de um *token*, como por exemplo a data de expedição, com a segurança de que não poderão ser adulteradas.

2.3.3. *PostgreSQL*

O *PostgreSQL* é um sistema de gerenciamento de banco de dados objeto-relacional, desenvolvido pelo Departamento de Ciência da Computação da Universidade da Califórnia em *Berkeley*. Sendo totalmente compatível com, Atomicidade, Consistência, Isolamento e Durabilidade (ACID). Fornecendo suporte completo a chaves estrangeiras, junções, visões e gatilhos. Inclui a maior parte dos tipos de dados da ISO SQL:1999. Suporta também o armazenamento de objetos binários, incluindo figuras, sons ou vídeos (PostgreSQL, 2021).

2.3.4. *GoLang*

A linguagem de programação *Go* foi criada em setembro de 2007 por Robert Griesemer, Rob Pike e Ken Thompson, todos do *Google*, e foi lançada em novembro de 2009. Os

objetivos da linguagem e das ferramentas que acompanham eram ser expressiva, ser eficiente tanto para compilar tanto para executar e eficaz para escrever programas confiáveis e robustos.

2.3.5. *NestJS*

O *NestJS* é uma estrutura para a construção de aplicativos *Node.js* do lado do servidor eficientes e escalonáveis. Ele usa *JavaScript* progressivo, é construído e suporta totalmente *TypeScript* (ainda permite que os desenvolvedores codifiquem em *JavaScript* puro) e combina elementos de Programação Orientada a Objetos (POO), Programação Funcional (FP) e Programação Reativa Funcional (FRP) (Nestjs, 2021).

O *NestJS* faz uso de estruturas robustas de servidor HTTP como a biblioteca *Express* ou *Fastify*, fornecendo um nível de abstração acima dessas estruturas, mas também expõe sua *Application Programming Interface* (API) diretamente para o desenvolvedor, permitindo o surgimento de módulos de terceiros (Nestjs, 2021).

2.3.6. *Firebase Cloud Message*

O *Firebase* é definido como um *Backend-as-a-service* (BaaS) um serviço de computação em nuvem que fornece aos desenvolvedores uma variedade de ferramentas e serviços para ajudá-los a desenvolver aplicativos de qualidade, aumentar sua base de usuários e ser mais lucrativo (Firebase, 2021).

O *Firebase Cloud Messaging* um modulo do *Firebase* que disponibiliza o envio de notificações sem custo, ele herdou do seu antecessor *Google Cloud Messaging* (GCM) a infraestrutura confiável e escalável com isso é possível desenvolver uma aplicação com *push notification* de maneira rápida e eficiente sem ter grandes preocupações (FCM, 2021).

2.3.7. *VueJS*

Uma tradução direta, o termo *front-end* significa "parte da frente", e dentro do contexto de aplicações *Web* se refere à parte da aplicação que interage diretamente com o usuário final, ou seja, faz referência à parte visível de um site (da Fonseca e Balbino, 2019). O *front-end* é composto por todo o código que é executado no lado do cliente, dentro do navegador do usuário. Desse modo, ele vai além das telas, botões, imagens, campos de entrada e formulários, sendo composto também por validações, tratamentos de erros entre outros.

Os *Single Page Application* (SPA) são desenvolvidos para expandir o alcance por meio do navegador, melhorando a experiência do usuário (UX). A SPA é capaz de ser composta devido à nova tecnologia emergente, apelidada de *Asynchronous JavaScript* e *Extensible Markup Language* (XML) o (AJAX). A importância dos serviços da *Web* devido à tecnologia imersa é ter um impacto significativo sobre domínio diferente. O SPA é composto por página individual que pode ser atualizada de forma independente em cada ação do usuário, de modo que a página inteira não precise ser recarregada como um

aplicativo da *Web* clássico. Isso, por sua vez, ajuda a aumentar os níveis de interatividade, capacidade de resposta e satisfação do usuário (Joseph, 2015).

O *Progressive Web App* (PWA) são aplicações *Web* que podem aparecer ao usuário como aplicativos tradicionais ou aplicativos móveis no que se refere a interface. Tem como objetivo combinar recursos oferecidos pela maioria dos navegadores modernos com os benefícios de aplicações nativas (Silva e Tiosso, 2020).

Oferecendo recursos como:

- Envio de notificações aos usuários;
- Atalho na área de trabalho do dispositivo;
- Disponível independentemente de conexão com a internet;
- Acesso a alguns recursos do sistema operacional como câmera, galeria;
- Disponível independentemente de conexão com a internet;
- geolocalização e agenda de contatos.

O *Vue* surgiu com a ideia de ser um *framework* que ajudaria na prototipagem rápida, oferecendo uma maneira fácil e flexível de ligação de dados reativos e componentes reutilizáveis.

O *Vue* é um *framework* progressivo do *JavaScript* de código aberto para a construção de interfaces de usuário (Silva e Silva, 2017). A biblioteca central é focada apenas na camada de visualização e é fácil de ser integrada a outras bibliotecas ou projetos existentes. Por outro lado, o *Vue* também é perfeitamente capaz de fornecer aplicativos de página única sofisticados (Vuejs, 2021a).

Uma das principais características do *Vue* é sua técnica de observação de objetos no DOM que faz com que essas alterações sejam refletidas nas páginas *HyperText Markup Language* (HTML) conforme a Figura 8.

Na figura 4, o início das mensagens parte dos dados, notificando seus observadores que por sua vez disparam a função de renderização do componente, atualizando a árvore do DOM Virtual que atualizara a pagina *Web*. Quando ocorrer a modificação dos dados haverá a coleta como dependência dos observadores que disparará a função de renderização do componente.

Figura 4. Estrutura de comunicação reativa do *VueJS*. (Vuejs, 2021a)

2.3.8. *Flutter*

O surgimento das tecnologias *Mobile* trouxe consigo uma nova visão sobre as interações humano-computador. Com o uso dos *smartphones*, e sua portabilidade, os sistemas tornaram-se capazes de acompanhar o usuário em suas atividades rotineiras, independente de sua localização (Castro e Tedesco, 2014). Assim, a relação humano-computador passou a se dar de forma atemporal, já que não havia mais a necessidade de um local específico para interagir com determinado aplicativo.

O Flutter é um *framework* desenvolvido pela *Google* em cima da linguagem *Dart* criada e mantida pela mesma, anunciado no *Google* IO/17. Segundo a documentação oficial do *framework*, define se, "*Flutter* é o *kit* de ferramentas de *User Interface* IU portátil de código aberto do *Google*, voltado para a criação de aplicativos bonitos e compilados de forma nativa para dispositivos móveis, *web* e *desktop* a partir de um única base de código. O *Flutter* é usado por desenvolvedores e organizações em todo o mundo" (Flutter, 2021).

As aplicações desenvolvidas em *Flutter*, são divididas em duas:

- Plataforma nativa : para aplicativos direcionados a dispositivos móveis e *desktop*, o *Dart* inclui uma *VM Dart* com compilação *just-in-time* (JIT) e um compilador antecipado *ahead-of-time* (AOT) para a produção de código de máquina;
- Plataforma da *Web* para aplicativos direcionados à *eeb*, o *Dart* inclui um compilador de tempo de desenvolvimento (dartdevc) e um compilador de tempo de produção (dart2js). Ambos os compiladores traduzem o *Dart* em *JavaScript*. A figura abaixo demonstra o esquema de funcionamento onde a cor azul forte representa o processo de compilação (JIT) e o tom mais claro representa o processo de compilação (AOT).

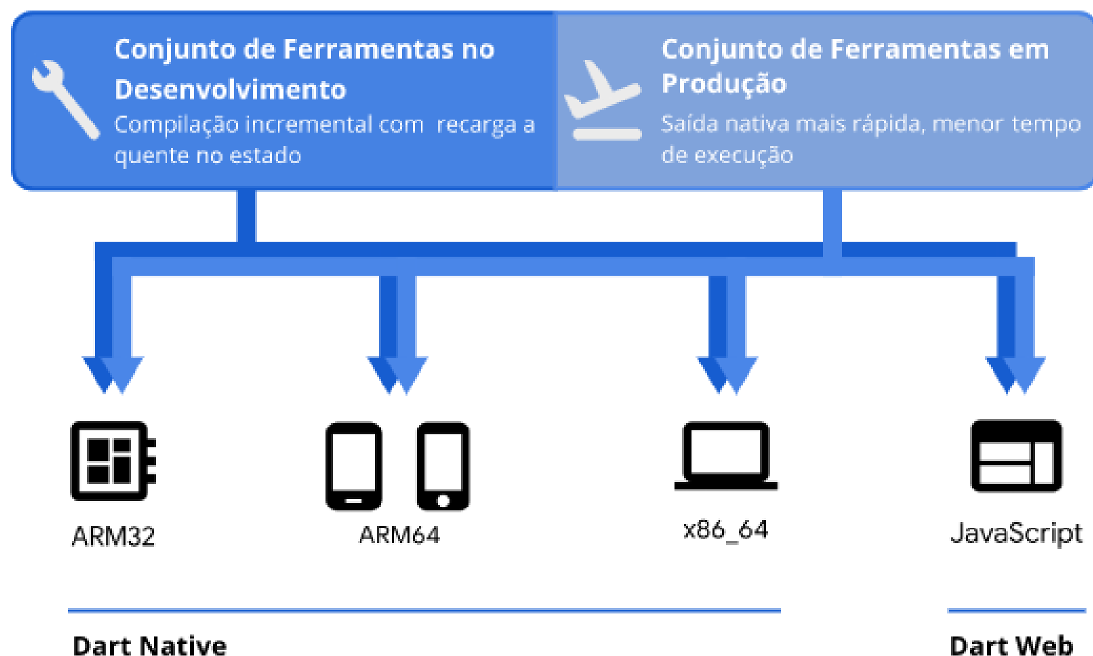

Figura 5. Compilador *Dart* adaptado de Dart (2021)

3. Modelagem do Sistema

Esta seção apresenta os quatro módulos utilizados para modelagem do sistema, sendo eles: requisitos funcionais, diagrama de atividades, modelo relacional e a prototipação das interfaces.

Inicialmente foi realizado uma entrevista com o chefe do Departamento Ensino, Pesquisa e Extensão (DEPE) e foram destacadas algumas dificuldades na comunicação do Câmpus com seus estudantes, conforme citado na entrevista em anexo. Os principais requisitos levantados na entrevista foram:

- Garantia da entrega das informações;
- Centralização das informações;
- Comunicação direta sem intermediários;

Após a realização da entrevista foram levantados os seguintes requisitos das aplicações:

- Como deveria funcionar toda a comunicação entre as aplicações.
- Como tornar a arquitetura escalável e estável.

Para o requisito da comunicação definimos a utilização da arquitetura cliente e servidor. Desta forma, o aplicativo acessaria as regras e funcionalidades referente ao estudante e o site teria acesso as regras e funcionalidades referente ao docente, normalmente a integração seria feita por um *web service*. Para que o sistema cumpra com o segundo requisito levantado, nos tivemos que elaborar uma arquitetura de microsserviços.

A arquitetura de microsserviços foi definida seguindo três pilares, *gateway*, *message broker* e os microsserviços, onde cada pilar teria sua função para que o sistema seja escalável e estável.

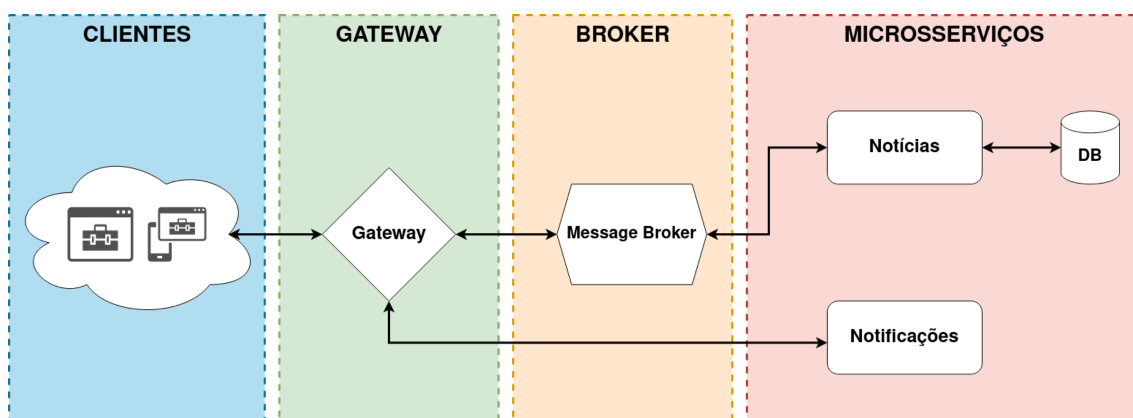

Figura 6. Arquitetura de microsserviços proposta

Os microsserviços são responsáveis por disponibilizar os serviços a serem consumidos pelas aplicações clientes, inicialmente definimos dois microsserviços, um responsável pelo gerenciamento dos usuários e noticias do sistema e outro responsável pelo envio das notificações ao aplicativo.

O *gateway* será responsável por intermediar as requisições dos clientes aos microsserviços, desta forma é possível que existam vários microsserviços distribuídos, mas apenas um ponto de entrada para a utilização dos serviços disponíveis, permitindo que seja integrado novos serviços sem grandes problemas na integração entre as aplicações clientes.

O *message broker* será responsável por gerenciar a comunicação do *gateway* com os microsserviços, definimos o uso da implementação do *RabbitMQ* para ser o responsável por gerenciar a comunicação. A fim de melhorar a compreensão da arquitetura do sistema, a figura 6 demonstra os módulos a serem criados neste trabalho.

3.1. Requisitos funcionais

O Quadro 1 apresenta os requisitos funcionais do sistema em geral, demonstra a ordem de prioridades a serem seguidas no desenvolvimento, definindo quais itens serão implementados primeiro e quais podem ser implementados posteriormente. Os requisitos foram levantados no projeto de pesquisa (Desenvolvimento de sistema sensível ao contexto para um ambiente educacional), utilizando-se de entrevistas com docentes e secretariado.

Quadro 1. Requisitos funcionais gerais

Nome Requisito	Prioridade
RF001. Gerenciar dados do usuário	Alta
RF002. Cadastrar as notícias	Alta
RF003. Envio da notificação (notícias)	Média
RF004. Sistema de tópicos	Baixo

O quadro 2 apresenta os requisitos funcionas voltado ao aplicativo, se limitando ao nível de usuário do estudante.

Quadro 2. Requisitos funcionais *Mobile*

Nome Requisito	Prioridade
RF001. Gerenciar conta do estudante	Alta
RF002. Mostrar notícias	Alta
RF003. Inscrições em tópicos	Baixo
RF004. Pesquisar por notícias	Baixa

O quadro 3 apresenta os requisitos funcionas voltado ao site, sendo limitado apenas para o nível de usuário do docente.

Quadro 3. Requisitos funcionais *Web*

Nome Requisito	Prioridade
RF001. Gerenciar conta do docente	Alta
RF002. Cadastrar as notícias	Alta
RF003. Envio da notificação (notícias)	Média
RF004. Pesquisar por notícias	Baixa

3.2. Diagramas

O sistema permite que os docente possam criar um canal de comunicação com os estudantes. A aplicação *Web* possuirá acesso apenas para os docentes, permitindo que eles enviem as notícias para os estudantes. A aplicação *Mobile* será acessado apenas pelos estudantes do Câmpus, permitindo que eles possam receber e visualizar as notícias de seu interesse.

3.2.1. Diagrama de caso de uso

Como forma de ilustrar as funcionalidades do sistema, foi criado dois diagramas de casos de uso da *Unified Modeling Language* (UML) utilizando a ferramenta *diagrams.net*, conforme é apresentado nas figuras 7 e 8.

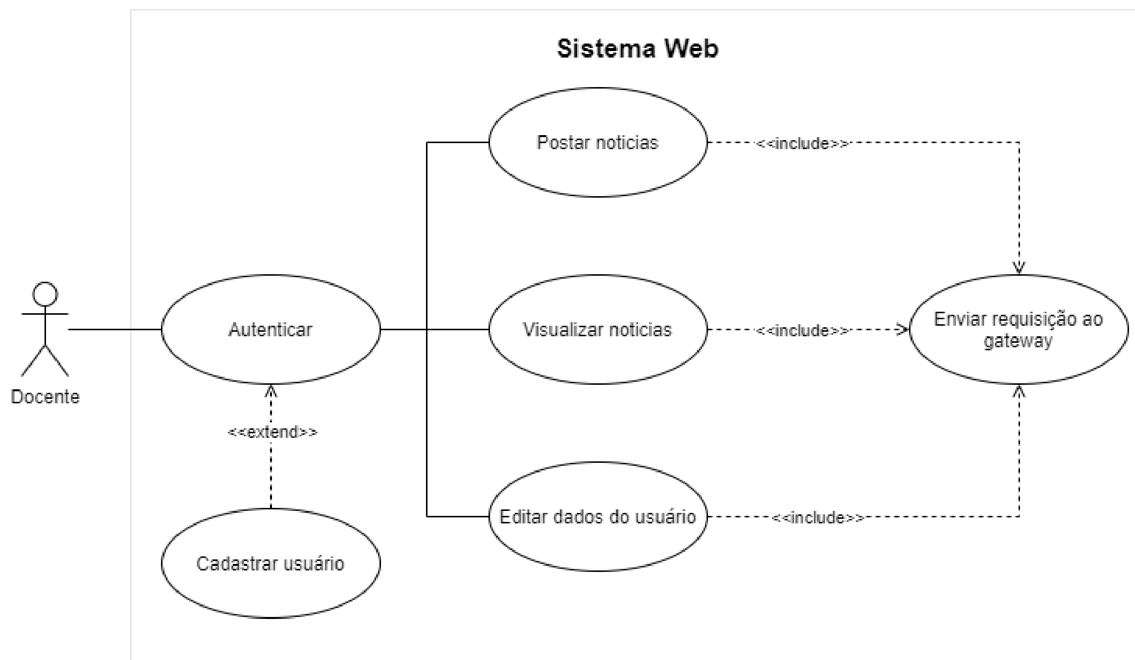

Figura 7. Diagrama de caso de uso da *Web*

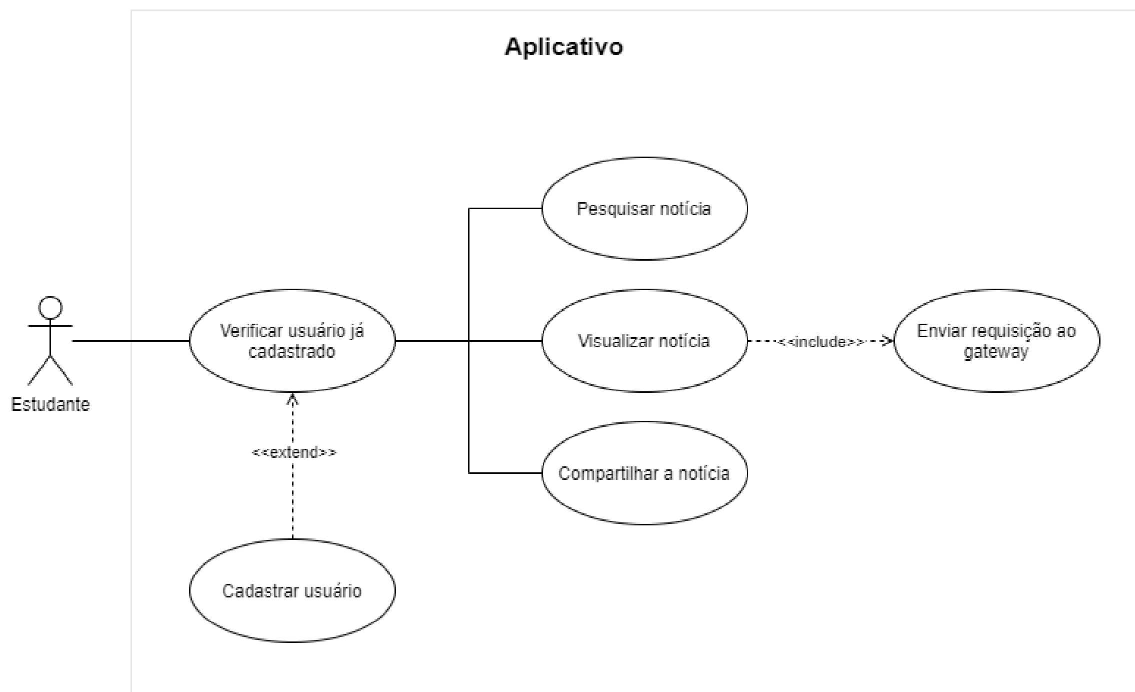

Figura 8. Diagrama de caso de uso do *Mobile*

3.2.2. Diagrama de atividades

Para um melhor entendimento das atividades do sistema foi dividido o diagrama em duas partes *web* e aplicativo, abstraindo a arquitetura de microsserviços sendo representado

como *gateway*, nas figuras 9 e 10.

O *Gateway* é a porta de entrada para o sistema, responsável por intermediar o canal de comunicação entre as aplicações *Web* e *Mobile*, permitindo acessar os microsserviços, como cadastro dos estudantes e autenticação dos docentes, cadastro das notícias, retornar as notícias e notificar os estudantes entre outros.

Os estudantes possuíram acesso ao sistema através do aplicativo *Mobile*, que irá permitir que eles possam se cadastrar e escolher tópicos que desejam receber as notícias. No cadastro, após inserir os dados do estudante, será necessário gerar um *token* de identificação, responsável por identificar o dispositivo do estudando, permitindo que seja possível o envio das notificações para o dispositivo. Após o cadastro, o estudante poderá receber as últimas notícias e notificações de novas noticias, filtradas por tópicos que o estudante esteja inscrito.

Os docentes possuíram acesso ao sistema através de uma aplicação *Web*, após a realização do cadastro e do *login*, as opções de cadastrar e visualizar as notícias estarão disponíveis, após o cadastro da notícia, automaticamente o sistema enviará a notificação aos estudantes inscritos no tópico da notícia. As figura 9 e figura 10 apresenta o diagrama de atividade referente ao canal de comunicação entre os docentes e os estudantes.

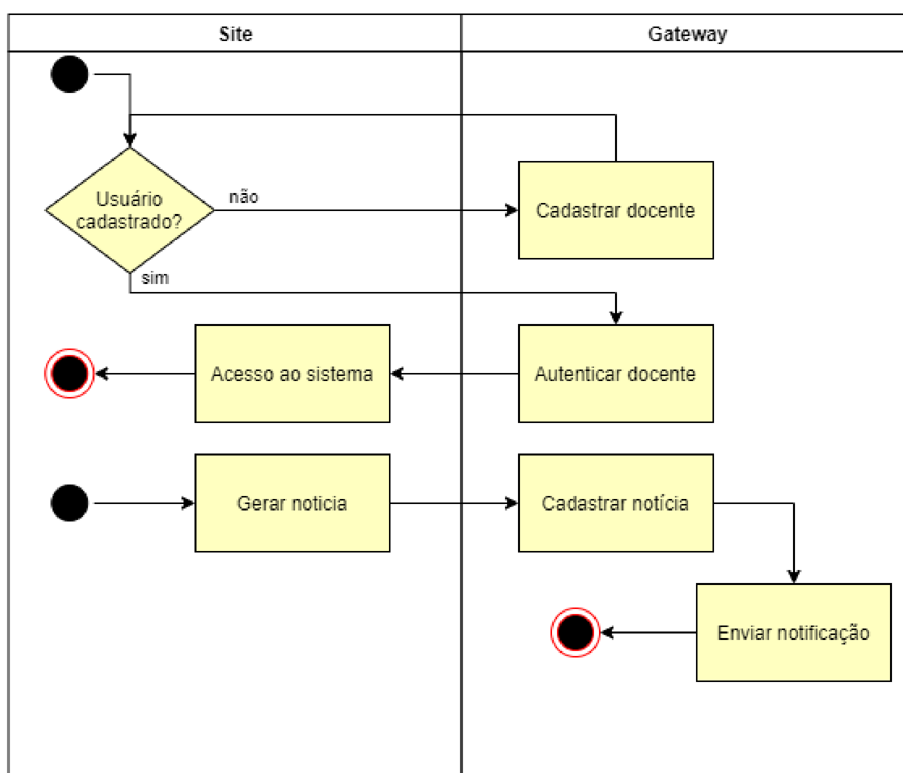

Figura 9. Diagrama de atividade da *Web*

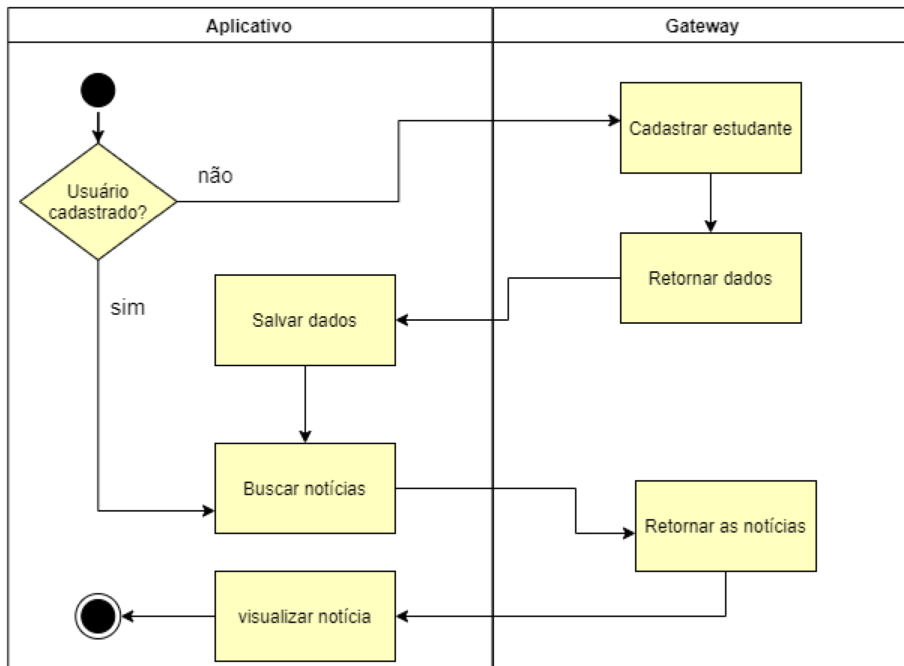

Figura 10. Diagrama de atividade do *Mobile*

3.2.3. Diagrama conceitual do banco de dados do microsserviço de notícias

Outra forma de representação do sistema, está no diagrama de conceitual, este diagrama descreve o domínio do problema proposto, apresentando as entidades e seus relacionamentos. A entidade *Teachers* se relaciona com a entidade *News* permitindo que seja gerado nenhuma ou mais notícias no sistema, sendo vinculado a um único docente, a entidade*News* possui um relacionamento com a entidade *Topics*, onde cada noticia possuirá um ou mais tópicos vinculados, e por fim a entidade *Students* possui um relacionamento com *Topics*, sendo um ou mais estudantes que poderão se inscrever em um ou mais tópicos. As figuras 11 detalha melhor o modelo conceitual.

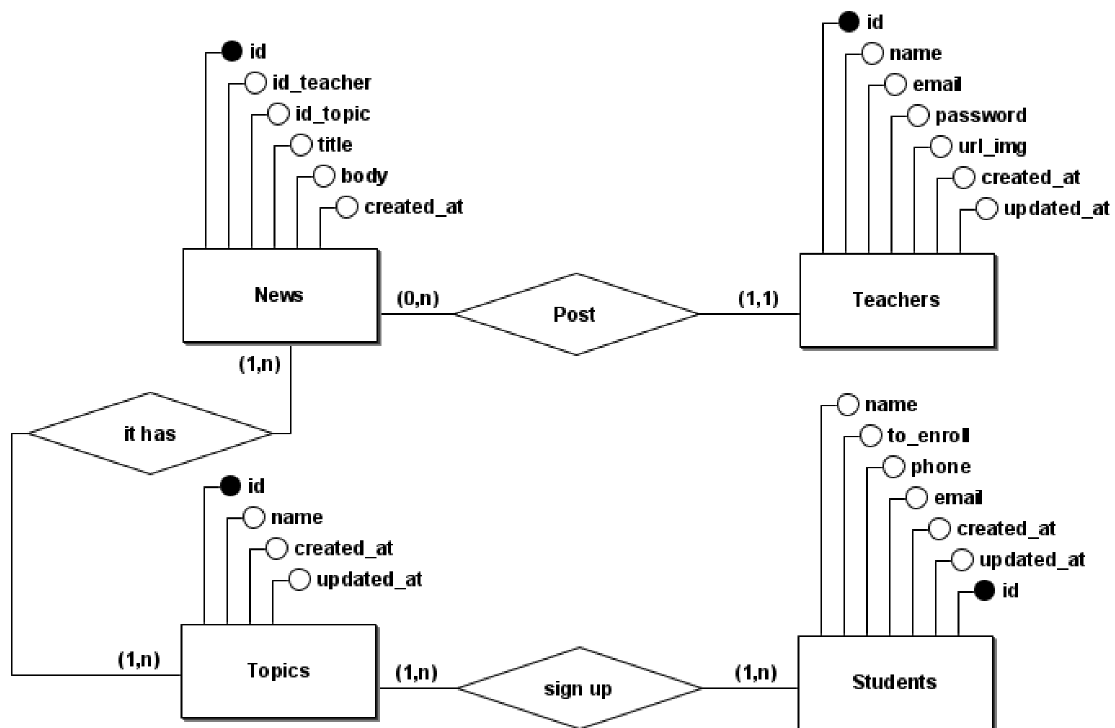

Figura 11. Modelo conceitual do banco de dados do microsserviço de notícias

3.3. Prototipação de Interfaces

A prototipação das interfaces é um passo importante para ter como base o que será desenvolvido, essa estratégia serve como guia para os desenvolvedor e até como forma de validação de fluxos de software.

Utilizamos a ferramenta *Frame* para prototipar a aplicação *Web* e *Mobile*. A figura 12 e 13 mostra algumas interfaces da aplicação *Web*.

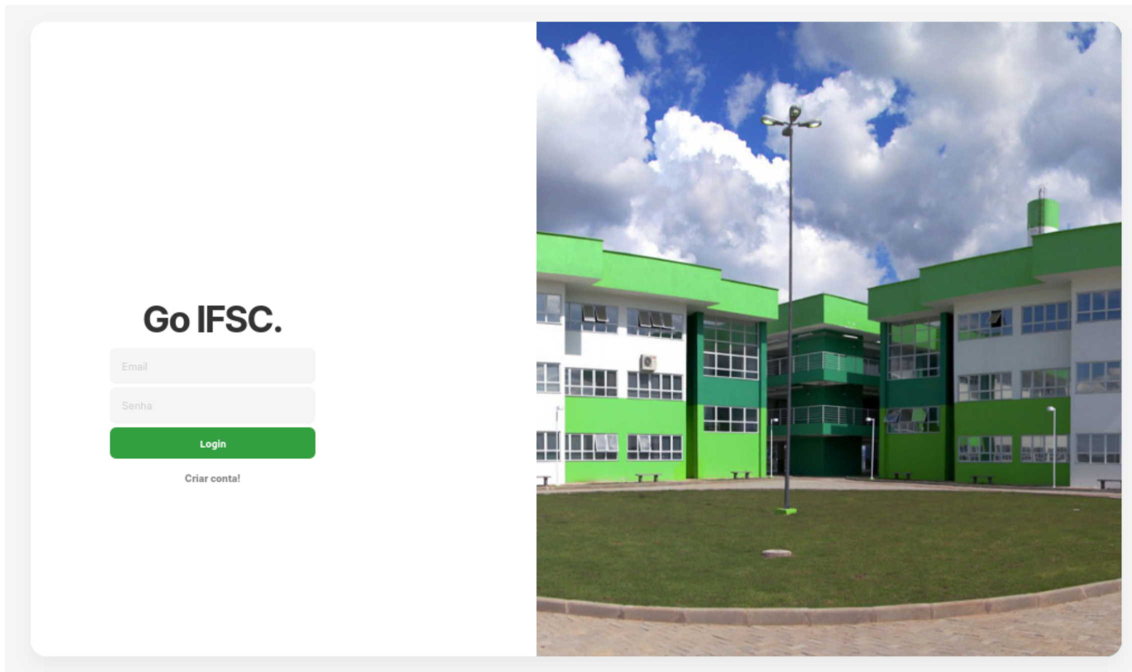

Figura 12. Tela de *Login* versão *Web*

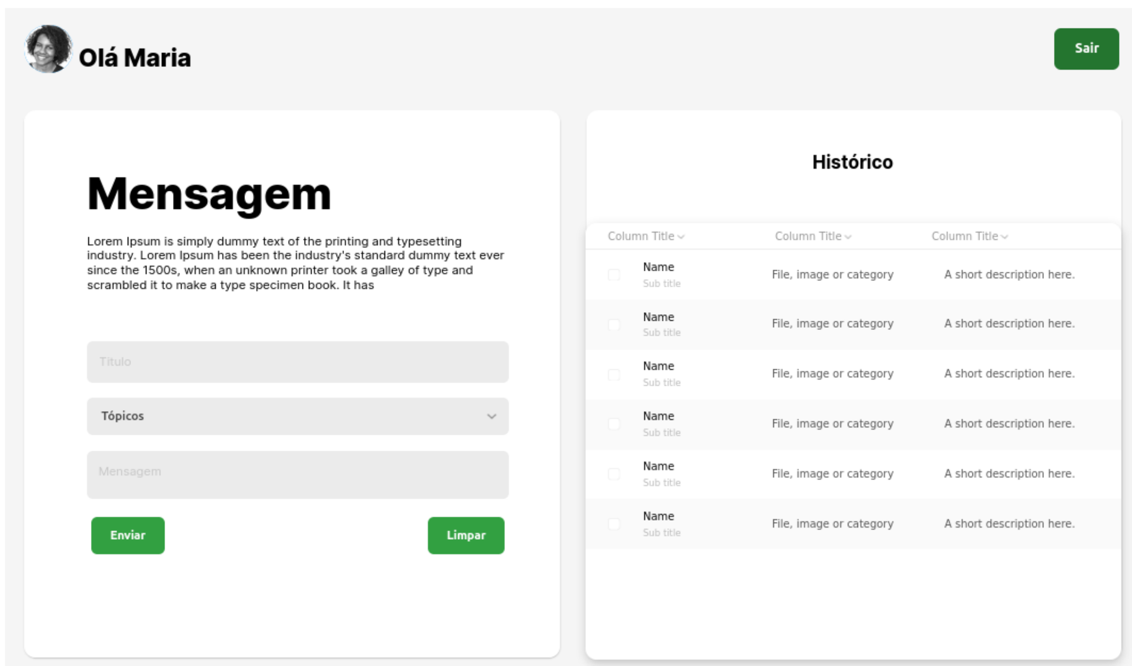

Figura 13. Tela de *Dashboard* versão *Web*

A figura 14 mostra algumas interfaces da aplicação *Mobile*.

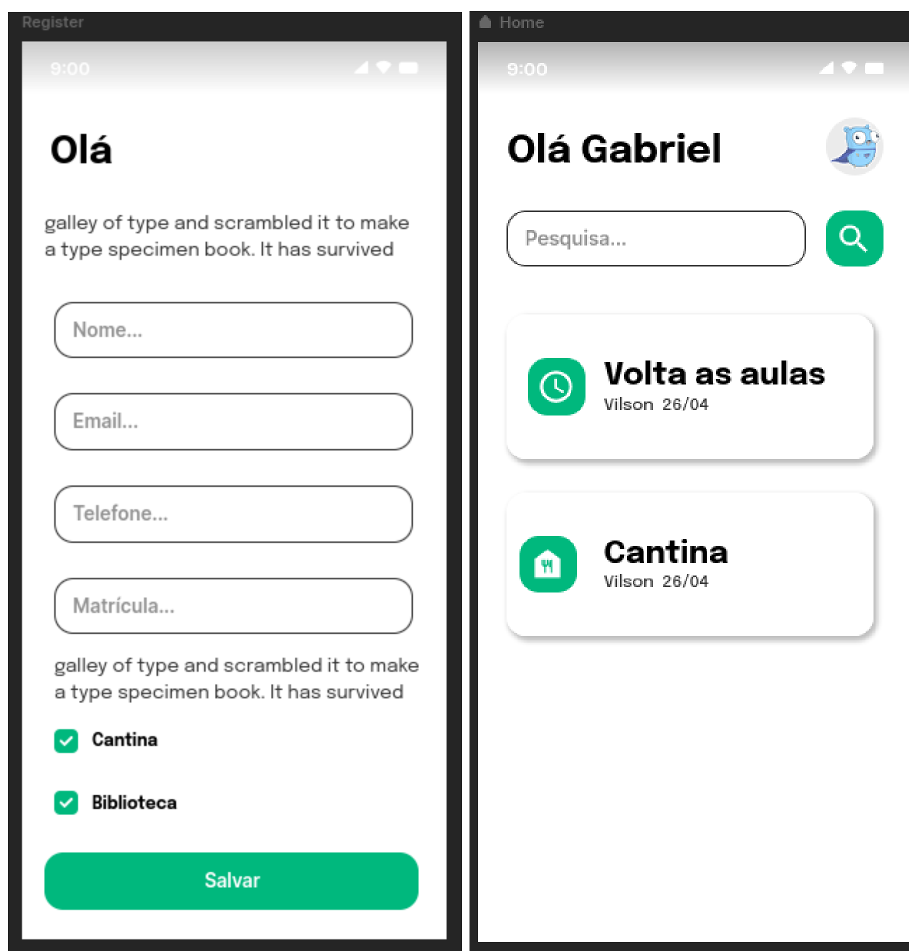

Figura 14. Tela de cadastro do estudante e a *Dashboard* do *Mobile*

4. Apresentação do sistema

Esta seção apresenta o desenvolvimento e as tecnologias utilizadas na concepção desse sistema de comunicação. A subseção 4.1 descreve o funcionamento do ecossistema de microsserviços e as formas de comunicação. A subseção 4.2 apresenta o Sistema *Web* e suas tecnologias é a subseção 4.3 apresenta as funcionalidades presentes no aplicativo *Mobile*.

4.1. Microsserviços

Esta subseção está dividida em quatro subseções. A subseção 4.1.1 apresenta as estratégias de persistência de dados utilizadas pelos microsserviços. A subseção 4.1.2 apresenta a principal forma de comunicação utilizada entre nossos microsserviços. A subseção 4.1.3 e 4.1.4 apresenta os microsserviços e as tecnologias utilizadas.

4.1.1. Banco de Dados

O Banco de Dados utilizado nos microsserviços para armazenamento das informações relevantes aos servidores, estudantes e notificações foi o *PostgresSQL*.

Além das estratégias de armazenamento citadas acima, para criar e acessar as informações do banco de dados *PostgresSQL* foi utilizado o *Framework*, *TypeORM*, uma ferramenta de mapeamento objeto-relacional (ORM) para linguagem *JavaScript*. O *TypeORM* juntamente com o *Framework NestJS* fornecem abstrações e funcionalidades para persistir informações através de uma interface.

A figura 15 apresenta o resultado da normalização dos dados no modelo lógico do banco de dados.

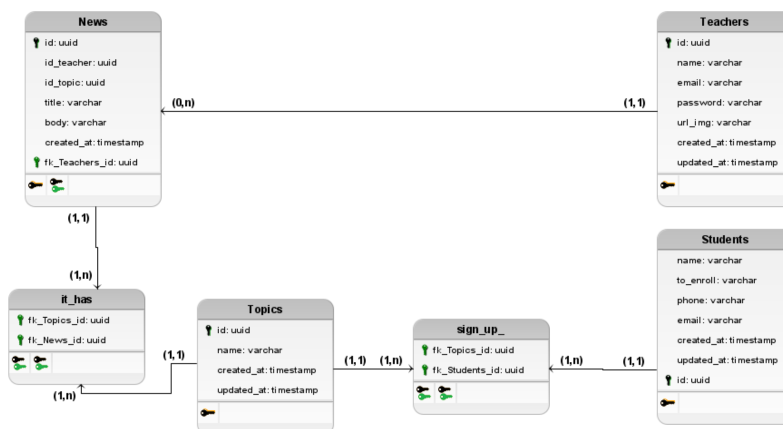

Figura 15. Modelo lógico do banco de dados do microsserviço de notícias

4.1.2. Broker

Para gerenciar e controlar as comunicações entre cada serviço foi utilizado um *Message Broker* a fim de facilitar e centralizar as informações em um único lugar.

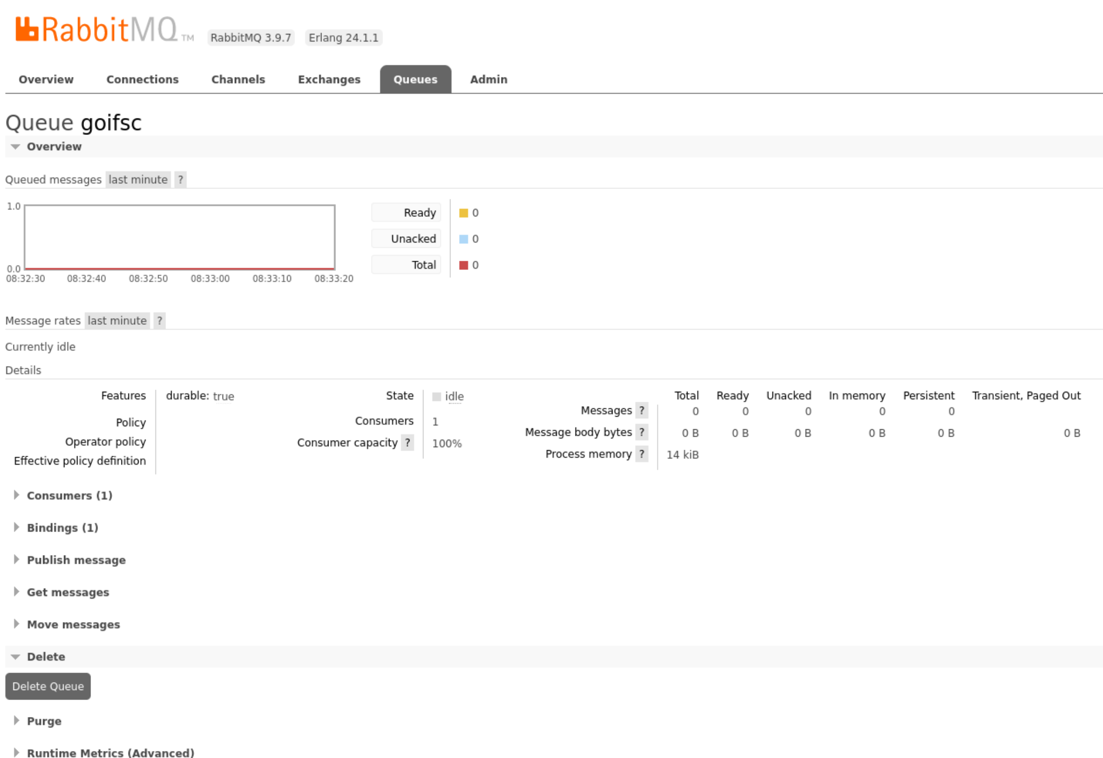

Figura 16. *RabbitMQ* tela de *Queues*

A implementação que utilizamos para gerenciar as filas como modulo intermediário foi em cima do protocolo AMQP mais especificamente usando o *software RabbitMQ*, a figura 16 mostra a tela de gerenciamento de *Queues*, neste contexto estamos trabalhando com uma única *Queue* representada na figura com o nome *goifsc*.

4.1.3. *Gateway*

O *Gateway* foi desenvolvido utilizando o *Framework* de código aberto, *NestJs* juntamente com o *Superset TypeScript* e linguagem de programação *Javascrit*. Para fornecer comunicação, acesso aos dados e às funcionalidades dos sistemas foi utilizado os protocolos *AMQP* e *HTTP* fornecendo rotinas através do padrão *JSON*.

Para tornar a comunicação entre o sistema *Web* e o *Gateway* mais segura, foi adicionado uma estratégia de autenticação para proteger determinadas rotinas que exigem um usuário autorizado para consumi-las. A integração da estratégia de autenticação é feita através da biblioteca de código aberto *passport* e alguns módulos para integrar ao *framework NestJs*.

Após a configuração da estratégia de autenticação, a biblioteca fornece o *Decorator UseGuards* para definir quais rotinas deveram ser autenticadas, o *Decorator* recebe dois paramenteiros denominados como *guards*, que serão os responsáveis por definir a estratégia da autenticação, no sistema, está sendo usado o *JWT* juntamente com *session* que gerá um *token* com validade. A figura 17 apresenta um exemplo a autenticação de uma rotina.

```
@Get(':id')
@HttpCode(HttpStatus.OK)
@UseGuards(SessionAuthGuard, JwtAuthGuard)
public findOne(@Param('id') id: string): Observable<FindTeacherDto> {
  return this.teacherService.findById(id);
}
```

Figura 17. autenticação de uma rotina

Para realizar o envio da notícia e acessar as funcionalidades do sistema, foram criadas as seguintes rotinas, utilizadas tanto no sistema *Web* quanto no *Mobile* para acessar as funcionalidades dos serviços do sistema, As rotinas criadas foram divididas em grupos de acordo com a funcionalidade executada.

O primeiro grupo, apresentado no Quadro 1, contém a funcionalidade de despacho de notificações, utilizado pela aplicação *Web*.

Quadro 1. Rotina de despacho de notificações

Método HTTP	Rota/*Path*	Definição
POST	*/notification*	Envio da notificação
GET	*/notification*	Retorna todas as notificações
GET	*/notification/:title*	Retorna as notificações pelo título
GET	*/notification/:id*	Retorna as notificações pelo Servidor que as enviou
GET	*/notification/by-topics*	Retorna as notificações pelos tópicos
DELETE	*/notification/:id*	Deleta a notificação

O segundo grupo apresenta as rotinas utilizadas para realizar autenticação e gerenciamento da conta dos docentes denominado-se como *create, read, update e delete* (CRUD), conforme apresentado no Quadro 2, também utilizado pela aplicação *Web*.

Quadro 2. Rotina de gerenciamento da conta do docente

Método HTTP	Rota/*Path*	Definição
POST	*/teachers*	Criação da conta do Docente
GET	*/teachers/*:id	Busca pelo Docente
PUT	*/teachers/*:id	Edição dos dados do Docente
DELETE	*/teachers/*:id	Deleta conta do Docente

O terceiro grupo apresenta as rotinas utilizadas no gerenciamento da conta dos estudantes, conforme apresentado no Quadro 3, utilizado pela aplicação *Mobile*.

Quadro 3. Rotina de gerenciamento da conta do estudante

Método HTTP	Rota/*Path*	Definição
POST	*/student*	Criação da conta do Estudante
GET	*/student/*:id	Busca pelo Estudante
PUT	*/student/*:id	Edição dos dados do Estudante

O envio da notícia é realizado através da rotina */notification* usando o método *POST*, onde será enviado alguns dados, como título, tópico e a mensagem. Este método possui validações dos campos, não permitindo o cadastro de informações em branco ou inválidas. Caso alguns dos dados informados sejam inválidos, o *Gateway* retornará uma mensagem sobre o erro e o campo incorreto.

A figura 18 apresenta um exemplo da requisição em *JSON* para realizar o envio da notícia.

```json
{
  "to": "/topics/ciencias",
  "platform": 2,
  "teacher": "9db7bd8f-3263-4752-b55a-d8ed34bf59f3",
  "title": "Aula de Sistemas Distribuidos",
  "message": "O Tema da Aula sera Microsservicos em GoLang <3",
  "topics": [
    "ciencias"
  ]
}
```

Figura 18. *JSON* para enviar uma notícia

Após o envio da requisição, descrita na Figura 15, o *Gateway* irá emitir a requisição para a fila *create-notification* do *Message Broker* que será consumido pelo microsserviço responsável e retornará uma responta em *JSON*. A figura 19 demonstra o método responsável por emitir o cadastro da notícia ao microsserviço responsável.

```typescript
public sendNotification(notificationDto: NotificationDto): Observable<any> {
    this.logger.log(
      `\nNotification to other Microsservice: ${JSON.stringify(
        notificationDto,
      )}\n`,
    );

    return this.httpService
      .post(this.configService
      .get<string>('NOTIFICATION_MICROSSERVICE'), {
        notifications: [notificationDto],
      })
      .pipe(
        map((response) => {
          if (response.status == 200)
            this.client.emit<any>('create-notification', notificationDto);

          return response.data;
        }),
      );
}
```

Figura 19. Método para despachar o cadastro da notícia

O microsserviço possui um método responsável por consumir a fila *create-notification* do *Message Broker*, realizando o cadastro da notícia e o envio da notificação através do microsserviço de notificação. A figura 20 demonstra o método.

```
@EventPattern('create-notification')
public async createNotification(
  @Ctx() context: RmqContext,
  @Payload() notification: CreateNotificationDto,
): Promise<void> {
  const channel = context.getChannelRef();
  const originalMessage = context.getMessage();

  try {
    this.logger.log(
      `\nCreate Notification: ${JSON.stringify(notification)}\n`,
    );

    await this.notificationService.create(notification);
    await channel.ack(originalMessage);
  } catch (channelError) {
    const filterAckError = this.ackErrors.filter((ackError) =>
      channelError.error.code.includes(ackError),
    );

    if (filterAckError.length > 0) {
      await channel.ack(originalMessage);

      throw new CustomException(channelError.error.message);
    }
  }
}
```

Figura 20. Método responsável por consumir a fila

4.1.4. Microsserviço de Notificações

O microsserviço de Notificações utilizado na nossa arquitetura e um projeto de código aberto criando na linguagem de programação *Go* e mantido pelo autor Bo-Yi-Wu (2021). As úncias ações necessárias para poder utilizar este serviços na nossa arquitetura foi:

- Configurar o microsserviço para que ele executasse seguindo o protocolo HTTP com as devidas regras do padrão *Rest*;
- Respeitar o contrato do *endpoit* que iria despachar o JSON para o serviço da *Google*;
- Configurar as credenciais para que o microsserviço tivesse acesso ao projeto do *Firebase*.

O quadro 8 apresenta todos as rotas fornecidas pelo microserviço:

Quadro 8. Microsserviço de Notificações

Método HTTP	Rota/*Path*	Definição
GET	*/api/stat/go*	Informações de uso da CPU e Memória
GET	*/api/stat/app/*:id	Quantidade de notificações enviadas com sucesso ou falha
GET	*/api/config/*:id	*Endpoint* para gerar arquivo de configuração do microsserviço
POST	*/api/push/*:id	*Endpoint* para despachos das notificações

O envio da notificação é realizado através de uma rotina */api/push*, onde será enviado alguns dados referente ao corpo da notificação no formato *JSON*, igual demonstrado na figura 21.

```
{
  "notifications": [
    {
      "topic": "ciencias",
      "platform": 3,
      "message": "Este e o corpo da notificacao!",
      "title": "Este e o titulo"
    }
  ]
}
```

Figura 21. *JSON* para envio de notificações

4.2. Aplicação Web

O sistema *Web* foi desenvolvido usando o *Framework VueJs*, uma estrutura para construir aplicativos clientes progressivos (PWA) de página única (SPA), usando *HTML* e *Javascript*. A arquitetura utilizada para o projeto *Web*, consiste em uma arquitetura baseada em módulos, onde cada módulo possui responsabilidade única, divido em duas principais funcionalidades, as páginas onde será implementado os componentes e a gerencia de estados *Vuex* que será responsável por garantindo que o estado só possa ser alterado de forma previsível. A figura 22 detalha o fluxo do módulo.

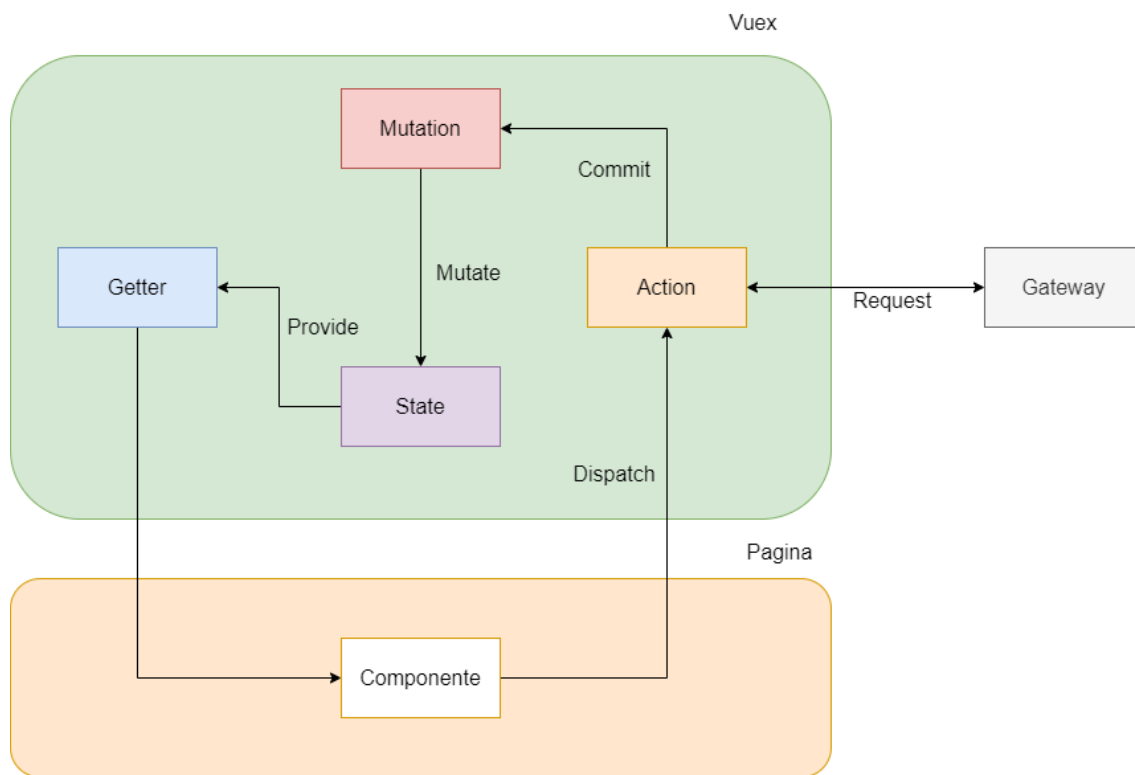

Figura 22. Fluxo do módulo adaptado de Vuejs (2021b)

A aplicação será responsável por gerenciar o envio das notificações, onde cada docente poderá selecionar um ou mais tópicos referente a notícia e despachá-la para os estudantes.

A página de *Login* possui validações em todos os campos, não permitindo inserir informações em branco e nem inválidas, após preenchimento de todos os dados corretamente, o botão de login é habilitado automaticamente, caso ocorra algum erro na comunicação com o *Gateway*, será mostrado um *SnackBar* informando o usuário sobre o erro ocorrido, caso o docente não esteja cadastrado no sistema, será possível acessar o formulário para cadastra-lo através da tela de login. A figura 23 demonstra a página em detalhes.

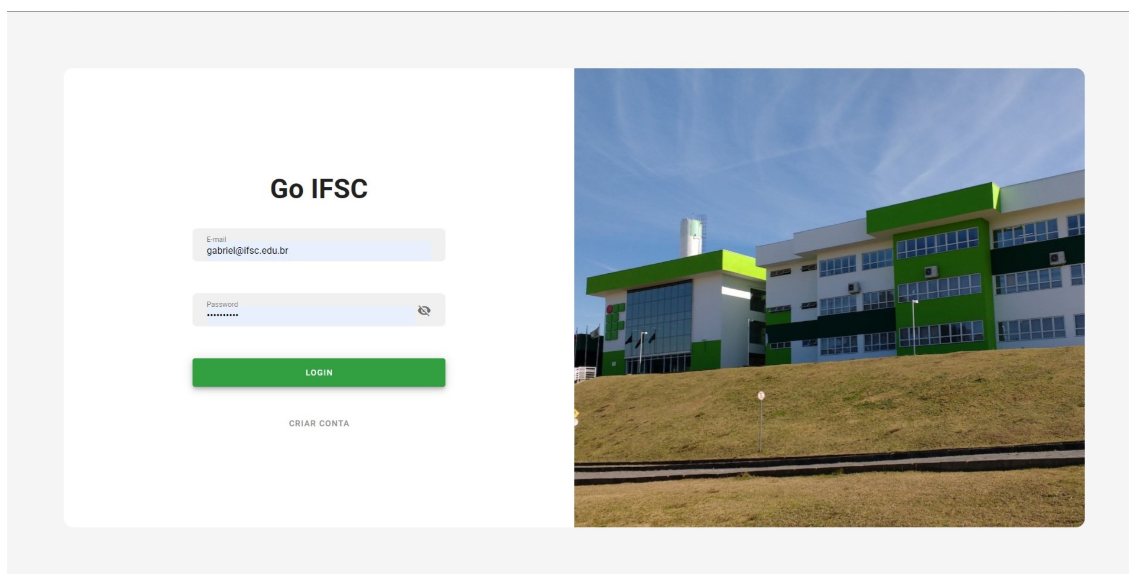

Figura 23. Página de *Login*

Após o *Login* o docente irá acessar a página de *Dashboard*, responsável por visualizar as notícias já cadastradas no sistema e despachar uma nova notícia para os estudantes, os campos do formulário possuem validação para que não seja informado nenhuma informação inválida. A figura 24 demonstra a página em detalhes.

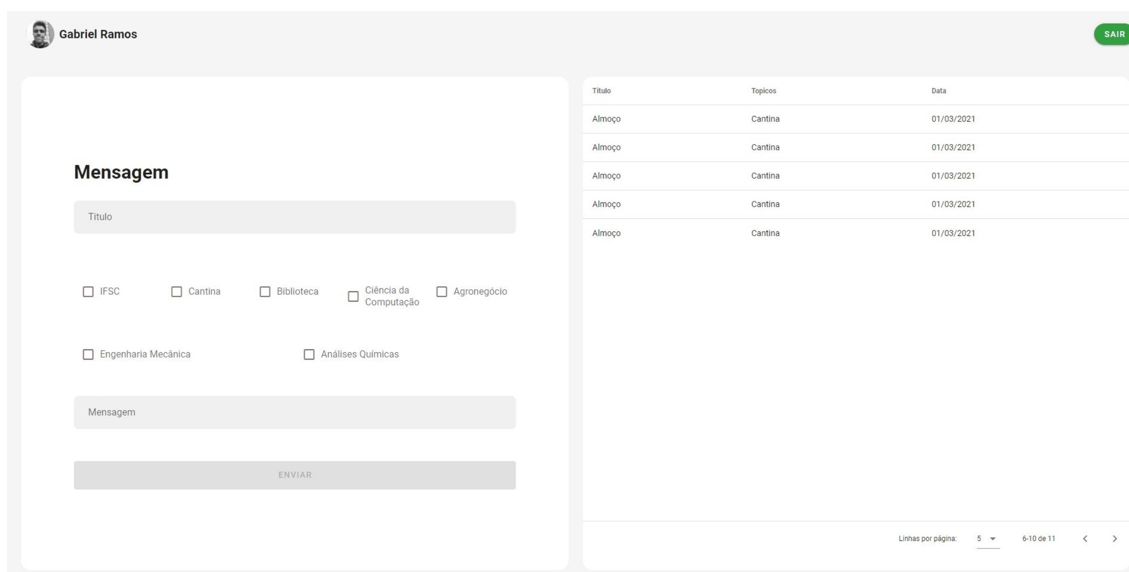

Figura 24. Página de *Dashboard*

4.3. Aplicação Mobile

O aplicativo *Mobile* foi desenvolvido usando a linguagem de programação *Dart*, que em 2017 teve um grande avanço com a criação do *Toolkit Flutter* que é mantido pelo *Google*. O aplicativo *Mobile* possui funcionalidades exclusivas para os estudantes do Câmpus, pois a partir dele será possível receber e visualizar as informações que serão enviadas via

Push Notification. Para estabelecer uma comunicação e enviar as requisições gerenciar os tópicos a serem escutados entre outras funcionalidades, o aplicativo foi construído com base em uma arquitetura limpa. A figura 25 detalha melhor cada camada implementada.

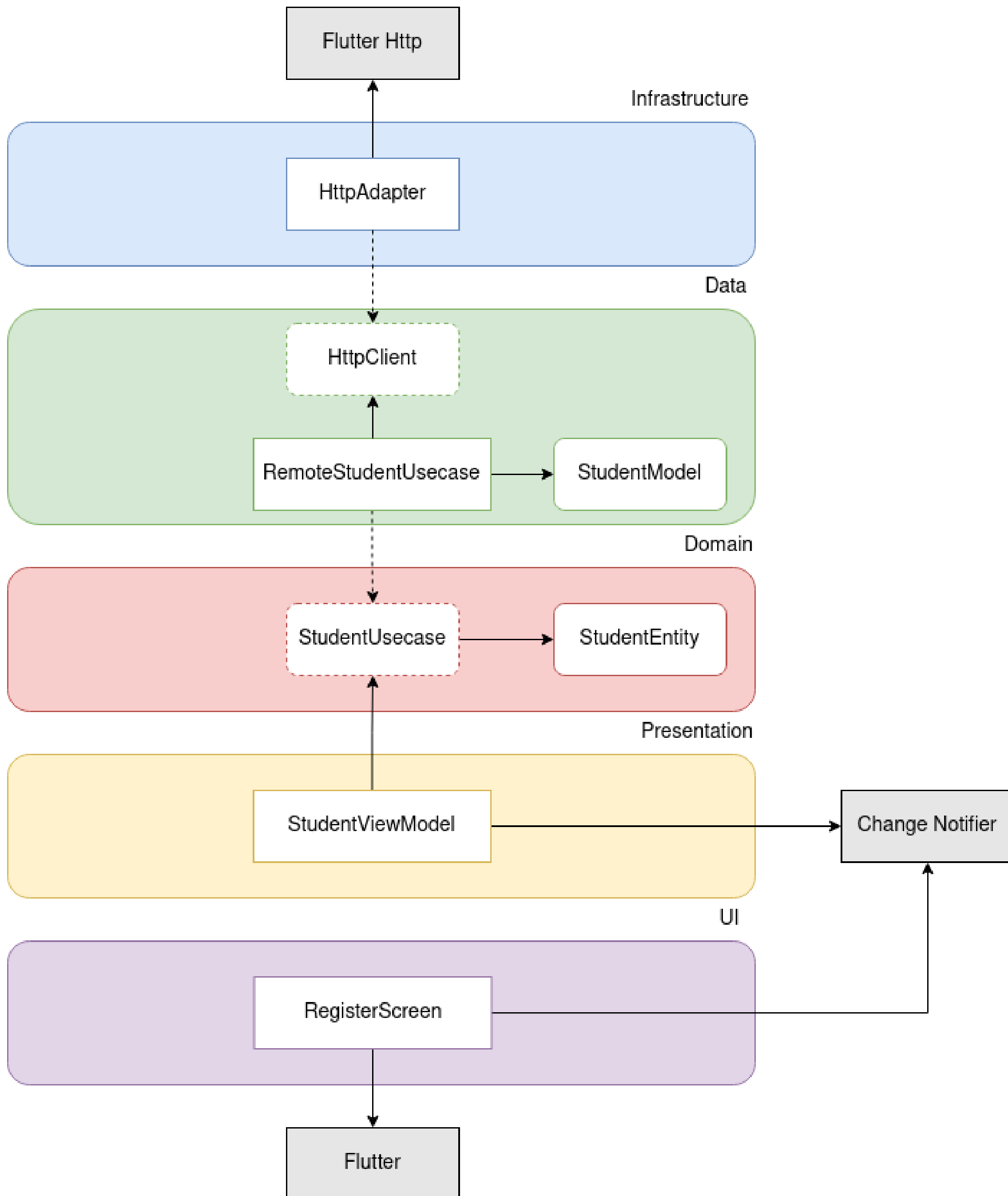

Figura 25. Representação da arquitetura em camadas

A camada de *infrastructure* (Infraestrutura) e a camada onde isolamos bibliotecas de terceiros, como mostrado na figura 25. Uma das implementações que temos nesta camada e a do protocolo *http*, a biblioteca que usamos de mesmo nome e mantida pelo time de engenharia da *Google*. A figura 26 demonstra a implementação da biblioteca *http*. Esta classe implementa outra classe abstrata chamada *HttpClient*, essa classe se encontra

na camada *Data* e ela nós obriga a assinar um contrato de implementação chamado *request* assim podemos inverter a dependência chamando diretamente a assinatura do método da classe abstrata, desacoplando o código.

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

  HttpAdapter({
    required this.client,
  });

  @override
  Future<dynamic> request({
    required String url,
    required HttpMethod method,
    Map<String, dynamic>? body,
    Map<String, String>? headers,
  }) async { ... }

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

Figura 26. Implementação da biblioteca *http*

Além de isolar esta implementação que possui dependência direta com uma biblioteca externa, também utilizamos da injeção de dependências para poder prover as funcionalidades necessárias para funcionamento desta camada. Se no futuro precisarmos trocar a biblioteca de chamadas *http* por outra seria necessário mexer somente nesta implementação.

A camada de *Data* (Dados) possui algumas funcionalidades importantes nessa arquitetura, uma delas e a representação do modelo de dados que serão obtidos do *gatway*. A figura 27 mostra os dados do modelo que iremos obter ao realizar a *request* no *gatway* especificamente para obter os dados dos estudantes.

```
class StudentModel {
  late final String id;
  late final String name;
  late final String phone;
  late final String email;
  late final List<String> topics;
  late final String matriculation;

  StudentModel({ ... });

  factory StudentModel.fromJson(Map<String, dynamic> json) {
    return StudentModel( ... );
  }

  StudentEntity get toEntity => StudentEntity( ... );
}
```

Figura 27. Implementação do modelo de estudante

Essa classe tem duas funções muito importantes além de representar os dados que entraram na aplicação. A *factory fromJson* e a fábrica responsável por receber os dados no formato *JSON* e converte-lo para nossa classe de modelo. A figura 28 demonstra a implementação desta fábrica.

```
factory StudentModel.fromJson(Map<String, dynamic> json) {
  return StudentModel(
    id: json['id'],
    name: json['name'],
    phone: json['phone'],
    email: json['email'],
    topics: json['topics'].cast<String>(),
    matriculation: json['matriculation'],
  );
}
```

Figura 28. Implementação da *factory fromJson*

Outra método importante e o *getter toEntity* como o próprio nome diz esse *getter* e responsável por a partir do modelo devolver nossa entidade, *StudentEntity*. A figura 29 mostra a implementação deste *getter*.

```
StudentEntity get toEntity => StudentEntity(
  id: id,
  name: name,
  phone: phone,
  email: email,
  topics: topics,
  matriculation: matriculation,
);
```

Figura 29. Implementação do *getter toEntity*

Também e função desta camada conter o código dos casos de uso que irão chamar implementações da camada de *infrastructure*. A figura 30 mostra a implementação do caso de uso referente a criação do estudante, mais uma vez utilizamos a injeção de dependências para prover as funcionalidades necessárias para o funcionamento desta classe. Essa classe estende de outra classe abstrata chamada *StudentUsecase* que se encontra na camada de *Domain*, a classe *StudentUsecase* contem a assinatura do método de criação do estudante e seus parâmetros.

```
class RemoteStudentUsecase extends StudentUsecase {
  late final HttpClient httpClient;

  RemoteStudentUsecase({
    required this.httpClient,
  });

  @override
  Future<StudentEntity> create(
    StudentUsecaseCreateParams params,
  ) async {
    try {
      final _response = await httpClient.request(
        body: params.toJson,
        method: HttpMethod.post,
        url: AppContants.urlStudent,
      );

      return StudentModel
        .fromJson(_response as Map<String, dynamic>)
        .toEntity;
    } on HttpError catch (_) {
      rethrow;
    }
  }
}
```

Figura 30. Implementação do caso de uso *RemoteStudentUsecase*

A camada de *Domain* (Domínio), contem todo o domínio que estamos trabalhando ou seja aqui estão os dados que iremos prover para as demais camadas. A figura 31 mostra a entidade de estudante.

```
class StudentEntity extends Equatable {
  final String id;
  final String name;
  final String phone;
  final String email;
  final List<String> topics;
  final String matriculation;

  const StudentEntity({ ... });

  const StudentEntity.empty({ ... });

  StudentEntity copyWith({ ... }) {
    return StudentEntity( ... );
  }

  @override
  String toString() => '''
    {
      "id": "$id",
      "name": "$name",
      "email": "$email",
      "phone": "$phone",
      "topics": "$topics",
      "matriculation": "$matriculation"
    }
  ''';

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

Figura 31. Implementação da entidade estudante

Também e responsabilidade desta camada conter os casos de uso que por sua vez possuem as classes abstratas com as assinaturas utilizadas na camada *Data*. A figura 32 mostra a implementação da classe abstrata que prove a assinatura do método de criação do estudante e sua classe de parâmetro.

```
abstract class StudentUsecase {
  Future<StudentEntity> create(
    StudentUsecaseCreateParams params,
  );
}

class StudentUsecaseCreateParams extends Equatable {
  final String name;
  final String phone;
  final String email;
  final List<String> topics;
  final String matriculation;

  const StudentUsecaseCreateParams({ ... });

  Map<String, dynamic> get toJson => { ... };

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

Figura 32. Implementação do caso de uso do estudante

A camada de *Presentation* (Apresentação) e a camada onde contem toda a nossa gerencia de estados, como o *Flutter* possui a criação de interfaces declarativas a necessidade de gerenciar estados para atualizar o contexto da aplicação e inevitável. Optamos por utilizar recursos que o próprio *Toolkit* nos fornece, o *ChangeNotifier* e uma classe que pode ser herdada ou combinada para fornecer uma API de notificações de alteração usando *VoidCallback* para notificações, isso remete ao padrão de projeto *Observer*. O *ChangeNotifier* é otimizado para pequenos números (um ou dois) de ouvintes. É O(N) para adicionar e remover ouvintes e $O(N^2)$ para despachar notificações (onde N é o número de ouvintes).

A figura 33 mostra a implementação do nosso *ChangeNotifier* seguindo o padrão de projeto *Model View View Model* (MVVM).

```dart
class StudentViewModel extends ChangeNotifier {
  late final StudentUsecase studentUsecase;
  late final StorageUsecase storageUsecase;

  StudentViewModel({
    required this.studentUsecase,
    required this.storageUsecase,
  });

  bool _loading = false;
  bool get loading => _loading;
  Future<void> setLoading(bool loading) async {
    _loading = loading;
    notifyListeners();
  }

  bool _successful = false;
  bool get successful => _successful;
  Future<void> setSuccessful(bool successful) async {
    _successful = successful;
    notifyListeners();
  }

  String _error = '';
  String get error => _error;
  Future<void> setError(String error) async {
    _error = error;
    notifyListeners();
  }

  bool formIsValid(GlobalKey<FormState> formKey) =>
      formKey.currentState?.validate() ?? false;

  Future<void> handlerCreateStudent(
    StudentUsecaseCreateParams params,
  ) async { ... }

  Future<void> _handlerTopicsFirebase(List<String> topics) async { ...
    }
}
```

Figura 33. Implementação do MVVM do estudante

O importante dessa implementação são as funções chamadas de *notifyListeners*, elas que irão notificar a interface de usuário que algo aconteceu ou esta acontecendo.

A camada de *UI* (Interface de usuário) e a camada que conterá as interfaces de usuário, umas das vantagens de utilizar um arquitetura neste nível e a capacidade de isolar as coisas, com o *Flutter* não e diferente o único lugar que o SDK do *Flutter* esta sendo usado e nesta camada e na camada *Main*. Esta camada também contem toda a parte de validação dos campos, por exemplo o cadastro do estudante e todo validado com base em regras criadas por nos.

Eventualmente o time de engenharia do *Google* muda como os *Widgets* são cons-

truídos ou declarados, com essa abordagem de arquitetura fica mais fácil controlar essas mudanças externas que não estão sobre nosso controle, focando sua implementações em pequenos *Widgets* separados, seguindo outro principio de implementação de UI o *Atomic Design*. A figura 34 mostra a implementação da interface de cadastro de estudantes.

```
class RegistrationScreen extends StatefulWidget {
  const RegistrationScreen({Key? key}) : super(key: key);

  @override
  _RegistrationScreenState createState() => _RegistrationScreenState();
}

class _RegistrationScreenState extends State<RegistrationScreen>
    with KeyboardManagerMixin {
  late final List<String> _topics = [Topics.ifsc.topic];
  late final _debouncer = Debaouncer(milliseconds: 4);
  late StudentEntity _studentEntity = const StudentEntity.empty();
  late final GlobalKey<FormState> _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Theme.of(context).colorScheme.background,
      appBar: AppBar(
        elevation: 0,
        title: Text(
          AppLocalizations.of(context)!.appScreenRegistrationTitle,
          style: Theme.of(context).textTheme.headline4,
        ),
        backgroundColor: Theme.of(context).colorScheme.background,
      ),
      body: SafeArea(
        child: Consumer<StudentViewModel>( ... );
          },
        ),
      ),
    );
  }

  void _setTopics(bool isMarked, String topic) { ... }

  void _handlerError({
    required String message,
    required BuildContext context,
  }) { ... }

  void _navigateToHome(BuildContext context, String name) { ... }

  Future<void> _handlerCreate(StudentEntity params) async { ... }
}
```

Figura 34. Implementação da interface de cadastro do estudante

O importante dessa implementação e o *Widgets Consumer*, esse *Widgets* por meio

de *generics* e o contexto do *Flutter* consegue identificar e ouvir nosso *View Model*, como ele possui um *Builder* e um próprio contexto toda atualização que acontece no *View Model* ele reage e reconstrói o que for necessário.

Também temos a camada *Main* (Principal ou Inicial) a camada mais importante da arquitetura pois e nela que resolvemos toda a injeção de dependências para que as demais camadas funcionem bem. Decidimos prover nossas dependências manualmente sem o uso de bibliotecas de terceiros, desta forma temos um maior controle e nenhuma dependência além do próprio *Dart* e o *Flutter*. A figura 35 mostra a implementação da injeção de dependência.

```
class DependencyInjectRegisterWidiget extends StatelessWidget {
  final Widget child;

  const DependencyInjectRegisterWidiget({
    Key? key,
    required this.child,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      lazy: true,
      create: (_) => StudentViewModel(
        studentUsecase: RemoteStudentUsecase(
          httpClient: HttpAdapter(
            client: Client(),
          ),
        ),
        storageUsecase: RemoteStorageUsecase(
          storageClient: StorageAdapter(
            storageAdapter: StorageSingleton.instance.preferences,
          ),
        ),
      ),
      child: child,
    );
  }
}
```

Figura 35. Implementação da injeção de dependência

A injeção de dependência e um padrão de projeto que visa manter um baixo nível de acoplamento entre as diversas camadas da arquitetura. A ideia e satisfazer todas as instâncias e dependências das classes pretendias via construtor, sendo assim o ponto que amarra todas as camadas.

O resultado do aplicativo, utilizando a arquitetura em camadas e seguindo o protótipo, foi a tela de cadastro do estudante onde é possível selecionar os tópicos que irá se inscrever para receber as notícias que julga relevante, após o cadastro será exibido a tela de *Dashboard* com as notícias cadastradas no sistema sendo possível filtrá-las pelo campo de pesquisa e por fim ao tocar no *Card* será exibido o detalhamento da notícia. A figura 36 detalha melhor as telas citadas.

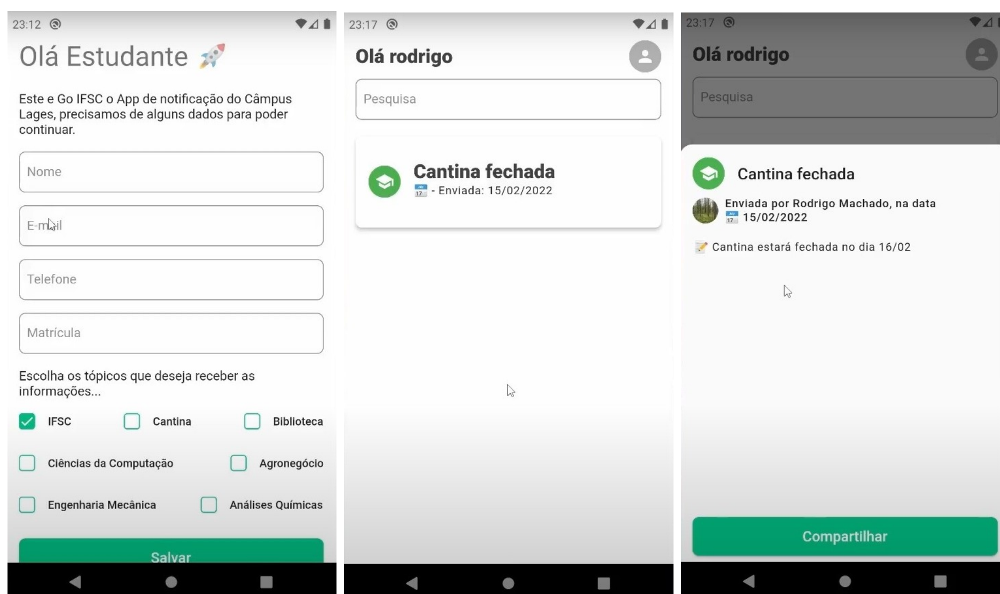

Figura 36. Tela de Cadastro, *Dashboard* e Detalhes da Notícia

5. Conclusão

Este trabalho apresentou o desenvolvimento de uma plataforma de Notícias para instituições educacionais (**GoIFSC**), sendo composto por uma aplicação *Web*, um aplicativo *Mobile* e um conjunto de microsserviços, para auxiliar os docentes no compartilhamento de noticias do Câmpus.

A aplicação *Web* e o aplicativo *Mobile* utilizam-se dos serviços disponibilizados pelo *Gateway*. Ao preencher o formulário para enviar uma notícia e selecionar um ou mais tópicos, a mesma será cadastrada no banco, e o aplicativo irá receber uma notificação informando o estudante com o título e a descrição, a notícia ficará disponível tanto na tela principal da aplicação *Web* quanto a *Mobile*.

As aplicações *Web* e *Mobile* atingiram todos os requisitos levantados, os microsserviços também atingiram todos os objetivos propostos segundo a arquitetura definida para o sistema. Desta forma os objetivos foram cumpridos parcialmente, pois não foi possível realizar a avaliação da usabilidade da plataforma, devido a restrições de implantação e disponibilidade de recursos em nosso cronograma, por estes motivos não realizamos os testes.

Como trabalhos futuros, fica a possibilidade de implantar o sistema na infraestrutura do Câmpus, realizar os testes de integração e aceitação, adicionar novos recursos de acessibilidade e a possibilidade de editar as informações dos docentes na aplicação *Web* e as informações do estudante na aplicação *Mobile*, bem como a possibilidade de envio de imagens via *Push Notification*.

Outro aspecto a ser implementado é a possibilidade da integração de novos microsserviços bem como a visualização de notas e horários das aulas para os estudantes, essa abordagem é possível devido a adoção da arquitetura de microsserviços no formato *Plugin in Play*.

Referências

Bo-Yi-Wu (2021). Disponível em: <https://github.com/appleboy/gorush>. Acesso em 08 out 2021.

Castro, M. F. e Tedesco, P. (2014). Aplicação de conceitos de wayfinding em interfaces mobile de recomendação de rota. In *Anais do X Simpósio Brasileiro de Sistemas de Informação*, pages 470–481. SBC.

da Fonseca, C. H. C. e Balbino, F. (2019). Uma aplicação web para produção de textos narrativos com enredos alternativos. In *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, volume 8, page 1272.

Dart (2021). Disponível em: <https://dart.dev>. Acesso em 08 jun 2021.

de Oliveira, C. (2015). Tic's na educação: a utilização das tecnologias da informação e comunicação na aprendizagem do aluno. *Pedagogia em ação*, 7(1).

FCM, F. (2021). Disponível em: <https://firebase.google.com/docs/cloud-messaging/>. Acesso em 10 jun 2021.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

Firebase (2021). Disponível em: <https://firebase.google.com/docs>. Acesso em 10 jun 2021.

Flutter (2021). Disponível em: <https://flutter.dev>. Acesso em 08 jun 2021.

Fowler, S. J. (2017). *Production-ready microservices: building stable, reliable, fault-tolerant systems*. OReilly.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., e Patterns, D. (1995). *Elements of reusable object-oriented software*, volume 99. Addison-Wesley Reading, Massachusetts.

Hohpe, G. e Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

IFSC (2019). Histórico. Disponível em: <https://www.ifsc.edu.br/web/campus-lages/historico>. Acesso em 11 jun 2021.

Joseph, R. J. (2015). Single page application and canvas drawing. *arXiv preprint arXiv:1502.03530*.

Miranda, G. L. (2016). Limites e possibilidades das tic na educação. *Sísifo*, pages 41–50.

Nestjs (2021). Disponível em: <https://docs.nestjs.com>. Acesso em 09 jun 2021.

Neto Moreira, M. P. M., Augusto, V. d. S., et al. (2021). Padrões para produção de aplicações utilizando microsserviços. *Padrões para produção de aplicações utilizando Microsserviços*.

Paula, Clarice Santos Grazziotin, d. J. (2020). *Prefácio*. IFSC.

PostgreSQL (2021). Disponível em: <https://www.postgresql.org/about/>. Acesso em 06 jun 2021.

RabbitMQ (2021). Disponível em: <https://www.rabbitmq.com>. Acesso em 06 jun 2021.

Robbins, S. P. (2005). *Comportamento organizacional*, volume 11. Pearson Prentice Hall.

Robbins, S. P., Judge, T. A., e Sobral, F. (2011). *Comportamento organizacional: teoria e prática no contexto brasileiro*, volume 14. Pearson Prentice Hall.

Salvadori, I. L. et al. (2015). Desenvolvimento de web apis restful semânticas baseadas em json. *Desenvolvimento de Web APIs RESTful semânticas baseadas em JSON*.

Shaw, M. e Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*, volume 1. prentice Hall Englewood Cliffs.

Silva, G. N. d. e Silva, T. N. d. (2017). Desenvolvimento de uma aplicação para disponibilização de aplicativos classificados conforme taxonomia de bloom. *DESENVOLVIMENTO DE UMA APLICAÇÃO PARA DISPONIBILIZAÇÃO DE APLICATIVOS CLASSIFICADOS CONFORME TAXONOMIA DE BLOOM*.

Silva, J. K. e Tiosso, F. (2020). Revisão bibliográfica sobre conceito de progressive web applications (pwa). *Revista Interface Tecnológica*, 17(1):53–64.

Tolfo, C., Wazlawick, R. S., Ferreira, M. G. G., e Forcellini, F. A. (2011). Agile methods and organizational culture: Reflections about cultural levels. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(6).

Vuejs (2021a). Disponível em: <`https://vuejs.org/v2/guide/#What-is-Vue-js`>. Acesso em 09 jun 2021.

Vuejs (2021b). Disponível em: <`https://vuex.vuejs.org/vuex.png`>. Acesso em 09 jun 2021.