**UNIVERSIDADE ESTADUAL DO RIO GRANDE DO SUL**

**ENGENHARIA DE COMPUTAÇÃO**

**VANESSA LOPES KLEIN**

**DESENVOLVIMENTO DE UMA APLICAÇÃO MOBILE PARA GERENCIAMENTO DE PROJETOS CIENTÍFICOS NA COMUNIDADE ACADÊMICA**

**GUAÍBA**

**2021**

**VANESSA LOPES KLEIN**

**DESENVOLVIMENTO DE UMA APLICAÇÃO MOBILE PARA GERENCIAMENTO DE PROJETOS CIENTÍFICOS NA COMUNIDADE ACADÊMICA**

Trabalho de Conclusão de Curso. Apresentado como requisito parcial para obtenção do título de Bacharelado na Universidade Estadual do Rio Grande do Sul.

Orientadora: Prof. Dra. (MSc) Débora Matos

**GUAÍBA**

**2021**

**VANESSA LOPES KLEIN**

**DESENVOLVIMENTO DE UMA APLICAÇÃO MOBILE PARA GERENCIAMENTO DE PROJETOS CIENTÍFICOS NA COMUNIDADE ACADÊMICA**

Trabalho de Conclusão de Curso. Apresentado como requisito parcial para obtenção do título de Bacharelado na Universidade Estadual do Rio Grande do Sul.

Aprovada em ..../..../.....

BANCA EXAMINADORA:

Prof. Dra. Adriane Parraga

Universidade Estadual do Rio Grande do Sul

Prof. Dra. Fabrícia Damando

Universidade Estadual do Rio Grande do Sul

Prof. Dr. Fulano

Universidade Estadual do Rio Grande do Sul

Este trabalho dedico ...........................

.....................

**RESUMO**

Pretende-se neste trabalho, abordar a questão dos custos relacionados à produção da silagem, como um modo alternativo para a alimentação do gado em época de seca. Para isso foram feitos levantamentos bibliográficos sobre custos e silagem, verificou-se também características da região do Mato Grosso do Sul, em especial no município de Tacuru onde está situada a sede da fazenda analisada e com base nos dados coletados calculou-se os custos de produção da silagem. O estudo apresentado é baseado fundamentalmente na revisão literária de custos, onde se encontram suas classificações, formas e métodos de custeio. Utilizou-se na metodologia os tipos de pesquisa quanto aos objetivos, quanto aos procedimentos e quanto à abordagem do problema, empregando a pesquisa exploratória, o estudo de caso, a pesquisa bibliográfica, documental e classificando como abordagem qualitativa. Tratando-se de contabilidade rural, buscou-se também colaborar com os agricultores e pecuaristas e àqueles que desejarem ter um conhecimento sobre o custo e a produção da silagem.

Palavras chave: Custo. Silagem. Gado.

O resumo, para Martins (2000, p.51), “trata-se da apresentação concisa dos pontos relevantes de um texto. Visa fornecer elementos capazes para permitir ao leitor decidir sobre a necessidade de consulta integral do texto. O resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do trabalho”.

O resumo da monografia deve conter no máximo 250 palavras e ser composto de um único parágrafo com espaço simples entre as linhas.

**LISTA DE FIGURAS**

|  |  |
| --- | --- |
| Figura 1 – Modelo geral da aplicação | 10 |
| Figura 2 – Prototipação do aplicativo na ferramenta Figma | 11 |
| Figura 3 –Título ................................................................................................... | 12 |
| Figura 4 –Título ................................................................................................... | 13 |

Deve-se utilizar a lista de ilustrações caso ocorram, durante o texto, um número maior que três, caso contrário esta lista torna-se opcional.

De acordo com a ABNT 14724, toda e qualquer ilustração, foto, fluxograma, esquema, passa a se denominar figura e receberá uma numeração seqüencial.

As Figuras serão numeradas com algarismos arábicos de forma seqüencial e independente do Capítulo onde estão inseridas

**LISTA DE GRÁFICOS**

|  |  |
| --- | --- |
| Gráfico 1 –Título .................................................................................................. | 10 |
| Gráfico 2 –Título .................................................................................................. | 11 |
| Gráfico 3 –Título .................................................................................................. | 12 |
| Gráfico 4 –Título .................................................................................................. | 13 |

Deve-se utilizar a lista de gráficos caso ocorram, durante o texto, um número maior que três, caso contrário esta lista torna-se opcional.

Será considerada gráfico, toda ilustração que contiver obrigatoriamente um tratamento estatístico em forma de gráfico.

Os Gráficos serão numeradas com algarismos arábicos de forma seqüencial e independente do Capítulo onde estão inseridas

**LISTA DE QUADROS**

|  |  |
| --- | --- |
| Quadro 1 –Título ................................................................................................. | 10 |
| Quadro 2 –Título ................................................................................................. | 11 |
| Quadro 3 –Título ................................................................................................. | 12 |
| Quadro 4 –Título ................................................................................................. | 13 |

Deve-se utilizar a lista de quadros caso ocorram, durante o texto, um número maior que três, caso contrário esta lista torna-se opcional.

Será considerada Quadro, toda ilustração que contiver obrigatoriamente descrição de textos na forma ilustrativa, demonstrativa, comparativa, entre outras. Não admite-se números estatísticos no quadro, apenas descritivos.

Os quadros serão numeradas com algarismos arábicos de forma seqüencial e independente do Capítulo onde estão inseridas

**LISTA DE TABELAS**

|  |  |
| --- | --- |
| Tabela 1 –Título .................................................................................................. | 10 |
| Tabela 2 –Título .................................................................................................. | 11 |
| Tabela 3 –Título .................................................................................................. | 12 |
| Tabela 4 –Título .................................................................................................. | 13 |

Deve-se utilizar a lista de tabelas caso ocorram, durante o texto, um número maior que três, caso contrário esta lista torna-se opcional.

Será considerada Tabela, toda ilustração que contiver obrigatoriamente um tratamento estatístico de seus dados.

As Tabelas quadros serão numeradas com algarismos arábicos de forma seqüencial e independente do Capítulo onde estão inseridas

**LISTA DE ABREVIATURAS E SIGLAS**

CRUD *Create, Read, Update, Delete*

API *Application Programming Interfaces*

JSON *JavaScript Object Notation*

SQL *Structured Query Language***SUMÁRIO**

[**1**](#_heading=h.1rvwp1q) **INTRODUÇÃO 3**

[1.1](#_heading=h.4bvk7pj) TEMA E PROBLEMA 3

[1.2](#_heading=h.2r0uhxc) OBJETIVOS 3

[1.2.1](#_heading=h.1664s55) Objetivo Geral 3

[1.2.2](#_heading=h.3q5sasy) Objetivos Específicos 3

[1.3](#_heading=h.25b2l0r) JUSTIFICATIVA 3

[**2**](#_heading=h.kgcv8k) **FUNDAMENTAÇÃO TEÓRICA 3**

[2.1](#_heading=h.3tbugp1) O COOPERATIVISMO A PARTIR DA REVOLUÇÃO INDUSTRIAL 3

[2.2](#_heading=h.34g0dwd) DESENVOLVIMENTO HISTÓRICO DO SETOR ATACADISTA E VAREJISTA 3

[**3**](#_heading=h.1jlao46) **METODOLOGIA 3**

[3.1](#_heading=h.43ky6rz) DELINEAMENTO DA PESQUISA 3

[3.2](#_heading=h.2iq8gzs) INSTRUMENTO DE COLETA DE DADOS 3

[3.3](#_heading=h.xvir7l) PROCEDIMENTOS DE COLETA E ANÁLISE DOS DADOS 3

[3.4](#_heading=h.3hv69ve) LIMITAÇÕES DA PESQUISA 3

[**4**](#_heading=h.1x0gk37) **COLETA E ANÁLISE DE DADOS 3**

[**5**](#_heading=h.2zbgiuw) **CONCLUSÃO 3**

[**REFERÊNCIAS 3**](#_heading=h.4h042r0)

[**GLOSSÁRIO 3**](#_heading=h.2w5ecyt)

[**APÊNDICES 3**](#_heading=h.1baon6m)

[**ANEXOS 3**](#_heading=h.3vac5uf)

1. **INTRODUÇÃO**

* 1. TEMA E PROBLEMA

* 1. OBJETIVOS

* + 1. Objetivo Geral

* + 1. Objetivos Específicos

* 1. JUSTIFICATIVA

1. **FUNDAMENTAÇÃO TEÓRICA**

1. **PROPOSTA**

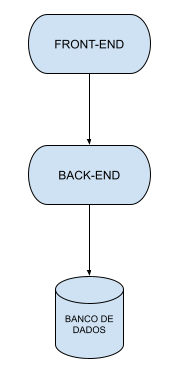
Para solucionar o problema descrito neste trabalho foi proposto o desenvolvimento de uma aplicação onde será possível que os usuários encontrem facilmente projetos, onde possam se inscrever, e também a possibilidade de cadastrar um projeto para assim procurar por bolsistas.

Para implementar o software proposto, foi desenvolvido um aplicativo mobile intitulado de *SharedProjects,* que representa a ideia da aplicação de possibilitar o compartilhamento acessível de projetos.

* 1. MODELAGEM DA PROPOSTA DO SISTEMA

Para desenvolver a aplicação mobile proposta e com as funcionalidades (CRUD) o trabalho requer de três etapas de desenvolvimento e planejamento essenciais, sendo elas:

* *Front-end:* trata-se da parte visual, onde o usuário pode interagir com o sistema, a interface do usuário onde vai estar presente todas as ações que o usuário poderá realizar no sistema.
* *Back-end:* é a parte *Web* que fica por trás da aplicação, funciona como a ponte entre os dados que vem da aplicação e o banco de dados. Onde fica a maior parte da lógica necessária para o funcionamento do aplicativo.
* *Banco de Dados:* onde os registros são salvos e podem ser acessados, são conjuntos de dados que têm um relacionamento entre si.



**Figura 1: Modelo geral da aplicação.**

Para a solução proposta é necessário um desenvolvimento *FullStack*, ou seja, desenvolver todas as três etapas acima citadas.

Para começar o planejamento do projeto, a proposta é que o aplicativo mobile tenha essas três principais camadas de interação, então com esse modelo geral proposto, nos próximos capítulos serão vistos as tecnologias escolhidas para cada uma das três etapas e posteriormente como cada tecnologia atuou no desenvolvimento da aplicação.

1. **METODOLOGIA**

Neste capítulo serão descritas todas as tecnologias utilizadas no projeto proposto, explicando como funciona cada ferramenta e como cada uma delas é utilizada na área de desenvolvimento de software, detalhando o que é cada tecnologia e ferramenta e mostrando em que etapa e o porquê a mesma é utilizada no projeto.

Para desenvolver uma aplicação é necessário planejamento, para isso, o primeiro passo para pensar em como o projeto seria construído foi a etapa da prototipação, com isso é possível levantar os requisitos do sistema e ter uma real noção de como o aplicativo vai funcionar. Com o protótipo definido, foi então escolhida uma linguagem de programação e *framework* para o *front-end* que fizesse mais sentido para o objetivo do projeto. Por ser a parte mais visual e que entra em contato direto com o usuário, a escolha das tecnologias a serem utilizadas no *front* veio em primeiro lugar em relação ao *back-end*. A próxima etapa foi então escolher as tecnologias para compor o *back-end*, e já tendo uma escolha do *front*, nesta etapa se faz necessário pensar no que é mais adequado para a comunicação entre as partes. Por fim, foi realizada a escolha do banco de dados a ser usado no projeto, nesta etapa também é importante fazer esta escolha levando em conta as tecnologias utilizadas no *back-end*, dessa forma, foi pensado em utilizar o que tivesse uma relação mais próxima e mais simples para a comunicação entre eles.

A escolha destas tecnologias é muito importante para o desenvolvimento de uma aplicação. São escolhas que precisam ser estudadas pois precisam fazer sentido para o que é proposto no trabalho. Existem linguagens de programação que são mais adequadas para desenvolvimento *web* e outras para desenvolvimento *mobile*, por exemplo. Fazer escolhas mais adequadas para o objetivo do trabalho, faz com que o software tenha uma maior qualidade e também uma fácil manutenção no futuro, por isso foram escolhidas tecnologias e ferramentas acessíveis e que são muito utilizadas atualmente.

* 1. TECNOLOGIAS E FERRAMENTAS

Neste tópico serão descritas todas as ferramentas e tecnologias que fizeram parte de todo o ciclo de desenvolvimento da aplicação, desde a etapa de prototipação.

* + 1. **PROTOTIPAÇÃO**

A prototipação é uma etapa muito importante no desenvolvimento de um aplicativo pois auxilia o desenvolvedor na hora de levantar requisitos e no momento de desenvolver a UI da aplicação.

* + - 1. **Figma**

*Figma* é uma ferramenta de *design* de *interface online*, e que possui recursos de prototipagem. Segundo (FIGMA, 2021), os recursos de prototipagem do *Figma* permitem fazer um protótipo enquanto projeta, com recursos de protótipos animados, fazendo com que o protótipo traga uma experiência real do aplicativo.

* + 1. **FRONT-END**
       1. **Framework Flutter**

No desenvolvimento da parte visual do projeto foi utilizado o framework Flutter, que é um kit de ferramentas para desenvolvimento mobile multiplataforma (ios e android), criado pela Google no ano de 2018. Por sua otimização e rapidez, o flutter foi escolhido para o desenvolvimento deste projeto. É uma tecnologia ainda recente, mas vem cada vez mais sendo usada para o desenvolvimento de aplicações *mobile*. (FLUTTER, 2021).

* + - 1. **Linguagem de Programação Dart**

O flutter utiliza da linguagem de programação *Dart*, uma linguagem otimizada para UI, com diversos recursos para mobile. Conta com um desenvolvimento produtivo, permitindo ao desenvolvedor fazer alterações no código e ver instantaneamente com o aplicativo em execução no emulador. (DART, 2021).

* + - 1. **Android Studio**

Foi utilizado também o Android Studio, um ambiente de desenvolvimento integrado oficial para o desenvolvimento de apps Android. Possui recursos focados para aumentar a produtividade na criação de aplicação mobiles android. Para executar o aplicativo foi utilizado o Android Emulator, que simula dispositivos Android no computador, podendo desta maneira testar o aplicativo. O emulador oferece recursos de um dispositivo Android real. A vantagem de usar o emulador é que assim é mais fácil e rápido testar a aplicação do que usando um dispositivo físico, pois os dados são transferidos mais rapidamente para o emulador. (GOOGLE, 2021).

* + - 1. **Visual Studio Code**

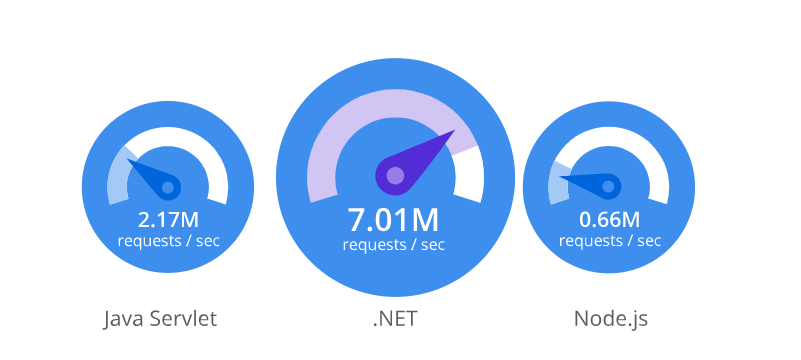
Para este trabalho, foi usado o Visual Studio Code para editor de código-fonte do front-end. O VScode foi desenvolvido pela Microsoft e foi escolhido neste projeto pois inclui suporte para depuração, versionamento de código no Git, entre outras funcionalidades que auxiliam no desenvolvimento. O VScode é um dos editores mais populares entre os desenvolvedores de software, possui a possibilidade de adicionar diversas extensões, incluindo extensões de Flutter e Dart, otimizando ainda mais o desenvolvimento da aplicação mobile. Ao usar o VScode, não é necessário iniciar o AndroidStudio no computador para usar o emulador pois o vscode conecta com os emuladores disponíveis, assim é possível iniciar o emulador no computador, a partir do próprio VScode, facilitando ainda mais e deixando o trabalho mais produtivo. (MICROSOFT, 2021).

* + 1. **BACK-END**
       1. **Linguagem de programação C#**

Segundo MICROSOFT(2021), a linguagem C# tem uma sintaxe moderna e é uma linguagem de programação orientada a objeto e orientada a componentes, permitindo ao desenvolvedor criar diversas aplicações dentro do ecossistema .NET. É uma linguagem desenvolvida pela Microsoft e que tem a sua sintaxe baseada na linguagem C. O C# possui diversos recursos que auxiliam na criação de aplicações duráveis e robustas, dando suporte a novas práticas de design de software.

* + - 1. **Plataforma .NET**

O projeto do back-end foi baseado no uso do .NET, uma plataforma de desenvolvimento de código multiplataforma criada pela Microsoft, que foi escolhida por ser gratuita e de software livre. Outro fator importante para escolha desta tecnologia é a rapidez, os aplicativos .NET fornecem melhores tempos de resposta e requerem um menor poder de computação para isso. Segundo o benchmark TechEmpower comparando estruturas de aplicativos web, o .NET saiu com um desempenho mais rápido que outras estruturas populares.(DOTNET, 2021).



**Figura 4: Comparação de desempenho de testes oficiais disponíveis na TechEmpower.**

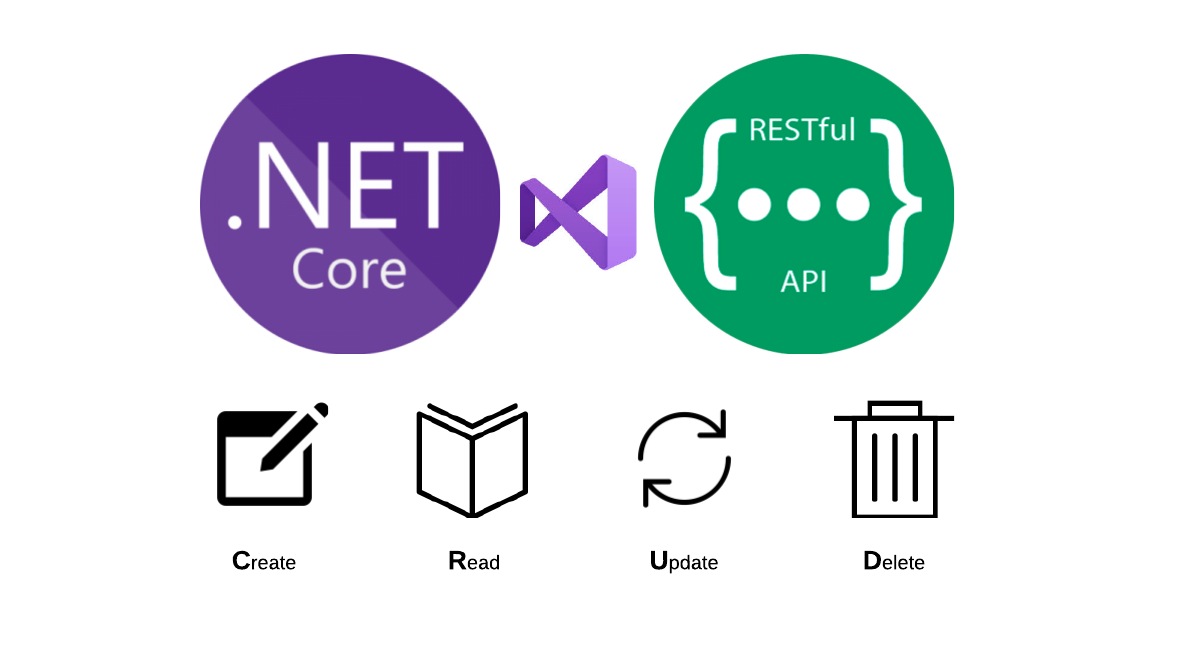
**Fonte [DOTNET, 2021]**

O .NET, suporta três linguagens de programação, sendo uma delas o C#, que foi usada no projeto deste trabalho.(MICROSOFT, 2021).

* + - 1. **ASP.NET Core**

Dentro da plataforma .NET foi usado o ASP.NET Core, uma estrutura de software livre, de alto desempenho para criar diversas aplicações, incluindo criação de web e serviços. Ideal para criar aplicativos web.

Usando o APS.NET Core MVC é possível compilar APIs Web e uma interface do usuário. Usando a arquitetura MVC é possível tornar as APIs Web testáveis. Neste trabalho foi usado ASP.NET Core para implementar APIs Web para métodos CRUD, aliados a serviços HTTP RESTful. (MICROSOFT, 2021).



**Figura 5: Modelo de metodologia usando .NET aliado a serviços RESTful para funcionalidades CRUD.**

**[WILCOCK, 2020]**

* + - 1. **API RESTful**

É uma interface de programação de aplicações que segue o padrão REST (Representational State Transfer), uma arquitetura de software que define restrições para criação de web services. (RESTFULAPI, 2020).

* + - 1. **JSON**

Segundo (JSON, 2021) o JSON(JavaScript Object Notation) é um formato de troca de dados simples entre sistemas. Um formato de texto, que possui uma coleção de pares nome/valor, formando uma lista ordenada de valores. JSON é um formato muito utilizado em desenvolvimento de software pois é uma estrutura de dados universal, onde quase todas as linguagens de programação modernas suportam seu uso. Assume a seguinte forma: um objeto é um conjunto não ordenado de pares nome / valor. Um objeto começa com { (chave esquerda) e termina com } (chave direita) . Cada nome é seguido por : (dois pontos) e os pares de nome / valor são separados por , (por vírgulas) . Podemos ver um exemplo da sintaxe de um JSON abaixo:

*{ “nome1”: “valor1”,*

*“nome2”: “valor2”,*

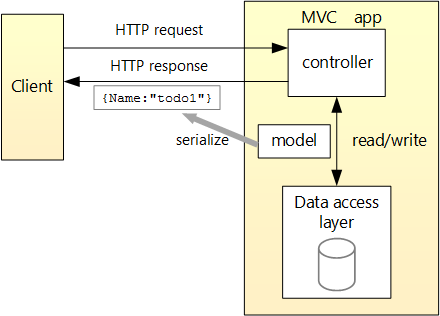
*… }*

* + - 1. **Arquitetura API Web usando ASP.NET Core**

Usando ASP.NET Core é possível criar as APIs para o funcionamento do CRUD do sistema. A tabela abaixo descreve os tipos de APIs essenciais para um sistema com CRUD básico.

|  |  |  |  |
| --- | --- | --- | --- |
| **API** | **Descrição** | **Corpo da solicitação** | **Corpo da resposta** |
| GET /api/TodoItems | Faz uma consulta de todos os itens. | Não contém | JSON |
| GET /api/TodoItems/{*id*} | Faz uma consulta de um item do *id* especificado. | *id* | JSON |
| POST /api/TodoItems | Adiciona um novo item. | JSON | Mensagem de resposta |
| PUT /api/TodoItems/{*id*} | Atualiza um item existente. | JSON | Mensagem de resposta |
| DELETE /api/TodoItems/{*id*} | Exclui um item. | *id* | Mensagem de resposta |

O diagrama que pode ser visto abaixo demonstra como funciona o design de uma aplicação API Web usando ASP.NET Core.



**Figura 6: Diagrama funcionamento de uma aplicação API Web.**

**Fonte [MICROSOFT, 2021]**

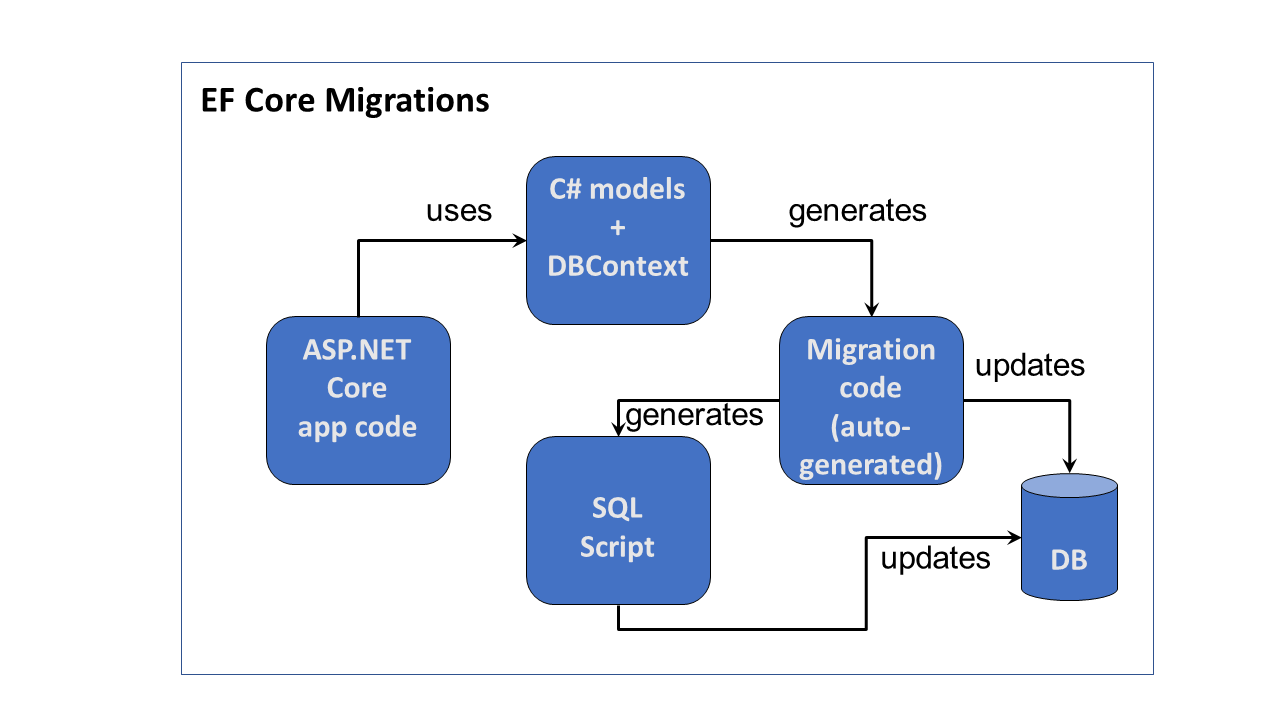
A comunicação entre o Client e a aplicação é feita através do uso de JSON. Para operações de GET por exemplo, não é necessário de um JSON para o request, mas é enviado um JSON como response, com todas as informações solicitadas pelo GET. Em operações de POST, é necessário que seja enviado um JSON no request, com todas as informações para inserir um novo item. Em uma aplicação ASP.NET Core MVC, usamos o Controller para fazer as operações de CRUD que foram citadas na tabela anterior. No controller são implementadas as funcionalidades GET, POST, PUT e DELETE, que são as APIs do sistema web. O cliente se comunica com as APIs através do método HTTP Restful, faz uma requisição que passa pelo controller e então gera uma ação no banco de dados.(MICROSOFT, 2021).

* + - 1. **Entity Framework Core**

Foi utilizado também o Entity Framework Core (EF Core), uma tecnologia de acesso a dados que permite trabalhar com banco de dados usando objetos .NET. Com o EF Core é usado um modelo para o acesso de dados, este é feito de classes de entidade de um objeto. É possível codificar manualmente um modelo para corresponder ao banco de dados, e com o modelo criado, usar as migrações do EF para criar um banco de dados baseado no modelo. O uso das migrações permite ao banco de dados evoluir conforme a mudança do modelo. (MICROSOFT, 2021).

* + - 1. **Migrations do EF**

Migrações são um recurso no EF Core, que oferece uma maneira de atualizar de forma incremental o esquema de banco de dados, deste modo mantendo em sincronia com a aplicação. Quando uma alteração é feita na aplicação, usando o recurso de migração do EF Core é possível adicionar as atualizações necessárias ao banco de dados, fazendo uma comparação do modelo atual com o antigo, gerando um arquivo da migração contendo as alterações, podendo assim acompanhar as atualizações realizadas no banco de dados. É um recurso muito útil para manter o banco de dados sempre em sincronia com o que está sendo desenvolvido na aplicação. Usar migrações facilita a atualização de qualquer tabela e relacionamento do banco de dados, com migrações é possível criar o banco de dados, baseado no modelo desenvolvido na aplicação, e por isso qualquer alteração no modelo vai ser sincronizada com o banco de dados ao usar este recurso do EF.(MICROSOFT, 2021).

**Figura 7: Fluxo de trabalho das migrações.**

**Fonte [SHAHED, 2018]**

* + - 1. **Visual Studio 2019**

A IDE(Integrated development environment) utilizada durante o projeto foi o Visual Studio, pois é uma IDE integrada da Microsoft com integração e suporte ao .NET. Possui todo o conjunto de ferramentas, desde o design até a implantação. Com ele é possível construir, depurar e testar aplicativos .NET. Possui também integração com GIT, mais um item útil para trabalhar com desenvolvimento.(MICROSOFT, 2021).

* + - 1. **Postman**

O postman é uma plataforma que auxilia no desenvolvimento de APIs, possui recursos para simplificar e otimizar cada etapa de construção de uma API. Uma ferramenta utilizada para realizar testes fáceis e rápidos em APIs e que facilita a etapa de desenvolvimento do *back-end*, tornando possível testar com maior clareza e facilidade as APIs desenvolvidas antes de ter as funcionalidades no front-end. Usando as URLs das APIs, e escolhendo o tipo de requisição (GET, POST, etc), no postman existe uma interface para trabalhar com APIs, podendo gerenciar requisições e respostas em uma só ferramenta.(POSTMAN, 2021)

* + 1. **VERSIONAMENTO DE CÓDIGO**

Usar um sistema de controle de versão de código é muito importante no desenvolvimento de software. O controle de versão permite ao desenvolvedor a rastreabilidade do código e a possibilidade de converter qualquer alteração que tenha sido feita ao código. Neste projeto foi usada a plataforma *GitHub* para hospedagem do código.

* + - 1. **Git**

Segundo (CHACON, 2014) Git é um sistema de controle de versão distribuído, é gratuito e de código aberto. O Git possui diversos recursos que o tornam vantajoso em relação a outros sistemas de controle de origem e tem sido uma das ferramentas mais utilizadas por grandes empresas como a Google, Facebook, Microsoft, Netflix entre tantas outras grandes empresas.

* + 1. **BANCO DE DADOS**
       1. **Microsoft SQL Server**

Para o banco de dados do projeto foi utilizado o Microsoft SQL Server, um sistema gerenciador de banco de dados relacional desenvolvido pela Microsoft. Neste projeto foi usada a versão 2019 do SQL Server.

Este banco de dados especificamente foi escolhido devido a facilidade de interagir com a tecnologia do .NET, por se tratar de uma tecnologia desenvolvida também pela Microsoft.

O SQL Server oferece um alto desempenho, segurança e análises avançadas, e tem compatibilidade com o Visual Studio, tornando a opção ideal para este projeto.(MICROSOFT, 2021).

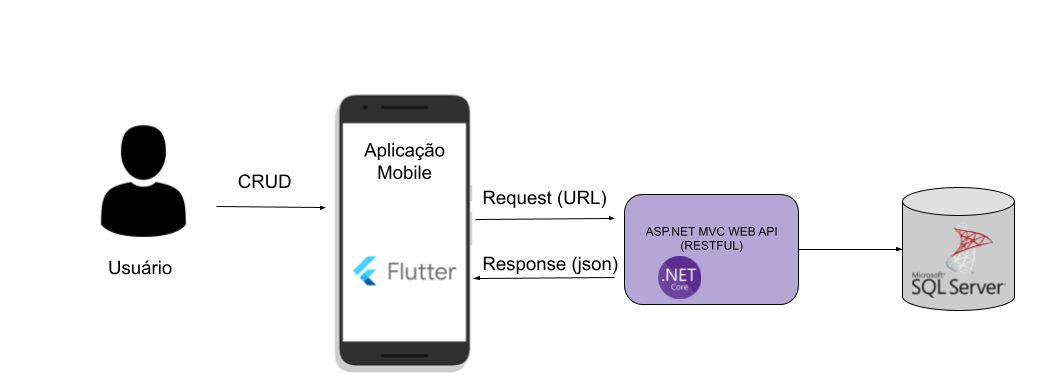
* + - 1. **SQL Server Management Studio (SSMS)**

O SQL Server Management Studio é um aplicativo de gerenciamento de banco de dados para qualquer infraestrutura SQL. Com o SSMS é possível acessar, configurar, gerenciar, administrar e desenvolver qualquer componente do SQL Server. Tem total compatibilidade com uma GUI (interface gráfica do usuário) e facilita ao usuário gerenciar seu banco de dados. Durante o projeto, tornou-se mais fácil usar querys SQL para testes no banco de dados usando esta ferramenta, foi uma ferramenta de grande apoio na fase de desenvolvimento do banco de dados e de criação do CRUD no *back-end*. (MICROSOFT, 2021).

1. **DESENVOLVIMENTO**

Neste capítulo vai ser detalhado como o sistema foi desenvolvido, demonstrando como as tecnologias citadas foram implementadas, como foi estruturada a arquitetura do projeto em todas as suas etapas e como foram aplicados os conhecimentos vistos até aqui para desenvolver a aplicação.

Após as escolhas das tecnologias e ferramentas a serem utilizadas no projeto, foi então desenvolvido um modelo para entender sobre como foi planejado o desenvolvimento do sistema. Para isso foi criado um diagrama do modelo de como o sistema funciona em uma visão geral, incluindo todas as suas etapas.



**Figura 8: Modelo do projeto.**

Em uma explicação mais ampla, o projeto funciona com um sistema de cliente-servidor. O usuário tem uma *interface* implementada em *flutter*, onde estão disponíveis as ações (CRUD) que o sistema permite que o usuário faça. Através da interface o usuário realiza essas ações, então a partir do código em linguagem *dart* são feitas requisições das APIs desenvolvidas pelo *back-end* da aplicação. Essas APIs então, através do código implementado no *back-end*, se comunicam com o banco de dados da aplicação, e assim o *back-end* faz as requisições ao banco, e através das APIs, estes dados percorrem de volta até o *front*, onde são mostrados ao usuário.

Com o fluxo do funcionamento do sistema definido, para uma boa prática de desenvolvimento de *software*, foi utilizada a metodologia *Scrum*, que traz mais organização e um desenvolvimento ágil ao projeto.

No caso deste trabalho o projeto foi desenvolvido apenas por uma pessoa, então isso foi levado em consideração no desenvolvimento de cada etapa do planejamento. Para definir como seria desenvolvido o projeto, foram realizadas cerimônias de *sprint planning*, para planejar o que seria desenvolvido em cada *sprint* do projeto. As *sprints* foram definidas para ter uma duração de duas semanas cada, mas no caso de concluir todas as tarefas da sprint podendo adiantar então tarefas de uma futura sprint. A reunião de planejamento é muito importante para definir o tempo em que as tarefas serão desenvolvidas, e ter uma estimativa de entrega do projeto. No caso deste trabalho, as reuniões de *demo*, que seriam as reuniões de entrega e demonstração das funcionalidades implementadas na *sprint*, foram realizadas junto à orientadora do trabalho, onde foram demonstrados os resultados obtidos na corrente *sprint*. Abaixo podemos ver como foram planejadas todas as *sprints* do projeto.

|  |  |
| --- | --- |
| **Número da Sprint** | **Etapa do projeto** |
| *Sprint* 1 | Desenvolvimento do protótipo e do *front-end* (telas) |
| *Sprint* 2 | Modelagem do banco de dados e criação de modelos no *back-end* |
| *Sprint* 3 | Desenvolvimento do *back-end* (construção das APIs) |
| *Sprint* 4 | Desenvolvimento do *front-end*(comunicação com as APIs) |

Para ter um bom controle de versionamento de código, durante as sprints foram feitos commits diários no Git, sempre com mensagens descritivas do que foi feito e em alguns casos criação de novas branchs para testar as novas funcionalidades sem riscos de prejudicar o código.

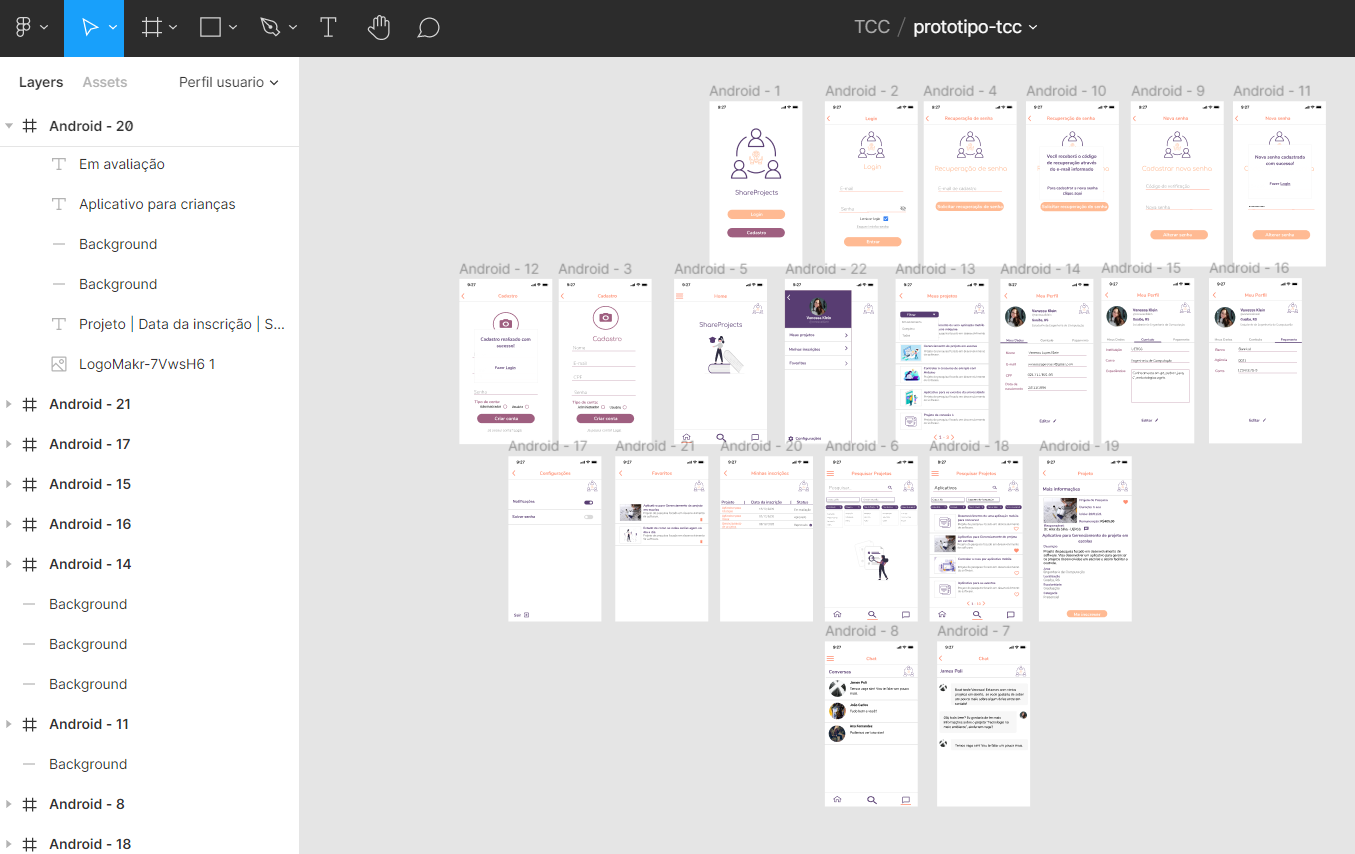
Para abordar cada etapa da aplicação em um só tópico por etapa, será explicado como cada camada da aplicação (protótipo, banco de dados, *back-end*, *front-end*) foi desenvolvida, independente da ordem das *sprints*, para desta forma concentrar todo o conteúdo da etapa em um só tópico do texto.

* 1. DESENVOLVIMENTO DO PROTÓTIPO

A prototipação é a construção de um modelo representativo do aplicativo a ser desenvolvido, e é importante pois é possível identificar botões, funcionalidades, fluxos que o usuário irá seguir e como será a aparência da aplicação. Nesta etapa é possível projetar as ideias e entender o que faz sentido ou não na aplicação, facilitando assim a etapa de desenvolvimento.

Nesta etapa do trabalho, onde foi desenvolvido o protótipo das telas e funcionalidades do aplicativo, foi usada a ferramenta *Figma*, um editor gráfico de prototipagem de projetos.

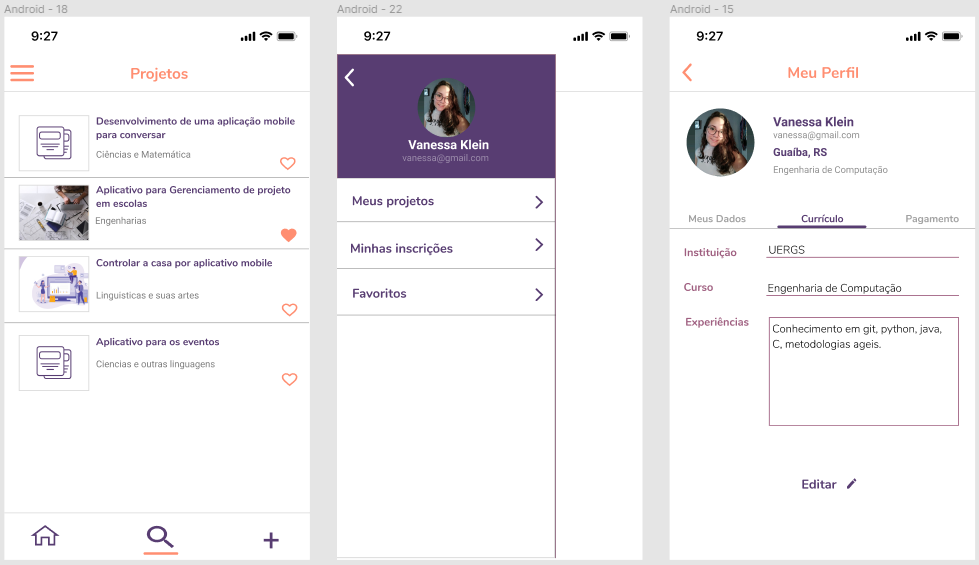
Com o Figma é possível desenvolver o *design* das interfaces do usuário e simular o fluxo das rotas entre as telas. A figura 2 apresenta a prototipação feita para este trabalho e é possível ver a aparência da ferramenta *Figma*.



**Figura 2: Prototipação do aplicativo na ferramenta Figma.**

**Fonte [autora, 2021]**

Com o Figma foi possível prototipar cada tela, definindo cada detalhe a ser usado no projeto, desde botões até o tipo de fonte para os textos. Poderia ser usado um modelo de protótipo pronto, mas com o desenvolvimento do protótipo foi possível pensar em uma interface amigável para o usuário, e ter uma interface personalizada, conforme as necessidades da aplicação. Cada tela foi pensada para um uso fácil e intuitivo do usuário.



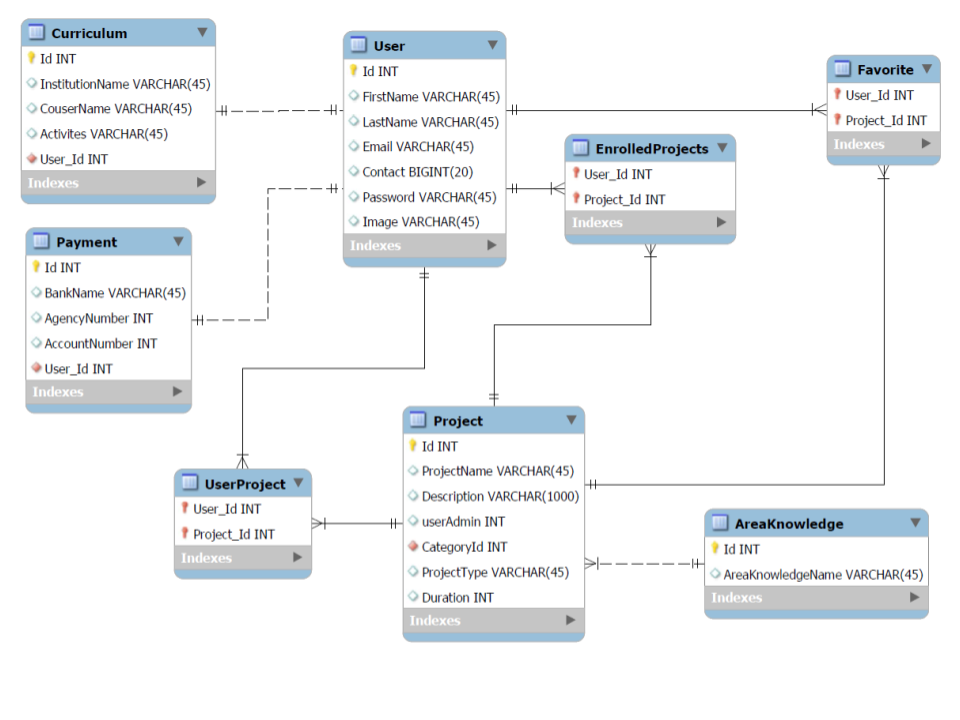
**Figura 3: Prototipação de telas na ferramenta Figma.**

**Fonte [autora, 2021]**

Com o *Figma* foi possível prototipar cada tela, definindo cada detalhe a ser usado no projeto, facilitando o entendimento dos requisitos que o sistema vai precisar para ser desenvolvido. Lembrando que o protótipo é uma etapa para facilitar o desenvolvimento do front-end, mas que não precisa necessariamente ser seguido à risca, sabendo isso durante o desenvolvimento foram feitas algumas alterações em relação ao protótipo, detalhes que durante o desenvolvimento percebeu-se que fazia mais sentido para o projeto.

* 1. MODELAGEM DO BANCO DE DADOS

Primeira etapa para começar o desenvolvimento do projeto foi fazer a modelagem do banco de dados, com um diagrama do relacionamento entre as entidades, os campos necessários e informações sobre o tipo do campo e quais seriam as PK e FK de cada tabela. É importante fazer o modelo de banco de dados para assim ter um modelo a seguir durante o desenvolvimento. Analisando o protótipo do aplicativo e as necessidades do sistema, foi então desenvolvido o seguinte diagrama ER.



**Figura 9: Diagrama entidade relacionamento do banco de dados do sistema.**

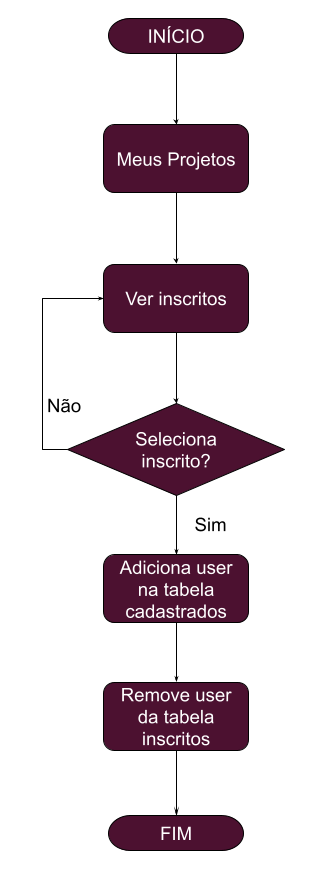
Como podemos ver no modelo de banco de dados, as duas principais entidades do projeto são Usuários e Projetos. A maior parte dos relacionamentos do sistema trata-se da conexão entre os usuários e os projetos.

O usuário possui suas informações pessoais e informações sobre currículo e pagamento, que são um tipo de relacionamento 1:1(um para um) com a tabela usuários, ou seja, cada usuário pode ter um currículo e um pagamento, assim como cada currículo e cada pagamento, pertence apenas a um único usuário.

A tabela Projetos possui suas informações do projeto e uma coluna com o *Id* da área de conhecimento do projeto (nomeada como *categoryId*), um relacionamento N:1(muitos para um) com a entidade Área de Conhecimento, onde um projeto possui apenas uma grande área de conhecimento, mas uma área de conhecimento pode pertencer a vários projetos.

Já a relação de Usuários e Projetos é mais complexa, eles têm uma relação N:N (muitos para muitos), ou seja, cada usuário pode ter vários projetos, e cada projeto pode ter vários usuários. Foi necessário modelar três tabelas para tratar o relacionamento dos usuários com os projetos. A tabela *UserProjects* é uma tabela que relaciona quais usuários fazem parte de qual projeto, ou seja, quais usuários são cadastrados num determinado projeto, incluindo o usuário que administra o projeto (campo *userAdmin Id*), ou seja que cadastrou o projeto no sistema. A segunda tabela de relacionamento de Usuário e Projeto é a de *EnrolledProjects*, que trata dos usuários que se inscreveram em determinados projetos. A terceira tabela de relacionamento entre Usuários e Projetos é a tabela *Favorite*, onde ficam as informações de quais usuários favoritaram qual projeto.

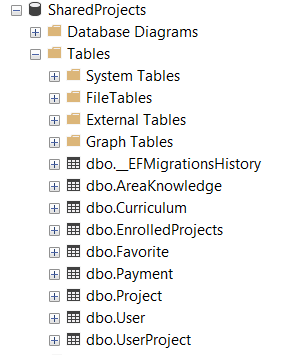
A tabela de inscritos vai poder informar ao usuário admin do projeto quantos e quais usuários estão inscritos nos seus projetos. A partir destas informações o administrador do projeto pode escolher quais inscritos quer cadastrar no seu projeto. Quando o administrador do projeto escolhe um inscrito, este sai da tabela de inscritos e passa a fazer parte da tabela de usuário cadastrados no projeto. Para entender melhor segue abaixo o fluxograma do caminho percorrido quando o administrador do projeto escolhe um usuário para seu projeto, desta forma explicando a existência de duas tabelas de relacionamento N:N entre usuários e projetos.



**Figura 10: Fluxo entre inscritos e cadastrados.**

**Fonte [autora, 2021]**

No desenvolvimento do back, com o uso de *migrations*, o banco vai ser criado a partir dos *models*, como será explicado no próximo tópico. Com o banco criado, usando o SSMS para auxiliar no desenvolvimento e na manutenção e testes do banco, foi possível obter a seguinte estrutura do banco.



**Figura 11:Estrutura de tabelas no banco de dados.**

**Fonte [autora, 2021]**

É possível notar que ao usar *migrations* o EF cria um histórico das *migrations* associado ao banco de dados, dessa forma é possível ter um controle de todas as alterações feitas no banco ao decorrer do tempo, garantindo uma opção de *rollback* quando necessário.

* 1. DESENVOLVIMENTO BACK-END

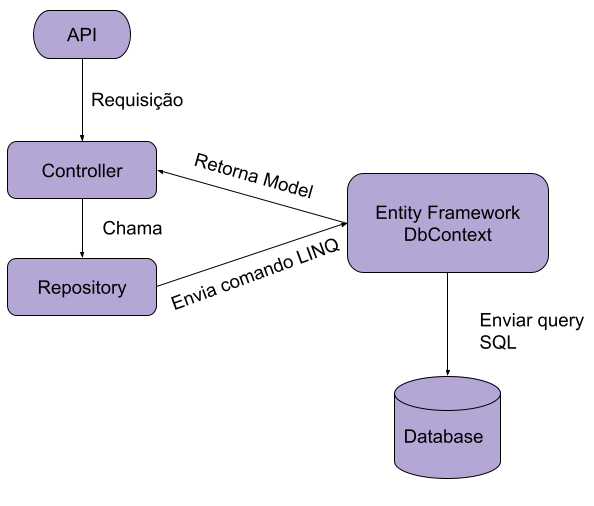
Com o modelo do banco de dados definido, é possível então planejar o desenvolvimento do *back-end*. A partir da escolha de usar *Migrations* para criar e alterar o banco de dados do projeto, a próxima etapa do desenvolvimento do sistema foi definir a arquitetura a ser usada no projeto de backend. Para isso é importante entender que para a conexão e fluxo de dados entre o *backend* e o *frontend*, foram implementadas APIs.

A API tem a finalidade de trocar dados entre diferentes sistemas, e no caso deste trabalho funciona como a “ponte” entre as informações do *front-end* e do *back-end*. As APIs foram implementadas utilizando a tecnologia ASP.NET Core com a arquitetura MVC. Através do *Controller* é feita a intermediação entre as requisições enviados pelo *front* com as respostas que o *Model* do *back-end* fornece, ou seja, processa os dados informados pelo usuário e repassa para as outras camadas da aplicação, resumindo o *model* recebe as informações do *controller*, então valida a requisição e retorna uma resposta.

Como neste trabalho foi usado o EF e as *Migrations*, a implementação dos modelos reflete no banco de dados do sistema. Quando uma *Migration* roda, ela se baseia nos modelos para criar as tabelas no banco de dados, por isso é importante ter os modelos bem definidos na hora de desenvolver o sistema do *back-end*.

* + 1. **Arquitetura e implementação do Back-end**

A arquitetura do back-end foi definida baseada na arquitetura para API. A imagem a seguir mostra como foi definida a arquitetura do sistema do *back-end*.

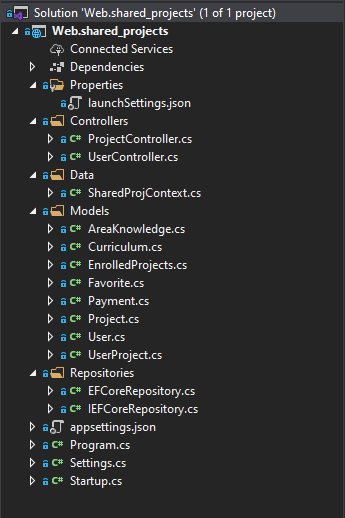


**Figura 12: Diagrama da arquitetura do sistema do back-end.**

**Fonte [autora, 2021]**

O sistema funciona conforme demonstrado na imagem, onde o *front* faz uma requisição através da API, esta requisição chega no *controller* que então chama uma função no *repository*, onde ficam todas as funções chamadas pelos *controllers*, o *repository* então envia um comando LINQ através do EF Dbcontext que então envia a *query* SQL pro banco, realizando a operação requisitada (CRUD).

Todo o *back-end* foi desenvolvido usando o Visual Studio pois esta IDE possui recursos que auxiliam no desenvolvimento com .NET, facilitando tanto o desenvolvimento da aplicação como também a visualização dos componentes do projeto do *back-end*.A partir dos conhecimentos adquiridos durante o estudo do trabalho e definidas as tecnologias e a arquitetura assumida pelo projeto, foram implementadas as camadas da arquitetura conforme pode ser visto na figura abaixo, que mostra a estrutura de pasta do projeto back-end desenvolvido.

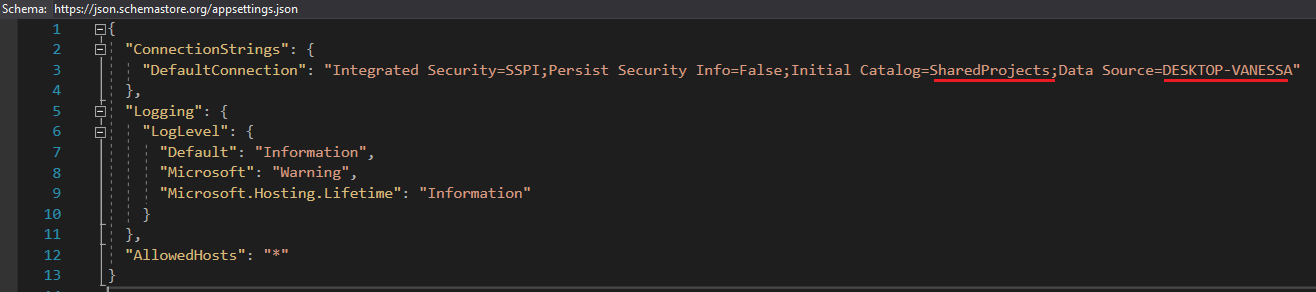


**Figura 13: Estrutura de pasta do projeto do back-end.**

**Fonte [autora, 2021]**

Primeiramente temos a pasta *Properties*, onde contém o arquivo que define as configurações de inicialização da aplicação. Outros arquivos como *Program*, *Startup*, *Settings*, também são arquivos onde contém informações e configurações da aplicação e é onde é desenvolvido o programa que inicializa o projeto e que chama as outras camadas.

É muito importante configurar o arquivo *appsettings* que é onde fica a *string* de conexão com o banco de dados, que vai ser por onde as *migration* vão identificar qual *Data Source* do banco deve conectar e qual o nome do banco de dados a ser criado.



**Figura 14: Arquivo appsettings.json de configuração da string de conexão com o banco.**

**Fonte [autora, 2021]**

O próximo passo foi implementar os *models* da aplicação. Como já falado aqui, os *models,* ao usar as *migrations*, precisam ser desenvolvidos conforme a necessidade das tabelas para o banco de dados, pois as tabelas do banco de dados serão o espelho dos modelos implementados do *backend*. Como o banco de dados já foi modelado, foram então implementados os *models* de cada tabela usando o modelo do banco com suas respectivas colunas e tipos de dados.

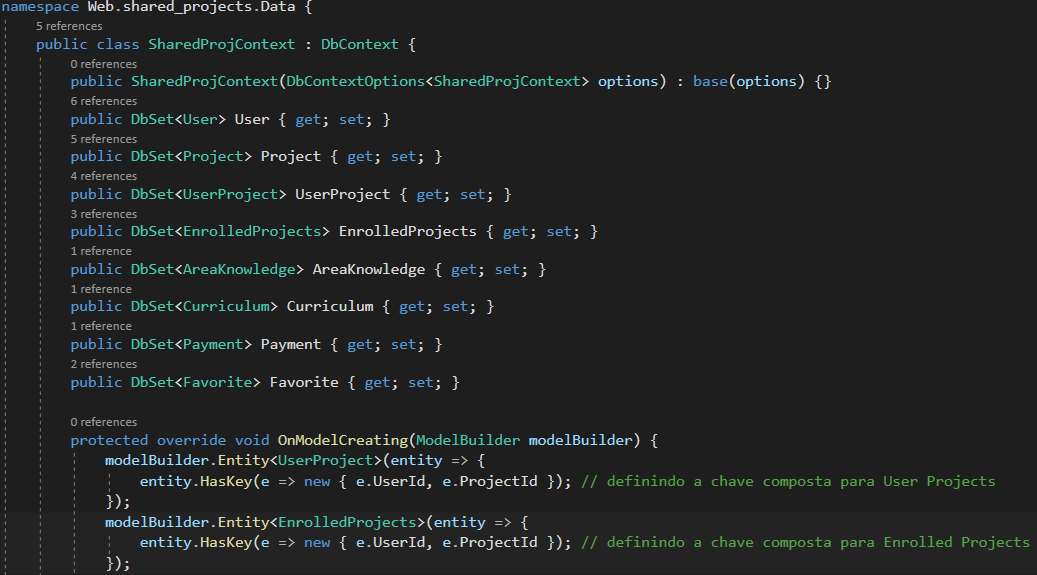


**Figura 15: À esquerda o código para implementação do Model User. A direita a tabela user no banco de dados.**

**Fonte [autora, 2021]**

Pode-se perceber que os dois itens *ICollection*, que são itens que retornam uma lista de dados e não apenas um único dado, são itens que não aparecem na tabela, isso acontece pois esses itens são referentes ao relacionamento N:N que existe entre as tabelas *User* e *Projects*. Com o relacionamento N:N se faz necessário criar uma nova tabela intermediária, logo essa nova tabela, intitulada *UserProjects*, tem que ser referenciada tanto no *model* do *User* quanto no *model* de *Projects*. O mesmo acontece nas tabelas *EnrolledProjects* e *Favorite*.

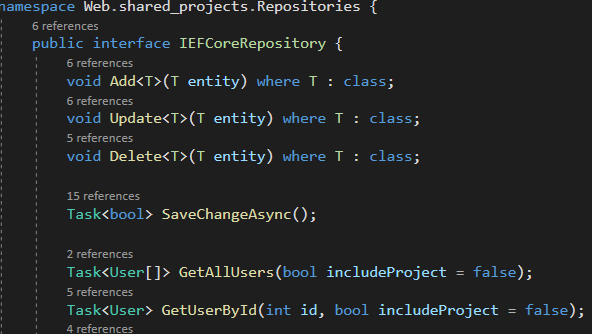
A partir dos *models* implementados, é preciso referenciá-los no arquivo de contexto, que fica na pasta *Data*, este arquivo que referencia o *DbContext* que faz com que o EF faça as requisições no banco de dados. É no arquivo de contexto do projeto que também é definida as chaves compostas que existem em uma tabela, onde é implementada a lógica para formar a chave composta e indicar ao banco de dados o relacionamento entre as tabelas.



**Figura 16: Código da implementação do arquivo de contexto do projeto de backend.**

**Fonte [autora, 2021]**

Contudo, a próxima etapa no desenvolvimento foi implementar os arquivos de *repository*. São dois arquivos, onde um é do tipo *Interface*, onde são declarados os métodos que pertencem ao arquivo de *repository*. O outro arquivo é onde são implementadas as lógicas das requisições ao banco, onde ficam as requisiçõe *LINQs* do sistema e as operações de *Add*, *Update* e *Delete*.

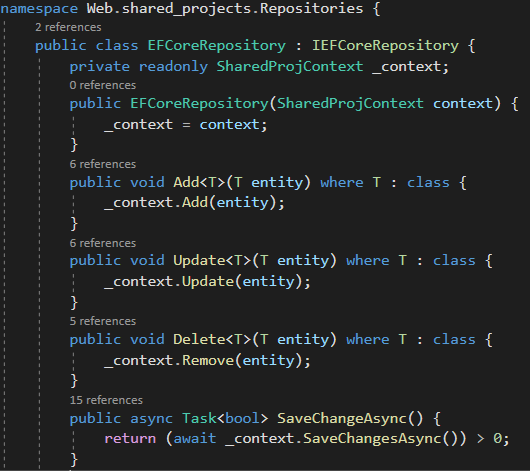


**Figura 17: Trecho de código da implementação da interface do repository do projeto de backend.**

**Fonte [autora, 2021]**

As instruções de *Add*, *Update* e *Delete* e os outros métodos com as operações LINQ com o *GetAllUsers*(Faz um *select* em todos os usuários)e o *GetUserById*(faz um *select* do usuário pelo *Id*)como é possível ver na imagem acima, são todas instruções que são chamadas nos *controllers* do projeto, como demonstra a imagem 12.

Agora com o arquivo de Interface implementado, foi desenvolvido outro arquivo repository com os métodos que realizam as operações solicitadas.

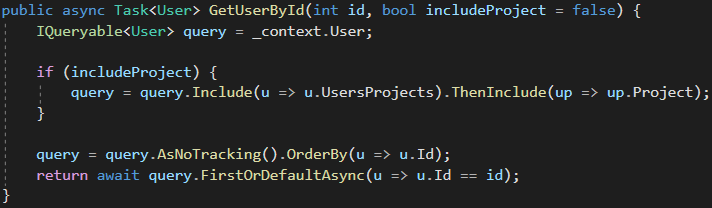


**Figura 18: Trecho de código da implementação do repository do projeto de backend.**

**Fonte [autora, 2021]**

No começo do código é instanciado o contexto do projeto, que pode ser visto na figura 16, para que as instruções LINQ e as operações identifiquem os *models* e através do *DbContext* as requisições são passadas ao banco. A implementação das operações *Add*, *Update* e *Delete* no *repository* aceitam qualquer tipo como parâmetro, facilitando a implementação e evitando repetir código. O método SaveChanges também é declarado no *repository* com a mesma intenção de reaproveitamento de código, mantendo uma boa organização e a cultura de código limpo.

A seguir, outro trecho do mesmo arquivo do *repositoy*, onde mostra o método *GetUserById*.



**Figura 19: Trecho de código do repository do método GetUserById.**

**Fonte [autora, 2021]**

Na imagem mostra como é montada uma *query* usando LINQ. A opção *incluiProject* pode ser opcional caso queira que a resposta da *query* retorne o usuário e os projetos em que o usuário está cadastrado, ou podendo retornar apenas as informações do usuário. Foram implementados diversos métodos conforme a necessidade que o sistema teve. Todos os métodos implementados no *repository* para requisições de *query* SQL usando LINQ estão descritos na tabela a seguir.

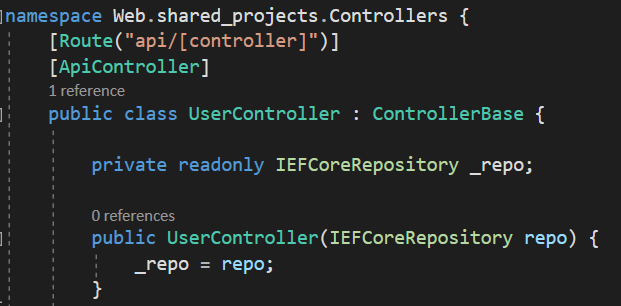
|  |  |  |  |
| --- | --- | --- | --- |
| **Método** | **Parâmetro** | **Retorno** | **Objetivo** |
| *GetAllUsers* | opcional : includeProject (tipo *bool*), por *default* é igual a *false.* | Uma lista de todos os usuários. | Retornar todos os usuários do sistema. |
| *GetUserById* | obrigatório: *id* (número do *id* do usuário).  opcional : includeProject (tipo *bool*), por *default* é igual a *false.* | Um único usuário. | Retorna o usuário do *id* solicitado.  Usado nos *controllers* para as APIs de *Get*, *Put*, *Delete* para identificar em qual usuário a operação vai ser feita. |
| *GetUserByEmail* | obrigatório: *email* (*email* do usuário).  opcional : includeProject (tipo *bool*), por *default* é igual a *false.* | Um único usuário. | Retorna o usuário com o email solicitado.  Usado para verificar na hora do cadastro de usuário se o email já existe ou não no sistema. |
| *GetAllProjects* | opcional : includeProject (tipo *bool*), por *default* é igual a *false.* | Uma lista de todos os projetos. | Retornar todos os projetos do sistema.  Usado para página de projetos do aplicativo, onde retorna todos os projetos. |
| *GetProjectById* | obrigatório: *id* (número do *id* do projeto).  opcional : includeProject (tipo *bool*), por *default* é igual a *false.* | Um único projeto. | Retorna o projeto do *id* solicitado.  Usado nos *controllers* para as APIs de *Get*, *Put*, *Delete* para identificar em qual projeto a operação vai ser feita. |
| *GetUserProj* | obrigatório: *id* do usuário. | Uma lista de projetos. | Retorna uma lista de todos os projetos em que um usuário com o *id* solicitado está cadastrado.  Usado para a página de ‘Meus Projetos’. |
| *GetEnrollUserProj* | obrigatório: *id* do usuário. | Uma lista de projetos. | Retorna uma lista de todos os projetos em que um usuário com o *id* solicitado está inscrito.  Usado para página de ‘Minhas inscrições’. |
| *GetUsersEnrollsinProjects* | obrigatório: *id* do projeto. | Uma lista de usuários. | Retorna uma lista de usuários que estão inscritos em um determinado projeto com o id solicitado.  Usado para a página ‘Ver inscritos’, para ver os inscritos em um determinado projeto. |
| *GetUsersinProjects* | obrigatório: *id* do projeto. | Uma lista de usuários. | Retorna uma lista de usuários que estão cadastrados em um determinado projeto com o id solicitado.  Usado para poder ver todos os usuários cadastrados em um determinado projeto. |
| *GetAllAreas* | Nenhum. | Uma lista de Áreas de Conhecimento. | Retorna a uma lista de todas as grandes áreas de conhecimento do sistema.  Usado para a página de cadastro de projeto, na de editar projeto e na de ver os detalhes do projeto, para mostrar nas telas de cadastro e editar quais as opções de áreas disponíveis. |
| *GetEnrollUserProj* | obrigatório: *id* do usuário e *id* do projeto. | Um item da tabela *EnrolledProjects.* | Retorna um item da tabela *EnrolledProjects* onde o id do usuário e o id do projeto solicitados coincida com o do item.  Usado para deletar uma inscrição na tabela *EnrolledProjects*. No *controller*, vai fazer uma requisição neste método, vai retornar uma inscrição e em seguida deletar essa inscrição da tabela. |
| *GetProjeByUserIdProj* | obrigatório: *id* do usuário e *id* do projeto. | Um item da tabela *UserProjects.* | Retorna um item da tabela *UserProjects* onde o id do usuário e o id do projeto solicitados coincida com o do item.  Usado para ver se o usuário do id solicitado está cadastrado no projeto do id solicitado. |
| *GetUsersByIdProj* | obrigatório: *id* do projeto. | Uma lista de itens da tabela *UserProjects.* | Retorna uma lista de itens da tabela *UserProjects* onde o *id*  do projeto é o *id* solicitado.  Usado para pegar todos os cadastros que existem para um determinado projeto, para verificar quais usuários estão cadastrados em um determinado projeto. |
| *GetFavoritesProj* | obrigatório: *id* do usuário e *id* do projeto. | Um item da tabela *Favorite.* | Retorna um item da tabela *Favorite*.  Usado para verificar se determinado projeto para determinado usuário é um dos favoritos ou não. Na tela de projetos usado para verificar o estado do botão de favoritar projeto. |
| *GetAllFavoritesProjByuser* | obrigatório: *id* do usuário | Uma lista de itens da tabela *Favorite*. | Retorna uma lista de itens da tabela *Favorite*.  Usado para pegar todos os projetos favoritos de um determinado usuário. Para a tela de ‘Meus Favoritos’. |
| *GetCurriculumByUserId* | obrigatório: *id* do usuário | Um único currículo. | Retorna o currículo do usuário com o *id* solicitado.  Usado na tela ‘Meu perfil’ onde consta o currículo do usuário. |
| *GetPaymentByUserId* | obrigatório: *id* do usuário | Um único item de pagamento. | Retorna o item de pagamento do usuário com o *id* solicitado.  Usado na tela ‘Meu perfil’ onde consta os dados de pagamento do usuário. |

Os métodos no *repository* foram criados, com todas as requisições SQL para o banco de dados, com isso agora é possível criar os *controllers*, que é onde acontece todo o controle das informações que transitam no sistema.

Algumas entidades, pelo fato de ter poucas requisições ao banco, como currículo e pagamento por exemplo, que tem as operações de *Add* e *Update* somente, foram requisições implementadas no controller de Usuário, pois são informações relacionadas ao usuário, deste modo centralizado as informações num só controller e evitar ter que desenvolver um *controller* para apenas dua operações.

Deste modo, foram criados dois *Crontollers*, o de Usuário e o de Projetos. Nos controllers são implementados os métodos HTTP do tipo *Get* (*Select*/*Read*), *Post* (*Insert*/*Create*), *Put* (*Update*) e *Delete*.

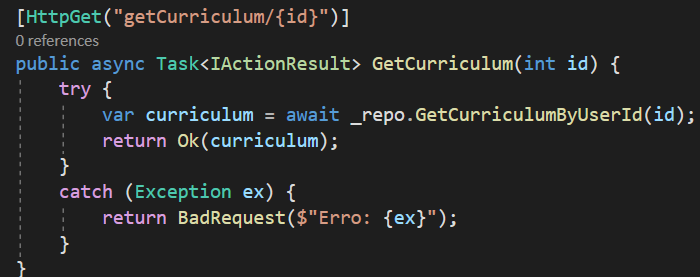
Quando um controller com a arquitetura API é implementado, por padrão a rota de acesso a URL dessa api é “*api/[controller]*”. Em seguida, é necessário instanciar o *repository* dentro do controller para que seja possível fazer as chamadas dos método que foram implementados no *repository*, que realizam as operações SQL.



**Figura 20: Trecho do código inicial do controller da entidade User.**

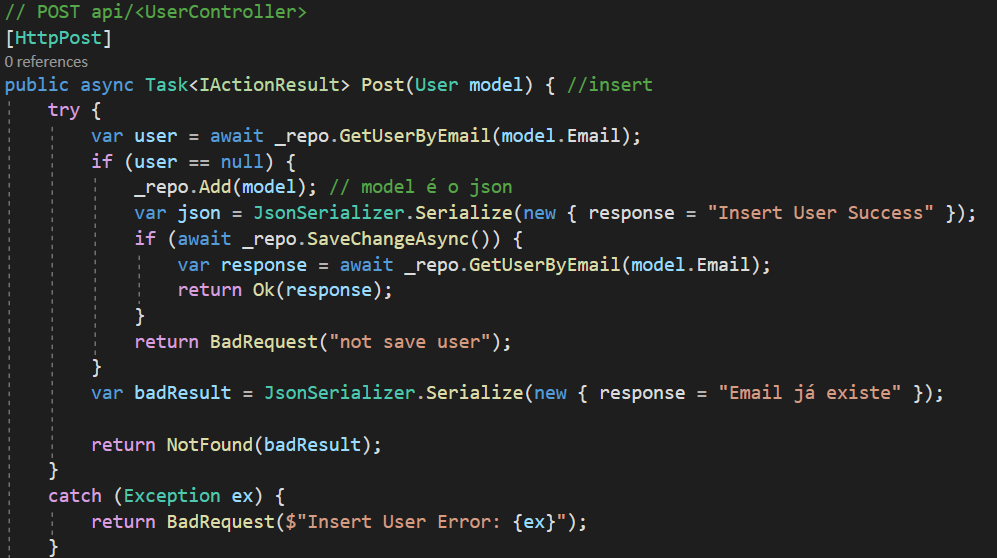
**Fonte [autora, 2021]**

Em seguida foram implementados os métodos HTTP conforme a necessidade do sistema.



**Figura 21: Trecho de código do método HTTP Get para Currículo.**

**Fonte [autora, 2021]**

****

**Figura 22: Trecho de código do método HTTP Post para usuário.**

**Fonte [autora, 2021]**

Como mostra a figura 21 de exemplo de um método HTTP, é possível adicionar um nome à URL (“*getCurriculum*”) e identificar que parâmetro vai ser recebido pela API (id), podendo também não ter um parâmetro. Em seguida é implementado uma estrutura *try/catch* para que o código tenha tratamento de erros. O método HTTP então faz uma chamada ao método *GetCurriculumByUserId*, passando o *id* recebido na API como parâmetro e retorna a consulta feita no banco de dados. Em caso de sucesso retorna uma mensagem com o *status Ok (status code 200)*, em caso de erro retorna uma mensagem de erro com o *status* de *BadRequest (status code 400)*.

No exemplo da figura 22 é implementado um método HTTP *Post*, que diferente do exemplo anterior, recebe um *Model* do tipo *User*. O *model* é no formato JSON.

O método Post deste caso é de inserção de usuário no sistema, logo é chamado o método *GetUserByEmail* para verificar se o email informado no *model* é um email que já existe cadastrado ou não. Se não existir, a variável *user* retornará com o valor *null* e então entra no *if* de verificação, onde possui a lógica para adicionar um registro no banco de dados através do método *Add*.

Segue na tabela abaixo todos os métodos HTTP implementados neste trabalho e como funciona cada um deles.

1. **Métodos HTTP do UserController:**

|  |  |  |  |
| --- | --- | --- | --- |
| **API** | **Descrição** | **Parâmetro da Requisição** | **Resposta** |
| GET  api/user | *Select* detodos os dados da tabela *User*. | Nenhum. | JSON com todos os usuários. |
| GET  api/user/{id} | *Select* do usuário com o *id* solicitado. | *id* | JSON com os dados do usuário. |
| GET  api/user/usersProj/{id} | *Select* de todos os registros da tabela *UserProjects* com o projeto do *id* informado. | *id* | JSON com todos os registros da tabela *UserProjects* com o projeto do *id* solicitado. |
| GET  api/user/getUsersSubProjects/{id} | *Select* de todos os usuários cadastrados no projeto com *id* informado. | *id* | JSON com todos os usuários cadastrados no projeto com *id* solicitado. |
| GET  api/user/getCurriculum/{id} | *Select* do currículo do usuário com o *id* informado. | *id* | JSON com o currículo do usuário do *id* solicitado. |
| GET  api/user/getPayment/{id} | *Select* do item de pagamento do usuário com o *id* informado. | *id* | JSON com o item selecionado de pagamento do usuário do *id* solicitado. |
| POST  api/user | *Insert* dos dados do usuário enviados no *model*. | *model* (JSON) | JSON com a mensagem de resposta. |
| POST  api/user/upsertCurriculum | *Insert* e *Update* dos dados do currículo enviado no *model*. | *model* (JSON) | Mensagem de resposta. |
| POST  api/user/upsertPayment | *Insert* e *Update* dos dados do pagamento enviado no *model*. | *model* (JSON) | Mensagem de resposta. |
| POST  api/user/Login | *Select* do usuário com o *email* informado no *model*. | *model* (JSON) | JSON com os dados do usuário. |
| PUT  api/user/{id} | *Select* do usuário com o *id* informado e *Update* desse usuário com o *model* informado. | *id* e *model* (JSON) | Mensagem de resposta. |
| DELETE  api/user/{id} | *Delete* do usuário com o *id* informado. | *id* | Mensagem de resposta. |

1. **Métodos HTTP do ProjectController:**

|  |  |  |  |
| --- | --- | --- | --- |
| **API** | **Descrição** | **Parâmetro da Requisição** | **Resposta** |
| GET  api/project | *Select* detodos os dados da tabela *Project*. | Nenhum. | JSON com todos os projetos. |
| GET  api/project/{id} | *Select* do projeto com o *id* solicitado. | *id* | JSON com os dados do projeto. |
| GET  api/project/User/{id} | *Select* dos projetos em que o *user* com o *id* informado está cadastrado. | *id* | JSON com todos os projetos do usuário com o *id* informado. |
| GET  api/project/getEnrollUsers/{id} | *Select* dos usuários inscritos no projeto com o *id* informado. | *id* | JSON com todos os usuários inscritos no projeto com o *id* informado. |
| GET  api/project/getEnroll/{id} | *Select* dos projetos em que o usuário do *id* informado está escrito. | *id* | JSON com os projetos que o usuário com o *id* informado está inscrito. |
| POST  api/project | *Insert* dos dados do projeto enviados no *model*. | *model* (JSON) | JSON com mensagem de resposta. |
| PUT  api/project/{id} | *Select* do projeto com o *id* informado e *Update* desse projeto com o *model* informado. | *id* e *model* (JSON) | JSON com mensagem de resposta. |
| POST  api/project/getUsersProjects | *Select* do registro na tabela *UserProjects* pelos *id* de usuário e *id* de projeto informados no *model.* | *model* (JSON) | JSON com o registro selecionado da tabela *UserProject*. |
| POST  api/project/getEnrollProj | *Select* do registro na tabela *EnrolledProjects* pelos *id* de usuário e *id* de projeto informados no *model*. | *model* (JSON) | JSON com o registro da tabela *EnrolledProjects.* |
| POST  api/project/getFavoritesProj | *Select* do registro da tabela *Favorite* onde o usuário tem o *id* informado e o projeto tem o *id* de projeto informados no *model*. | *model* (JSON) | JSON com o registro selecionado da tabela *Favorite*. |
| GET  api/project/getAllFavoritesByUserProj/{id} | *Select* todos os registros da tabela *Favorite* onde o usuário tem o *id* informado. | *id* | JSON com os registros selecionados da tabela *Favorite*. |
| POST  api/project/addFavoritesProj | *Insert* na tabela *Favorite* o *model* com as informações do registro de favoritos. | *model* (JSON) | JSON com mensagem de resposta. |
| POST  api/project/removeFavoritesProj | *Select* do registro da tabela *Favorite* onde o usuário tem o *id* informado e o projeto tem o *id* de projeto informados no *model*. | *model* (JSON) | JSON com mensagem de resposta. |
| POST  api/project/deleteEnroll | *Select* do registro na tabela *EnrolledProjects* pelos *id* de usuário e *id* de projeto informados no *model* e *Delete* deste registro. | *model* (JSON) | Mensagem de resposta. |
| DELETE  api/project/{id} | *Delete* do projeto com o *id* informado. | *id* | Mensagem de resposta. |

Como pode-se notar nas tabelas dos métodos HTTP dos *controllers* do sistema, alguns métodos com o objetivo de realizar uma operação de *delete* ou de *get* por exemplo, usaram o método *Post*, isso acontece no caso de precisar de um *model* para realizar a requisição, então não é possível implementar o método *Get* ou *Delete*, e então é preciso criar um método *Post* mas que na sua implementação faça uma requisição do método *delete* ou *get*, conforme for o caso. Para entender melhor o funcionamento de cada API, segue abaixo uma tabela da relação entre as APIs implementadas e os métodos do *repository* que cada API usa. Com o auxílio das tabelas anteriores e a tabela a seguir é possível entender o funcionamento de cada API especificamente.

|  |  |  |  |
| --- | --- | --- | --- |
| **Controller** | **API** | **Método do *repository* para consulta dos dados** | **Método do *repository* que realiza a ação requisitada pela API** |
| USER | GET  api/user | GetAllUsers | GetAllUsers |
| USER | GET  api/user/{id} | GetUserById | GetUserById |
| USER | GET  api/user/usersProj/{id} | GetUsersByIdProj | GetUsersByIdProj |
| USER | GET  api/user/getUsersSubProjects/{id} | GetUsersinProjects | GetUsersinProjects |
| USER | GET  api/user/getCurriculum/{id} | GetCurriculumByUserId | GetCurriculumByUserId |
| USER | GET  api/user/getPayment/{id} | GetPaymentByUserId | GetPaymentByUserId |
| USER | POST  api/user | GetUserByEmail | Add |
| USER | POST  api/user/upsertCurriculum | Não consta | Add ou Update |
| USER | POST  api/user/upsertPayment | Não consta | Add ou Update |
| USER | POST  api/user/Login | GetUserByEmail | GetUserByEmail |
| USER | PUT  api/user/{id} | GetUserById | Update |
| USER | DELETE  api/user/{id} | GetUserById | Delete |
| PROJECT | GET  api/project | GetAllProjects | GetAllProjects |
| PROJECT | GET  api/project/{id} | GetProjectById | GetProjectById |
| PROJECT | GET  api/project/User/{id} | GetUserProj | GetUserProj |
| PROJECT | GET  api/project/getEnrollUsers/{id} | GetUsersEnrollsinProjects | GetUsersEnrollsinProjects |
| PROJECT | GET  api/project/getEnroll/{id} | GetEnrollUserProj | GetEnrollUserProj |
| PROJECT | POST  api/project | Não consta | Add |
| PROJECT | PUT  api/project/{id} | GetProjectById | Update |
| PROJECT | POST  api/project/getUsersProjects | GetProjeByUserIdProj | GetProjeByUserIdProj |
| PROJECT | POST  api/project/getEnrollProj | GetEnrollUserProj | GetEnrollUserProj |
| PROJECT | POST  api/project/getFavoritesProj | GetFavoritesProj | GetFavoritesProj |
| PROJECT | GET  api/project/getAllFavoritesByUserProj/{id} | GetAllFavoritesProjByuser | GetAllFavoritesProjByuser |
| PROJECT | POST  api/project/addFavoritesProj | Não consta | Add |
| PROJECT | POST  api/project/removeFavoritesProj | GetFavoritesProj | Delete |
| PROJECT | POST  api/project/deleteEnroll | GetEnrollUserProj | Delete |
| PROJECT | DELETE  api/project/{id} | GetProjectById | Delete |

Todas as APIs e métodos de requisição a operações foram implementados, a próxima etapa então é testar as APIs, demonstrar como testar, as ferramentas utilizadas e o retorno da API.

Como já mostrado, a API recebe um JSON no corpo de requisição ou um número inteiro referente ao *id* solicitado, que é inserido na própria URL da API. Em algumas APIs, o retorno também é em formato JSON.

Abaixo segue exemplos de um JSON de *request* e um de *response*, respectivamente.

JSON de requisição para inserir ou atualizar os dados de um usuário:

{

"id": 7,

"firstname":"Inajara",

"lastName": "Lopes Figueira",

"email": "jo.silva@gmail.com",

"contact": "5144879653",

"password": "123456"

}

JSON de resposta de uma requisição *Get* no usuário com *id* = 6:

{

"id": 6,

"firstName": "Fernando",

"lastName": "Gomes",

"email": "fernando@e.com",

"contact": 22222222,

"usersProjects": [],

"enrolledProjects": **null**,

"password": "123456",

"image": "assets/img/undefined.png"

}

Os testes realizados para verificar o funcionamento do back-end serão demonstrados em uma seção mais à frente neste trabalho.

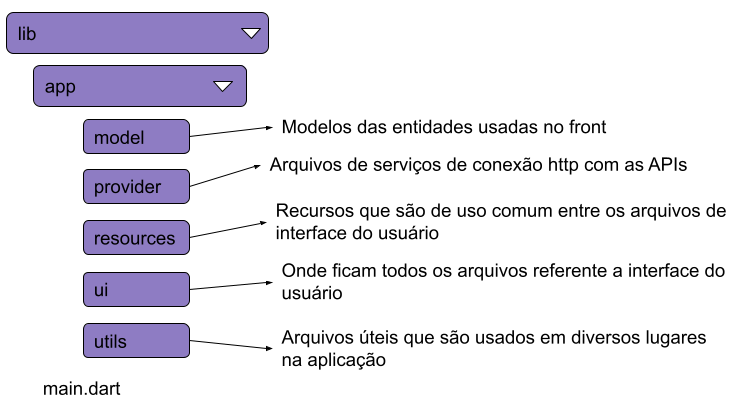
* 1. DESENVOLVIMENTO FRONT-END

A primeira etapa do desenvolvimento do *front-end* foi a implementação das telas e a definição das rotas que as telas percorrem no aplicativo. Nesta etapa foi utilizado o protótipo desenvolvido no começo da etapa de desenvolvimento como modelo para a criação das telas, desenvolvidas através do código na linguagem *Dart* com o *framework Flutter.* O protótipo serviu de base para o desenvolvimento do *design* de todo o *app*. Na implementação do *front-end* é muito importante pensar no desenvolvimento de uma interface que seja amigável e intuitiva ao usuário. Por isso é muito importante os estudos de usabilidades e aplicações de metodologias de UI/UX nesta etapa do desenvolvimento.

* + 1. **Arquitetura do projeto front-end**

É de suma importância também, ter uma arquitetura bem definida para ser usada no projeto, e no front-end, a arquitetura desenvolvida para o projeto foi baseada na arquitetura *GetX pattern*.

A partir das necessidades do sistema, foi definida a seguinte arquitetura para ser usada no projeto do *fron-end*:



**Figura 23: Arquitetura do projeto do fron-end.**

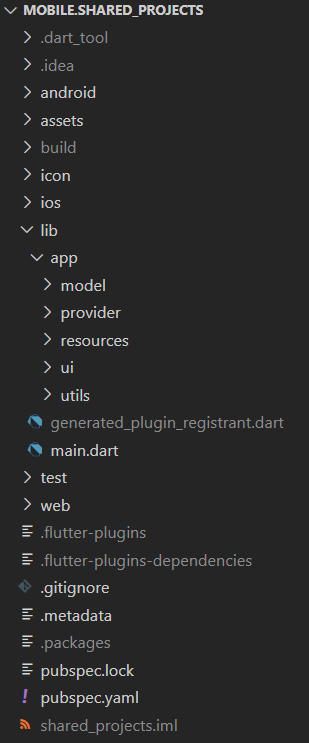
**Fonte [autora, 2021]**

A pasta *lib* é a pasta de referência no projeto *flutter*. Quando um projeto *flutter* é criado, uma série de pastas e arquivos padrões do *flutter* são criados. Dentro da pasta *lib* fica o arquivo *main.dart*, o arquivo principal que faz o projeto rodar e onde fica a chamada da primeira tela da aplicação.

Foi criada uma pasta uma pasta *app*, onde ficam todos os outros arquivos da aplicação *front-end*. Dentro desta pasta *app* é onde foi implementada a arquitetura baseada na arquitetura *GetX pattern,* utilizando de alguns conceitos vistos neste tipo de arquitetura*.*

Como explicado na figura, temos dentro da pasta *app* as pastas *model*, que se trata dos modelos referentes às entidades da aplicação, a pasta *provider* que contém os serviços que fazem as chamadas das APIs, a pasta *resources* que contém recursos de uso comum entre vários arquivos do projeto, a pasta *ui* que contém tudo relacionado a parte visual do sistema *front-end* e a pasta *utils* que foi criada para armazenar itens que são usadas em diversos momentos no código.

Com a arquitetura bem definida e através do uso do VSCODE, foi então criado o projeto flutter. A estrutura de pastas do projeto do front-end pode ser vista na imagem a seguir.



**Figura 24: Estrutura de pastas do projeto do fron-end.**

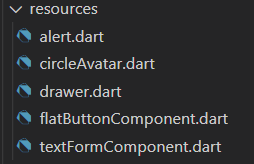
**Fonte [autora, 2021]**

* + 1. **Desenvolvimento das telas e rotas**

Para o desenvolvimento das telas foi criada uma pasta intitulada *ui*, onde ficam todos arquivos relacionados a interface com o usuário. Essa estrutura teve referência na arquitetura *GetX pattern,* como já mencionado.

Essa organização de pastas é muito importante para manter uma implementação de código limpo e manter sempre a organização dos arquivos, facilitando para que outros desenvolvedores entendam o código implementado.

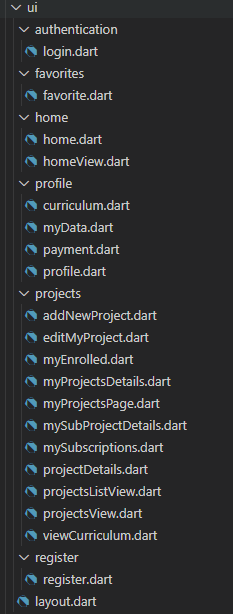
No desenvolvimento das telas e rotas foi também a etapa em que a pasta *resources* foi criada, pasta essa que possui vários recursos compartilhados, como *widgets* por exemplo. Nesta pasta foram criados os componentes mais usados durante todo o projeto, para facilitar a criação de widgets nas telas e fazer o aproveitamento de código. Os principais recursos criados foram o *pop-up* de alerta, o *circular avatar* para foto do perfil, o *drawer* que se trata do menu lateral, o falto button que é o botão que tem na maioria das páginas e um text form, que aparece também em uma grande parte das telas.



**Figura 25: Estrutura de arquivos da pasta resources do fron-end.**

**Fonte [autora, 2021]**

Com os recursos já criados, a próxima etapa foi então desenvolver todas as telas da aplicação. Dentro da pasta *ui*, foram criadas novas pastas separando os arquivos por suas respectivas *features*.



**Figura 26: Estrutura de pastas dentro da pasta ui projeto do fron-end.**

**Fonte [autora, 2021]**

A primeira tela que abre no aplicativo é a tela de *login*, onde possui uma opção de fazer um cadastro caso o usuário ainda não possua um registro no sistema. A tela de cadastro é da *feature register.* Ao fazer o login, ou fazer o cadastro, o usuário é redirecionado a tela *home*, onde no inferior da tela possui uma barra de navegação entre páginas. As páginas do menu de navegação são as telas Início(*homeView*), Projetos (*projectsView*) e Adicionar Projeto (*addNewProject*) respectivamente. Foi usado um menu de navegação para facilitar a experiência do usuário com as principais funcionalidades do sistema. Em todas as telas que fazem parte do menu navegação, consta o menu hambúrguer (*drawer*) para acessar as opções do perfil (clicando na foto do usuário), e opções de gerenciamento dos projetos.

Ao entrar na página de Meu Perfil (*profile*), existem três abas com informações sobre o usuário, sendo elas a tela de Meus Dados (*myData*), Currículo (*curriculum*) e Pagamento (*payment*).

No menu lateral tem também um item para cada opção relacionada a projetos, além também da página de Favoritos (*favorite*) e da opção de fazer *logOut* no sistema (redirecionado para a tela de *login*).

As páginas do menu lateral relacionadas aos projetos são respectivamente a tela Meus Projetos (*myProjectsPage*), e Minhas Inscrições (*mySubscriptions*). Na tela Meus Projetos existe a opção de ver detalhes do projeto (*myProjectsDetails*), onde existe a opção de editar (*editMyProject*) ou excluir o projeto, e tem a opção de ver os inscritos no meu projeto (*my Enrolled*), e dentro desta tela existe a opção de ver o currículo dos usuários inscritos (*viewCurriculum*). Já na página minhas inscrições existe também a opção de ver detalhes do projeto (*mySubProjectDetails*) onde possui a opção de cancelar a inscrição.

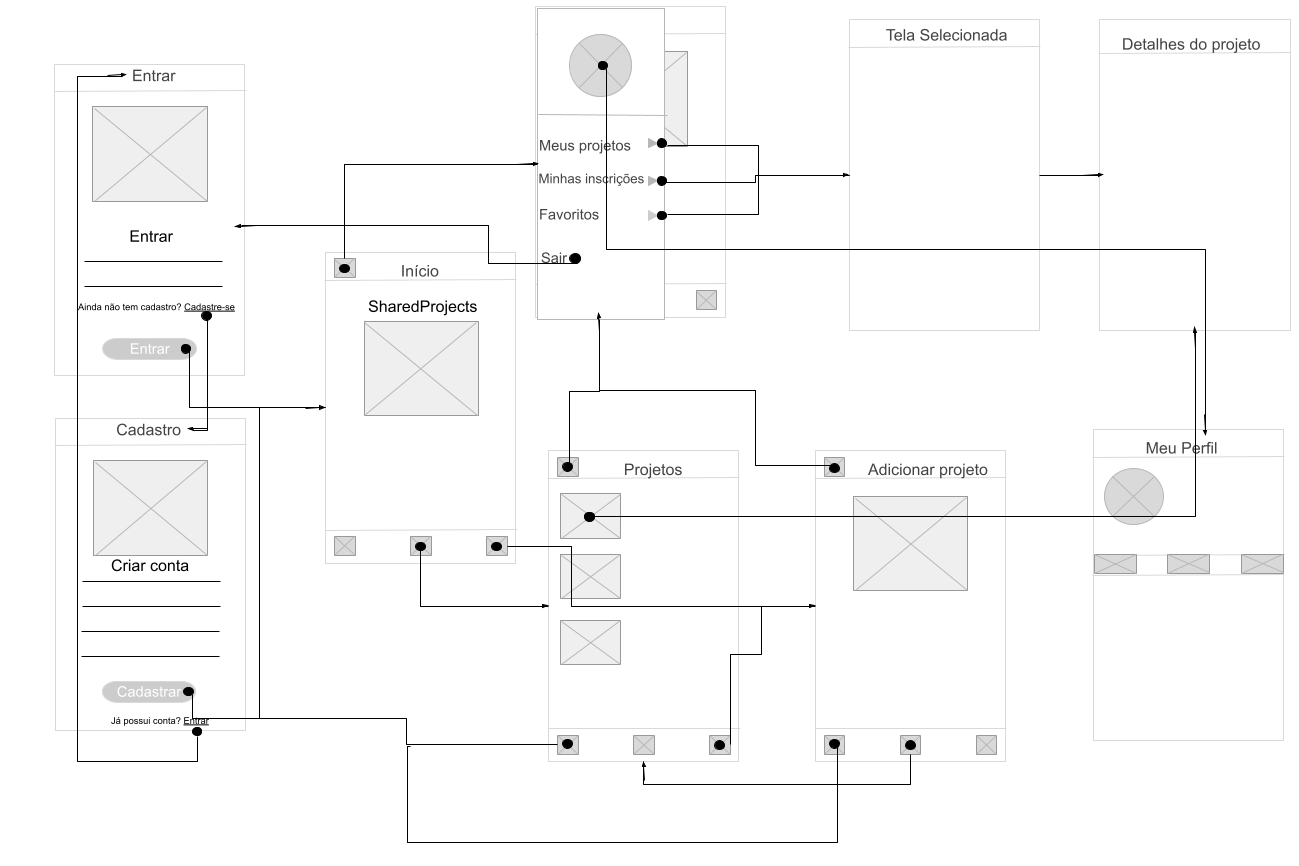
Na tela de Favoritos também existe a opção de ver detalhes do projeto, nesse caso chama a mesma tela (*projectDetails*) que a página Projetos (*projectsListView*) do menu inicial chama pra ver os detalhes do projeto selecionado.

O arquivo *layout* foi criado para ser usado em todos os arquivos de *ui*, com componentes que são necessários para todas as páginas.

* + - 1. **Fluxo das telas**

No item anterior já houve uma breve explicação sobre as telas, relacionando com os arquivos do projeto, e também com uma pequena explicação sobre o fluxo entre as telas.

Foi desenvolvido um diagrama mostrando o fluxo das principais telas do aplicativo, por onde o usuário vai transitar e através de quais botões é possível fazer a transição.

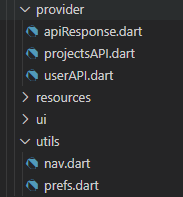


**Figura 27: Fluxo das principais telas do projeto do fron-end.**

**Fonte [autora, 2021]**

* + 1. **Desenvolvimento da comunicação com as APIs**

Com a implementação do back-end e do banco de dados prontos e o fluxo de telas funcionando, a próxima etapa foi desenvolver o código dos *providers* para a comunicação com as APIs do *back-end*. Nesta etapa também foi criada a pasta *utils* com arquivos que auxiliam nesta parte da implementação.



**Figura 28: Fluxo das principais telas do projeto do fron-end.**

**Fonte [autora, 2021]**

1. **CONCLUSÃO**

Na exposição das conclusões do trabalho, deve-se atender para os objetivos estabelecidos no início do trabalho. É preciso evidenciar em que parte do trabalho cada um dos objetivos foi atendido, bem como uma síntese dos resultados alcançados (BEUREN, 2003).

Há autores com a visão de que as conclusões devem ser curtas e focadas no que foi estabelecido no plano de trabalho. Outros entendem que é o momento do autor contribuir com suas observações. Recomenda-se é o equilíbrio entre essas duas posturas, a fim de que as conclusões sejam reais e comprováveis. Ressalta-se aqui o cuidado para não incorrer no erro de concluir sobre o que não foi investigado, a fim de evitar o comprometimento dos objetivos propostos.

Deve constar da conclusão uma recapitulação sintetizada dos capítulos e a autocrítica, onde você fará um balanço dos resultados obtidos pela pesquisa.

Andrade (1995) ressalta que a conclusão deve ser “breve, exata e convincente”.

**REFERÊNCIAS**

MICROSOFT. **Documentação Técnica da Microsoft**. Disponível em: <https://docs.microsoft.com/pt-br/documentation/ >. Acesso em 10 mai.2021.

DOTNET, Microsoft. **Why Choose .NET?**. Disponível em: <https://dotnet.microsoft.com/platform/why-choose-dotnet >. Acesso em 10 mai.2021.

DART. **Dart overview**. Disponível em: <https://dart.dev/overview>. Acesso em 25 abr.2021.

FLUTTER. **Flutter, Ferramenta de desenvolvimento de software.**. Disponível em: <https://flutter.dev/ >. Acesso em 25 abr.2021.

DEVELOPERS, Google. **Android Studio**. Disponível em: <https://developer.android.com/studio?hl=pt >. Acesso em 26 abr.2021.

FIGMA. **Recursos de Prototipagem**. Disponível em: <https://www.figma.com/prototyping/ >. Acesso em 13 jan.2021.

SHAHED. **EF Core Migrations in ASP .NET Core**, 2018. Disponível em: <https://wakeupandcode.com/ef-core-migrations-in-asp-net-core/ >. Acesso em 15 mai.2021.

RESTFULAPI. **REST API Tutorial**, 2020. Disponível em: <https://medium.com/imaginelearning/asp-net-core-3-1-microservice-quick-start-c0c2f4d6c7fa>. Acesso em 02 mai.2021.

WILCOCK, Dustin. **ASP.NET Core 3.1 Web API Quick Start**, 2020. Disponível em: <https://restfulapi.net/ >. Acesso em 02 mai.2021.

CHACON, Scott. **Pro Git Book**, 2014. 2. ed. Disponível em: <https://git-scm.com/book/en/v2 >. Acesso em 02 mai.2021.

JSON. **Introducing JSON**. Disponível em: <https://www.json.org >. Acesso em 13 mai.2021.

POSTMAN. **The Collaboration Platform for API Development**. Disponível em: <https://www.postman.com/ >. Acesso em 12 mai.2021.

BERTALANFFY, Ludwig Von. **Teoria geral dos sistemas**. 2. ed. Petrópolis: Vozes,1975.

BEUREN, Ilse M. et al. **Como elaborar trabalhos monográficos em contabilidade :** teoria e prática. São Paulo: Atlas, 2003.

CERVO, Amado L.; BERVIAN, Pedro A. **Metodologia científica**: para uso dos estudantes universitários. São Paulo: McGraw-Hill do Brasil, 1983.

COLAUTO, Romualdo Doglas; BEUREN, Ilse Maria. Coleta, análise e interpretação dos dados. In: **Como elaborar trabalhos monográficos em contabilidade:** teoria e prática. São Paulo: Atlas, 2003. p. 117-144.

COSTA, Magnus Amaral. **Contabilidade da construção civil e atividade imobiliária**. São Paulo: Atlas, 2000.

\_\_\_\_\_\_.**Fixação de padrões, cálculo e significância das variações:** um enfoque gerencial. 1981. Dissertação (Mestrado em Ciências Contábeis) – Departamento de Contabilidade e Atuária, Faculdade de Economia, Administração, Universidade de São Paulo, São Paulo, 1981

GIL, Antonio C. **Como elaborar projetos de pesquisa**. 3. ed. São Paulo: Atlas,1995,

\_\_\_\_\_\_. **Métodos e técnicas de pesquisa social**. 5. ed. São Paulo: Atlas,1999,

LAPONI, Juan C. **Estatística usando excel 5 e 7**. São Paulo: Lapponi Treinamento e Editora, 1997.

MARCONI, Mariana A; LAKATOS, Eva M. **Metodologia científica**. 5. ed. São Paulo: Atlas, 2002.

MICHAELIS: moderno dicionário da língua portuguesa. São Paulo: Companhia Melhoramentos, 1998.

RAUPP, Fabiano Maury; BEUREN, Ilse Maria. Coleta, análise e interpretação dos dados. In: **Como elaborar trabalhos monográficos em contabilidade:** teoria e prática. São Paulo: Atlas, 2003. p. 76-97.

RICHARDSON, Roberto J. **Pesquisa social**: métodos e técnicas. 3. ed. São Paulo: Atlas, 1999.

TRIPOLI, Tony; FELLIN, Fhillip; MEYER, Henry. **Análise da pesquisa social**. 2. ed. Rio de Janeiro: Francisco Alves, 1981.

TRIVIÑOS, Augusto. N. S. **Introdução à pesquisa qualitativa em ciências sociais**. São Paulo: Atlas, 1991.

**GLOSSÁRIO**

(Opcional)

Serve para apresentar, em ordem alfabética, as palavras e/ou expressões técnicas de uso restrito ou de sentido obscuro, utilizadas no texto, acompanhadas das respectivas definições.

**APÊNDICES**

(Opcional)

APÊNDICE é todo material elaborado pelo autor do trabalho, como texto, questionário, relatórios, entre outros. Este tem a finalidade de complementar ou argumentar, sem prejuízos do trabalho. Os apêndices são indicados por letras maiúsculas, consecutivas, travessão e título. No corpo do trabalho deve aparecer (APÊNDICE A).

Ex:

**APÊNDICE A – QUESTIONÁRIO SOBRE A PROFISSÃO CONTÁBIL**

**APÊNDICE B – RELAÇÃO DAS MAIORES EMPRESAS PESQUISADAS**

**ANEXOS**

(Opcional)

Anexo é todo material coletado de fontes externas, como formulários de empresas, artigos, leis, tabelas utilizadas em empresas, entre outros.

Serve para apresentar documentos não elaborados pelo acadêmico, os quais servirão como fundamentação, comprovação e ilustração.

Os anexos são identificados per letras maiúsculas consecutivas, travessão e título. No corpo do trabalho deve aparecer (ANEXO A).

Ex:

**ANEXO A – MODELO DE FICHAS DE ESTOQUE**

**ANEXO B – LEI DE RESPONSABILIDADE FISCAL**