Metodologia de Desenvolvimento de Software
da Defensoria Pública de Minas Gerais

1 – BACKEND

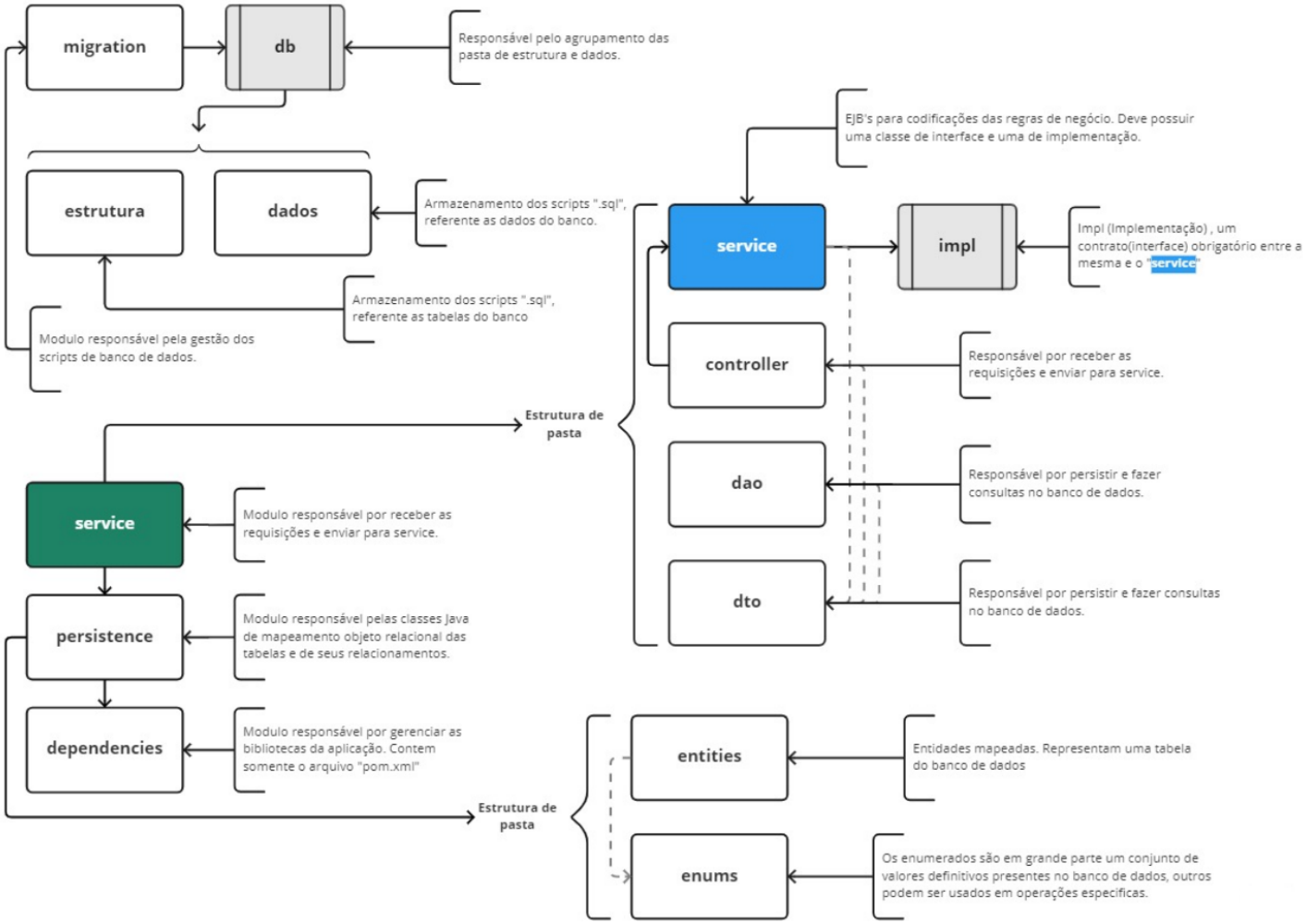
1.1 – JAVA

Baseado na escalabilidade necessária para melhor funcionamento dos sistemas da DPMG, de forma integrada, foi definido a utilização do JAVA e toda sua stack para implementações de aplicações robustas.
Java é uma linguagem de programação e plataforma computacional lançada pela pela Sun Microsystems em 1995. Existem muitas aplicações cujo funcionamento estão condicionados à presença do Java instalado. Java é uma linguagem gratuita, está entre as linguagens mundialmente mais usadas e atualmente é mantida pela Oracle.
Principais características do Java:

- Suporte à orientação a objetos;
- Portabilidade;
- Segurança;
- Linguagem Simples;
- Alta Performance;
- Dinamismo;
- Interpretada (o compilador pode executar os bytecodes do Java diretamente em qualquer máquina);
- Distribuído;
- Independente de plataforma;
- Tipada (detecta os tipos de variáveis quando declaradas).

Os sistemas comunicam entre si através de “cliente-servidor”, onde as aplicações frontends (cliente) realizam requisições na API Gateway (servidor) para manipulações e consultas de dados. As API’s são responsáveis por receber as request de microsserviços, validar o acesso aos endpoints via permissão do usuário, manipular o dado e retornar a informação solicitada via response.
Os microsserviços possuem definições a serem seguidas, como por exemplo a implementação de rollbacks em todos os métodos que utilize transação. Também deverão ser serviços simples e objetivos. Todos os endpoints devem ser documentados através do Swagger e com o SCSDP corretamente configurado.

1.1.1 – REPRESENTAÇÃO ARQUITETURAL



1.1.2 – ARCHETYPE BACKEND DPMG

Esse projeto foi criado para facilitar a criação de novos projetos JAVA na DPMG.

Todos os padrões no desenvolvimento serão mantidos nesse projeto para melhorar a integração de novos participantes na equipe e até mesmo a criação de projetos no dia a dia.

Esse projeto só precisará ser clonado caso queira contribuir com o archetype, fora isso não é necessário para criação de novos projetos

1.1.3 – COMANDOS BÁSICOS

Execute o comando na pasta onde deseja criar o projeto.

mvn archetype:generate -DarchetypeGroupId=br.def.mg.defensoria -DarchetypeArtifactId=archetype-backend -DarchetypeVersion=2.1.9 -Dprojeto=nome-projeto

Ps: Caso o settings do maven não esteja na pasta padrão deverá adicionar o seguinte atributo

-s /diretorio/arquivoSettings.xml

Ao gerar esse comando será gerado um projeto maven multi-modules com a camada de persistência e serviço, um modulo de dependências também será gerado com as dependências mais utilizadas nos projetos

1.1.4 – CONFIGURAÇÃO DO PROJETO

O projeto gerado será montado com a seguinte Stack de tecnologias

- Java 17
- JPA
- EclipseLink
- Apache Commons 3.1
- Jakarta JAX-RS
- Jakarta EJB
- Swagger
- JUnit 5
- Lombook
- Integração com SCSDP
- JACOCO
- Surefire reports

1.1.5 – DEPEDÊNCIAS ENTRE PROJETOS

- dpmg-commons
- scsdp-commons

As configurações de charset e encode já serão realizadas.

O pacote de todos os modulos já definidos por padrão com o seguinte valor:

br.def.mg.defensoria.nomeProjeto

O groupId é definido pelo padrão da defensoria:

br.def.mg.defensoria

Observação: O pacote default gerado deve ser alterado caso o nome do projeto seja composto, alterando para nome.projeto não é possível automatizar essa parte trocando “-” por “.”

1.1.6 – CONFIGURAÇÃO DO SONAR

Foi configurado os testes para conseguir analisar qual a porcentagem de coverage do código fonte.

1.2 – ESTRUTURA

Baseado em organização dos códigos para garantir melhor leitura e manutenção de serviços, foi definido uma arquitetura de forma que seus módulos possuam baixo acoplamento
entre si e de fácil compreensão, permitindo um desenvolvimento ágil.

A arquitetura base é composta por 1 projeto “pai” que agrupa 4 projetos “filhos”. Estes 4 projetos são definidos como módulos, e eles são separados em: Dependencies, Migration, Persistence e Service. Em casos específicos em que o projeto tenha automações, poderá ser incluído os módulos chamados de Job e Shared.

1.2.1 – DEPENDENCIES

Este modulo é responsável por gerenciar as bibliotecas utilizadas pelos demais módulos da aplicação. A imagem abaixo é um exemplo de como devem ser importadas as bibliotecas.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>br.def.mg.defensoria</groupId>
            <artifactId>dpmg-commons</artifactId>
            <version>3.5.16</version>
        </dependency>
        <dependency>
            <groupId>br.def.mg.defensoria.scsdp</groupId>
            <artifactId>scsdp-commons</artifactId>
            <version>4.2.1</version>
        </dependency>
        <dependency>
            <groupId>br.def.mg.defensoria.base.dpmg</groupId>
            <artifactId>base-dpmg</artifactId>
            <version>1.0.6</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

1.2.2 – MIRGRATION

O modulo de migration é responsável pela gestão dos scripts de banco de dados que deverão ser executados para o correto funcionamento do módulo Persistence. Os scripts
são organizados em pasta de estrutura e pasta de dados. Os dados devem ser separados por ambiente (Desenvolvimento, Homologação e Produção). Os scripts de estrutura
são executados pelo flyway ao rodar a pipeline.

1.2.3 – PERSISTENCE

Este modulo é responsável pelas classes Java de mapeamento objeto relacional das tabelas e seus relacionamentos. No persistence deve ter duas pastas, entities,
onde ficam as entidades mapeadas, e enums, onde ficam os enumerados.

1.2.4 – SERVICE

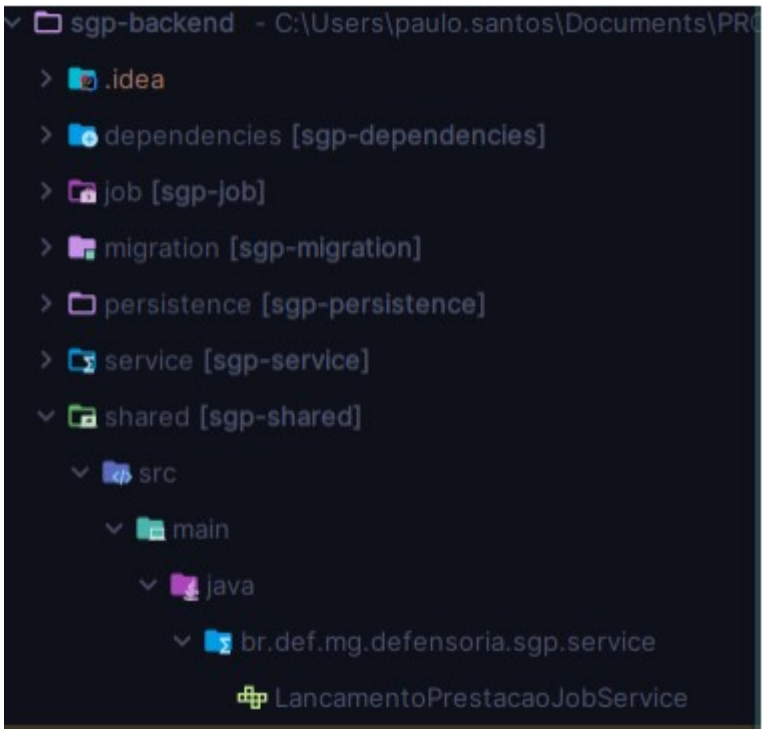
Neste modulo contém todo o fluxo de desenvolvimento das regras de negócio da aplicação. É nele que deve ser criado as Controllers (para serviços REST), os DTOs
(para objetos JSON), os EJB’s (para codificações das regras de negócio) e os DAOs (para persistir ou buscar no banco de dados).

1.2.5 – JOB

Este modulo possibilita o agendamento de tarefas, as quais serão executadas em horários pré-definidos. Lembrando que este modulo deverá existir somente em uma instância do WildFly, com isso evitará que as tarefas sejam executadas mais de uma vez devido as instâncias.

1.2.6 – SHARED

Seu objetivo é evitar que o Módulo Job tenha que fazer uma integração para acessar a serviços remotos dentro do projeto. Neste modulo é utilizado JNDI (Java Naming and Directory Interface), que é uma API que permite acesso à recursos externos através de um nome utilizando o conceito de lookup.



1.3 – DMPG-COMMONS

Projeto que centralizar as funcionalidades básicas de todos os sistemas.

1.3.1 – PARAMETROS UTIL

Utilizada para recupera os parâmetros do sistema.

```
ParametroUtil.getInstance().getParameter(ServerProperties.INSTITUCIONAL);
```

1.3.2 – DPMG API UTIL

Classe utilitária para a realização de integrações

1.3.2.1 – DEPENDÊNCIA

```xml
<dependency>
    <groupId>br.def.mg.defensoria</groupId>
    <artifactId>dpmg-commons</artifactId>
    <version>3.5.16</version>
</dependency>
```

1.3.2.2 – EXEMPLOS

- Requisições do tipo GET

```
return new DpmgApiUtil.Builder([URL])
        .timeout(Duration.ofSeconds([TIMEOUT]))
        .jsonResponse()
        .build()
        .realizarRequisicao(new DpmgApiUtil.ReturnType<>() {
        });
```

- Para adicionar query param:

```
{...}  .addParameter("[NOME PARAMETRO]", [NOME PARAMETRO]) ...
```

- Para ignorar a resposta:

```
{...}   .ignoreResponse()
                .build()
                .realizarRequisicao();
```

- Requisições tipo POST

```
return new DpmgApiUtil.Builder([URL])
        .timeout(Duration.ofSeconds([TIMEOUT]))
        .json()
        .jsonResponse()
        .utf8()
                .post(DpmgApiUtil.body([OBJETO]))
        .build()
        .realizarRequisicao(new DpmgApiUtil.ReturnType<>() {
        });
```

Onde:

Parâmetro	Tipo dado	Default	Exemplo
URL	String		http://dev-gerais.defensoria.mg.def.br/unidade/find-all
TIMEOUT (segundos)	int	5	10
OBJETO	Object		new Usuario()

- Para retornar uma resposta do tipo text/plain :

```
{...}  .textPlain() ...
```

- Requisições tipo PUT

```
return new DpmgApiUtil.Builder([URL])
        .timeout(Duration.ofSeconds([TIMEOUT]))
        .json()
        .jsonResponse()
        .utf8()
                .put(DpmgApiUtil.body([OBJETO]))
        .build()
        .realizarRequisicao(new DpmgApiUtil.ReturnType<>() {
        });
```

Onde:

Parâmetro	Tipo dado	Default	Exemplo
URL	String		http://dev-gerais.defensoria.mg.def.br/unidade/find-all
TIMEOUT (segundos)	int	5	10
OBJETO	Object		new Usuario()

- Requisições tipo DELETE

```
return new DpmgApiUtil.Builder([URL])
        .timeout(Duration.ofSeconds([TIMEOUT]))
        .jsonResponse()
        .delete()
        .build()
        .realizarRequisicao(new DpmgApiUtil.ReturnType<>() {
        });
```

Onde:

Parâmetro	Tipo dado	Default	Exemplo
URL	String		http://dev-gerais.defensoria.mg.def.br/unidade/find-all
TIMEOUT (segundos)	int	5	10

- Para adicionar token na requisição

```
{...}  .token([TOKEN]) ...
```

Onde:

Parâmetro	Tipo dado	Default	Exemplo
TOKEN	String	Bearer	eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ RMBwQR4C_0uehWmo_77ZrAB46YnPYmzJJ2Lb36GyyDXDwRP9Bt759hcVmUiGW

2 – PADRÃO DE ARQUITETURA FRONT-END

Os projetos Front-End da instituição DPMG e baseado no padrão de arquitetura “MVC” que melhora a conexão entre camadas de dados regras de negócio e a interação com usuário,
também seguimos os princípios do “SOLID“com ênfase nos dois primeiros princípios **“S – Single-responsibility Principle (Princípio da responsabilidade única)” e**
“O – Open-closed Principle (Princípio do aberto-fechado)”.

O padrão para criar aplicações VueJS v2.x e o “class component” que é estendido da pasta “data”, já no VueJS v3.x utilizamos o composition api o arquivo da pasta “data” e
declarado como uma constante.

Exemplo imagem VueJS v2.x e v3.x:

VueJS 2:

```vue
<template>
  <div>
    <h1>test</h1>
  </div>
</template>

<script lang="ts">
import ArquivoPdfData from "@/views/data/arquivo-pdf.data";
import Component from "vue-class-component";

@Component
export default class ArquivoPDF extends ArquivoPdfData {
  // ...
}
</script>

<style lang="scss" scoped></style>
```

VueJS 3:

```vue
<template>
  <h1>test</h1>
  <!-- ... -->
</template>

<script lang="ts" setup>
//...
</script>

<style lang="scss" scoped>
//...
</style>

```

2.2 – PLUGINS

As rotas são definidas com plugin vue-router para VueJS v2.x e v3.x.
O plugin de armazenamento de estado para VueJS v2.x e vuex, enquanto no VueJS v3.x e o Pinia.
O vue-i18n plugin de internacionalização do VueJS utilizado somente nas versões 3.x do vue.
Os plugins de front-end utilizados são Vuetify para VueJS v2.x e Quasar para v3.x.

Os plugins para criação de formulários para versão 3.x do Vue, será o Vee-validate em conjunto com Yup e Pinia.

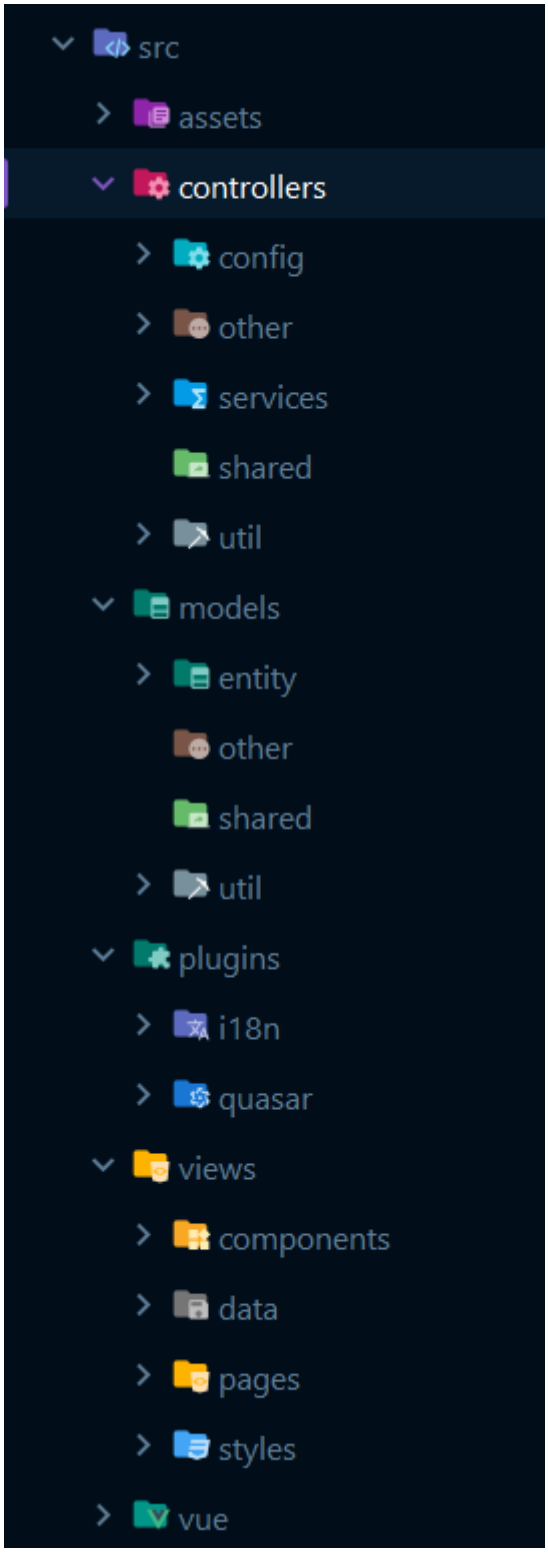
2.3 – REGRAS PARA CRIAÇÃO DE PASTAS E ARQUIVOS

Todas as pastas e arquivos são criados com letras minúsculas e separadas com hífen.
Todos arquivos com funções especificas como entity devem ser terminados com ponto seguido do nome da pasta, somente no caso de arquivos typescript.
Toda estrutura pode sofre adições de pasta como “assets”, “util” ou “@types” entre outras necessárias para facilitar o desenvolvimento, porém sempre
mantendo a estrutura base e seguindo os padrões estabelecidos.

2.3.1 – ESTRUTURA DE PASTA

- **Root** – Arquivos contendo variáveis de ambiente, configurações de lint entre outros arquivos básicos dependendo da aplicação.
- **Src** – Arquivos e pastas e iram compor o sistema.
- **Controllers** – Arquivos e pastas responsáveis por intermediar as requisições enviadas pela camada de interação com usuário.
- **Config** – Todos os arquivos que definem configurações ou meios de acesso a aplicações externas.
- **Services** – Todos os arquivos que conectam a API's.
- **Plugins** – Todos os arquivos que que fazem alterações no software permitindo utilização de fermentas externas como Quasar ou Vuetify.
- **Models** – Arquivos e pastas que controlam a forma como os dados se comportam por meio das funções, lógica e regras de negócios estabelecidas.
- **Entity** – Todas os arquivos que se referem a entidades pré-estabelecidas pela regra de negócio.
- **Shared** – Todas os arquivos que definem implementações obrigatórias com base nas regras definidas.
- **Views** – Arquivos e pastas responsáveis por apresentar informações de forma visual ao usuário.
- **Pages** – Todos os arquivos .vue que iram receber uma rota de acesso.
- **Components** – Todos os arquivos .vue que não recebem uma rota ou são amplamente utilizados no código.
- **Data** – Todos arquivos .ts que ofereceram suporte aos componentes .vue fazendo a comunicação com controller, quando
  utilizamos o framework Vue v2.x todos os componentes serão classes abstratas, no caso do Vue v3.x serão apenas classes.
- **Styles** – Folhas de estilo .scss ou .css serão utilizadas por todo aplicação ou apenas um componente especifico.

Exemplo na imagem abaixo.



2.4 – CONFIGURAÇÕES FRONT-END

- Ecma-version ES2021, ES2020
- Typescript
- Vite (Apenas Vue v3.x)
- Vue-i18n (Apenas Vue v3.x)
- Vee-Validate (Apenas Vue v3.x)
- Yup (Apenas Vue v3.x)
- Quasar (Apenas Vue v3.x))
- Pinia (Apenas Vue v3.x))
- Vuetify (Apenas Vue v2.x)
- Vuex (Apenas Vue v2.x)
- Vue-router
- Eslint
- Prettier
- Sass (utilizamos o scss)

2.5 – PALETA DE CORES

Variável	Hexadecimal	Cores
primary	#1e8364	
secondary	#3bc47f	
accent	#214843	
dark	#252525	
positive	#53645c	
negative	#a52a2a	
info	#dfd9cc	
warning	#ebbd57	

3 – MOBILE

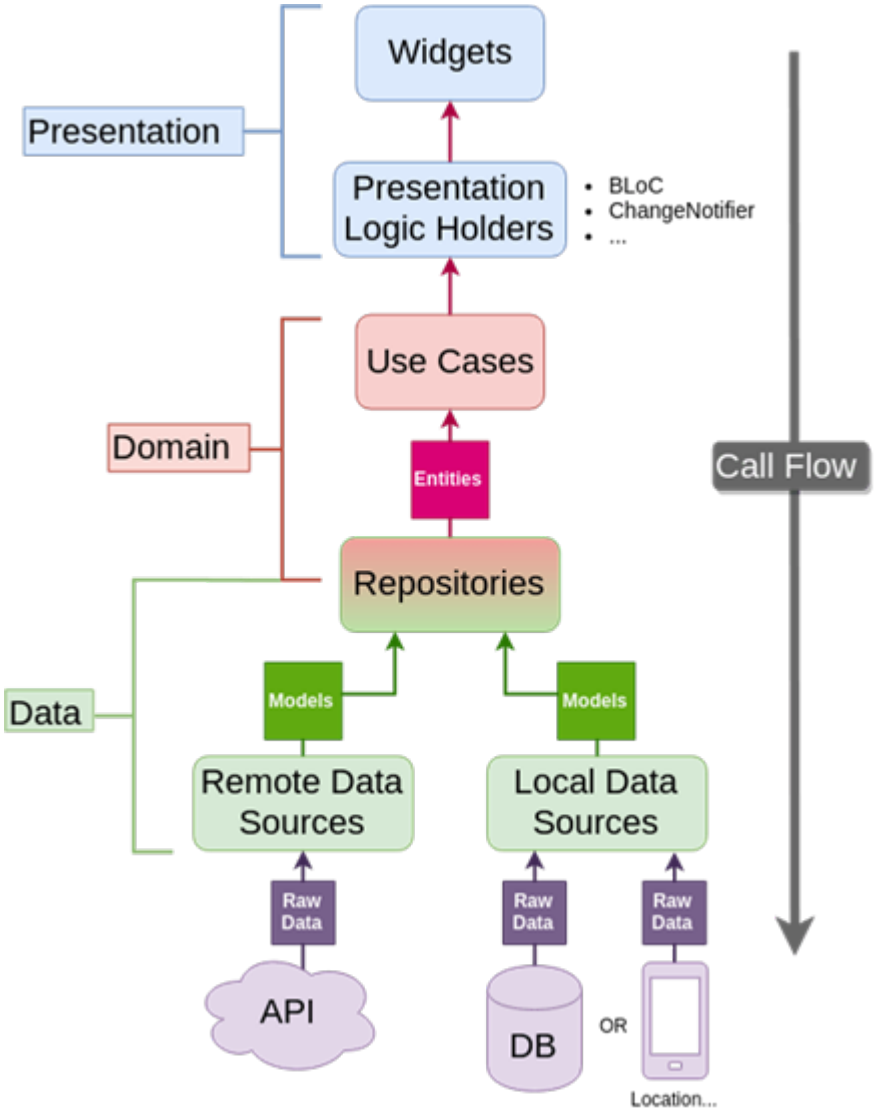
- Os projetos mobiles devem ser com Flutter (versão atual 3.0.5);
- A linguagem para desenvolvimento Flutter é o Dart (versão atual 2.17.6);
- O ambiente deve possuir o JDK 8, para a execução do SDK;
- A versão do Android SDK deve ser 32.1.0-rc1 ;
- As bibliotecas utilizadas no padrão/estrutura BLoC são 8.0.1;

É importante que os layouts, botões, fontes e ícones estejam de acordo com a Paleta de Cor da DPMG.

Partes de telas que serão utilizadas em mais de um lugar, deverão ser componentizadas, para tornar a manutenibilidade mais eficaz;

3.2 – CLEAN ARCHITECTURE

As aplicações de média e grande complexidade vêm tomando a Clean Architecture como uma das melhores soluções, de modo a criar uma arquitetura que seja facilmente escalável, testável e manutenível. Sendo assim, entendemos ser a melhor opção para aplicar nos projetos mobiles da DPMG.



3.2.1 – AS CAMADAS DO APP COM CLEAN ARCHITECTURE

A arquitetura se divide basicamente, nas seguintes camadas:

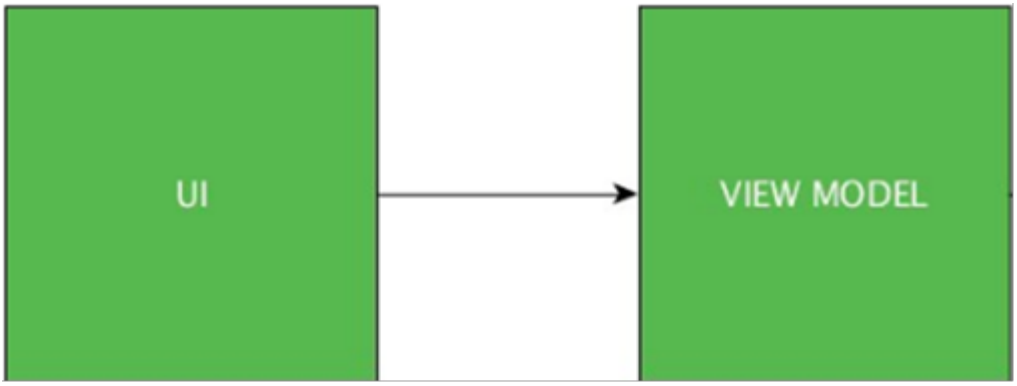
API / Local / DB ? Remoto / Local (DataSource) ? Repositório ? Casos de Uso ? BLoC ? View

A arquitetura segue as regras de que, cada camada deve se comunicar apenas com seu componente imediato, desta forma, segue conforme o esquema abaixo:

- A UI (User Interface) pode se comunicar somente, com a ViewModel;
- O BLoC pode se comunicar somente, com o UseCase;
- O UseCase pode se comunicar somente, com o Repository;
- O Repository pode se comunicar somente, com a DataSource.

3.2.2 – CAMADA DE APRESENTAÇÃO (PRESENTATION LAYER)

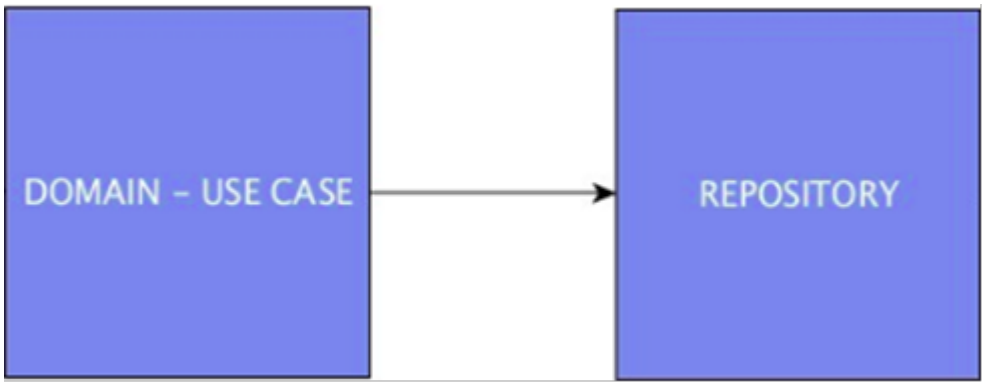
São representadas nesta camada, pelas Views e BLoC:



- A View é construída, utilizando o conceito de Widgets no Flutter, que descrevem como a interface visual deve ser representado, dado sua configuração e estado;
- O BLoC possui regras de negócio e é desenvolvida com objetivo de armazenar e gerenciar os dados, sendo eles recebidos
  por meio de requisições de API, banco de dados, etc.

3.2.3 – CAMADA DE DOMÍNIO (DOMAIN LAYER)

A composição desta camada, é feita pelos UseCases e Repositories:



- O UseCase é responsável por orquestrar o fluxo de dados das entidades e o direcionar para a regra de negócio, com a finalidade de atingir os objetivos que este é responsável;
- O Repository tem o objetivo de executar a lógica de acesso aos dados, sua responsabilidade é de obter e verificá-los
  onde estão, e decidir onde procurar a cada momento.

3.2.4 – CAMADA DE DADOS (DATA LAYER)

Nesta camada se encontram os dados e de onde eles podem ser acessados.

O DataSource é a fonte de dados propriamente dita, ou seja, é o que a implementação executa, para acessar os dados.

4 – DOCUMENTO ARQUITETURA DE SOFTWARE — NODE JS

O node e utilizado como microsserviços e websockets, a versão do node e a v18.x/v20.x. Framework escolhido para trabalhar com mesmo foi NestJS pois ele tem como proposta
facilitar o desenvolvimento de aplicações back-end independente do protocolo de comunicação que elas utilizem, segundo fator e sua documentação que nos traz diferentes
formas de estruturar a aplicação NestJS além de ser rica em detalhes e exemplos, a tornando de fácil entendimento. O ORM escolhido foi o Prisma. Toda projeto desenvolvido deve possuir documentação que e gerada com Swagger. Parte de segurança foi gerar com guard's que e mesmo conceito dos "guard's"
do Angular. O desenvolvimento e baseado na arquitetura "MVC" sendo aplicado também os conceitos do "S.O.L.I.D".

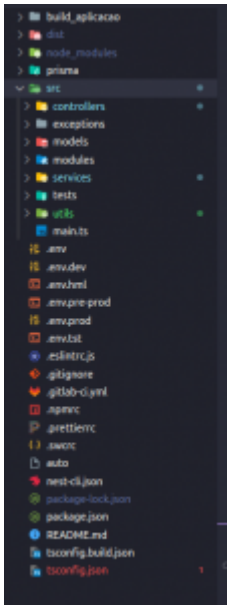
4.1 – REGRAS PARA GERAÇÃO DA APLICAÇÃO

- Aplicações devem ser criadas com NestJS, em casos extraordinários onde não se possa gerar uma aplicação por qualquer
  limitação técnica ou estrutural deve ser autorizada pelo gestor.
- O "ORM" utilizado por padrão e o Prisma.
- Aplicação deve contemplar o máximo possível dos conceitos do "S.O.L.I.D".
- Toda aplicação desenvolvida com node deve ser microsserviço ou websocket.
- Toda pasta com composições de nome devem ser separados com hífen.
- Toda pasta a partir da pasta "src" pode implementar pastas utilitárias, desde que a mesma seja coesa em sua criação.
- Toda aplicação deve ser protegida com Throttler do NestJs, caso aplicação não seja feita com o framework deve ser adotar uma solução do mesmo tipo.

4.2 – ESTRUTURA DE PASTA

- **Root** – Onde ficam configurações, básicas como arquivos “env”, “lint” entre outros.
- **Src** – Onde a estrutura da aplicação será gerada “controllers”, “models” e “swagger” também e possível criar pastas auxiliares.
    - **Controllers** – Aqui ficam os responsáveis por gerenciar a aplicação contendo pastas para serviços com segurança, banco de dados e chamados do client-side.
        - **Controllers –** Esta pasta contém os controladores do aplicativo NestJS, que são responsáveis por lidar com as requisições HTTP, processar dados e retornar respostas adequadas.
        - **Services** – Serviço que receberá chamados do client-side, cada service cria uma pasta com nome do serviço, gerando um aquivo “.module” e “.service” obrigatoriamente, caso necessário pode receber uma pasta “dto” ou outras pastas auxiliares com máximo de um nível. Uma “service” pode ou não implementar um “guard”.
        - **Models** – Aqui ficam os itens referentes a regra de negócio.
            - **Shared** – Conteúdo obrigatório a ser implementado pelas entidades ou qualquer item que contenha regra de negócio pré-definida.
            - **Entity** – Entidades com base nas abstrações, implementam os itens referentes a mesma contidos na pasta “shared”.
            - **Dto –** Contém classes ou interfaces que definem a estrutura dos dados transferidos entre diferentes partes do aplicativo, como requisições HTTP e respostas.
        - **Modules –** Contém os módulos do aplicativo NestJS, que agrupam funcionalidades relacionadas, como rotas, controladores, provedores de serviços e outros recursos.

Exemplo da estrutura de pasta:



4.3 – TECNOLOGIAS

- Ecmaversion 2021, 2020.
- Typescript
- Node v18.x/v20.x
- NestJs
- Prisma