GYM-OPS

Membros:

Jeferson Augusto de Melo Gomes

Guilherme Pereira Borges

Wendel Rodrigues Viana

Sheiely Do Ó

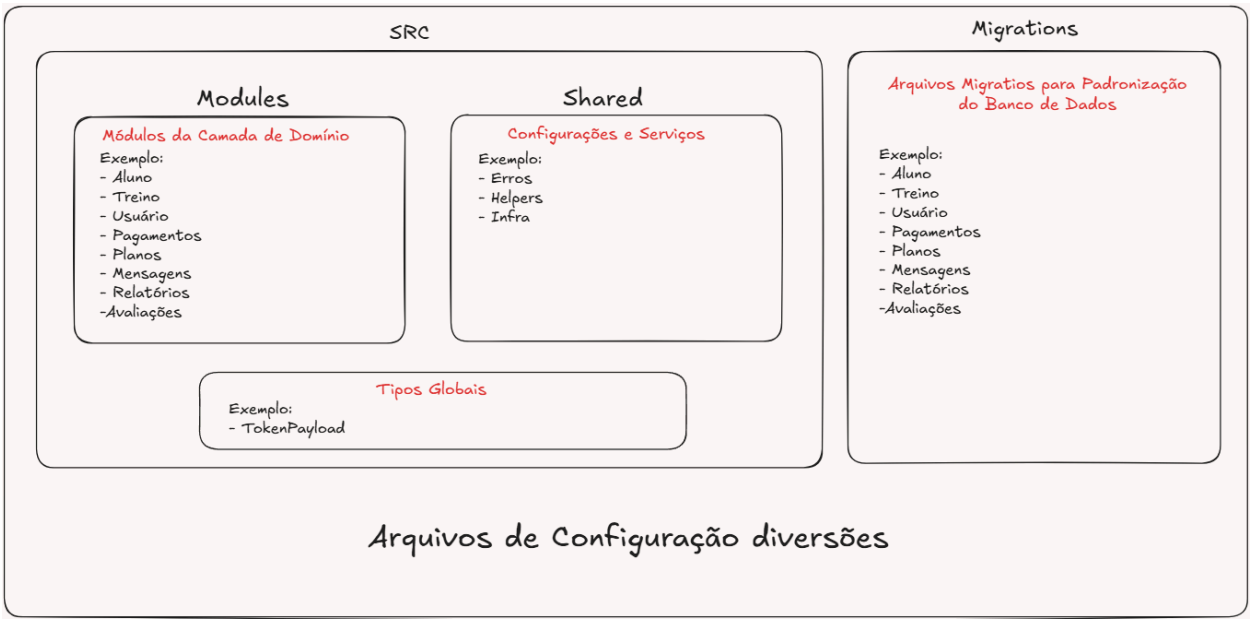
Sobre

O **GYM-OPS** é um sistema de gerenciamento para academias, desenvolvido como uma aplicação desktop para facilitar a administração de alunos, instrutores e avaliações físicas, sem acesso direto dos alunos. Ele oferece recursos como **cadastro e gestão de alunos e instrutores, controle de avaliações físicas, além de gerenciamento de planos e pagamentos**. O sistema também inclui notificações via e-mail para manter a comunicação eficiente entre os envolvidos.

Tecnologias

A aplicação utiliza **Node.js com TypeScript** no back-end e **JavaScript com React** no front-end, sendo implementada com o **Electron** para garantir compatibilidade multiplataforma. O banco de dados escolhido é o **PostgreSQL**, e o **Redis** é utilizado para implementar filas com **cache e otimização do desempenho do sistema**, garantindo maior eficiência no processamento de dados e redução da latência. O desenvolvimento segue uma abordagem ágil baseada em **Scrum**, com sprints semanais e reuniões para planejamento e revisão das entregas.

Estrutura das Pastas (Back)



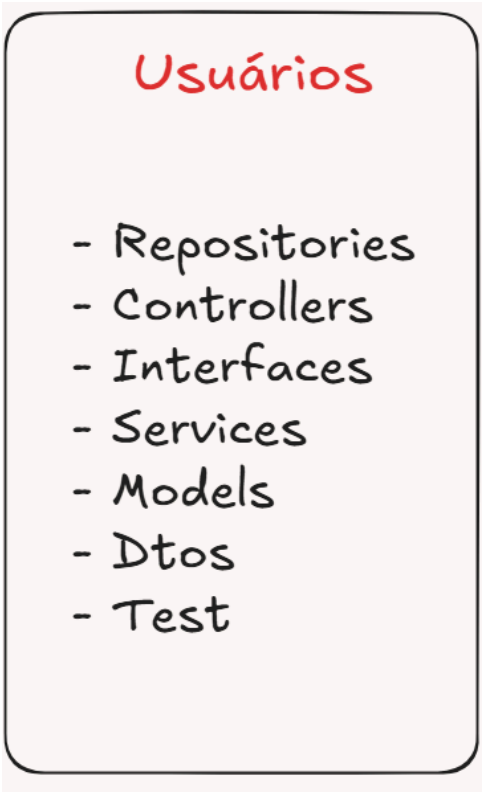
Src (Source)

O diretório **SRC** é o núcleo do código-fonte do sistema, onde estão localizados os módulos principais, serviços compartilhados, tipos globais e outras configurações importantes para o funcionamento da aplicação.

Modules (Módulos da Camada de Domínio)

A pasta **Modules** contém os módulos principais do sistema, representando as entidades do domínio da aplicação. Cada módulo encapsula sua lógica de negócio e operações associadas.

- **Exemplo de Modulo:**



Shared

A pasta
Shared contém arquivos e serviços compartilhados entre os diferentes módulos, garantindo reutilização de código e organização.

Subcategorias:

- **Erros**: Definição de classes e mensagens de erro para padronização do tratamento de exceções.

- **Helpers**: Funções auxiliares e utilitárias para diferentes partes do sistema.
- **Infra**: Configuração de infraestrutura do sistema, como conexão com banco de dados, cache e serviços externos.

Padrões Usados no Projeto

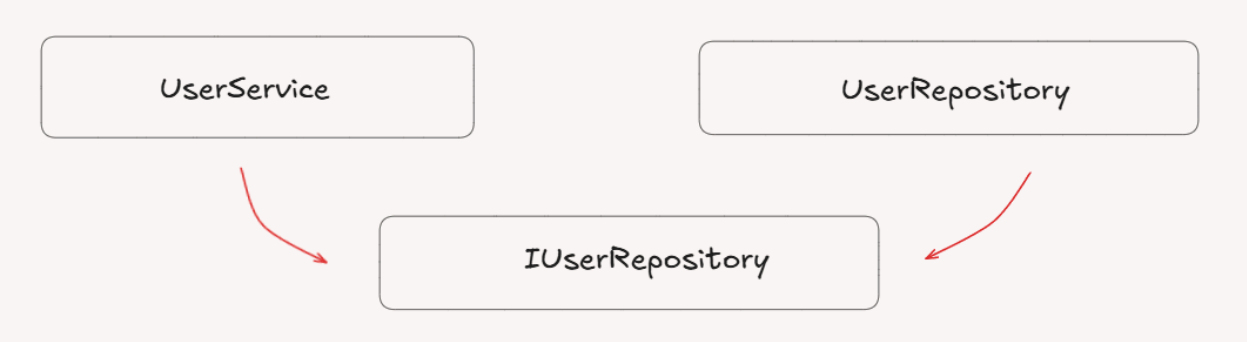
1. Inversão de Dependência

Seguimos o
Princípio da Inversão de Dependência (DIP - Dependency Inversion Principle), um dos cinco princípios do SOLID, que garante maior flexibilidade e desacoplamento entre as camadas do sistema. Esse princípio define que **módulos de alto nível não devem depender diretamente de módulos de baixo nível**, mas sim de abstrações (interfaces). Isso melhora a testabilidade, facilita a manutenção e torna o código mais modular.

Utilizamos o **tsyringe**, uma biblioteca de injeção de dependências para TypeScript, que nos permite registrar e gerenciar dependências automaticamente. Isso garante que cada camada da aplicação dependa de abstrações e não de implementações concretas.

Em **GYM-OPS**, aplicamos a inversão de dependência da seguinte forma:

1. **Criamos interfaces para definir contratos de repositórios** (por exemplo, `IUserRepository`).
2. **As classes de repositório implementam essas interfaces** (exemplo: `UserRepository implements IUserRepository`).
3. **No container do tsyringe**, registramos os repositórios e serviços, permitindo que o framework resolva as dependências automaticamente.



Exemplo em código:

```
//gym-ops\Back\src\modules\alunos\Interface\IAlunoRepository.ts

import { Aluno } from "../models/Aluno";
import { z } from "zod";
import { AlunoSchema } from "../dto/AlunoSchema";

export interface IAlunoRepository {
  create(data: z.infer<typeof AlunoSchema> & { adm_id: number }): Promise<Alu
  list(adm_id: number, limit: number, offset: number): Promise<Aluno[]>;
  findById(id: number, adm_id: number): Promise<Aluno | null>;
  findByEmail(email: string, adm_id: number): Promise<Aluno | null>;
  findByCpf(cpf: string, adm_id: number): Promise<Aluno | null>;
  update(id: number, adm_id: number, data: z.infer<typeof AlunoSchema>): Prom
  delete(id: number, adm_id: number): Promise<void>;
  findByPlanId(plan_id: number, adm_id: number): Promise<Aluno[] | null>;
  listByFrequency(adm_id: number, offset: number, limit: number): Promise<Alunc
  getEmail(adm_id: number): Promise<string[]>;
  listRecentRecords(adm_id: number, offset: number, limit: number): Promise<Alu
  listRecentFrequency(adm_id: number, offset: number, limit: number): Promise<A
}
```

```
//gym-ops\Back\src\modules\alunos\service\AlunoService.ts
```

```
@injectable()
export class AlunoService {
  constructor(
    @inject("AlunoRepository")
    private alunoRepository: IAlunoRepository,
    @inject("PlanoRepository")
    private planoRepository: IPlanoRepository,
    @inject("UserRepository")
    private userRepository: IUserRepository,
  ) {}


  /*Implementação dos Métodos*/
}
```

```
//gym-ops\Back\src\modules\alunos\repository\AlunoRepository.ts
@injectable()
export class AlunoRepository implements IAlunoRepository {
  constructor(@inject("Database") private db: Knex) {}

  /*Implementação dos Métodos*/
}
```

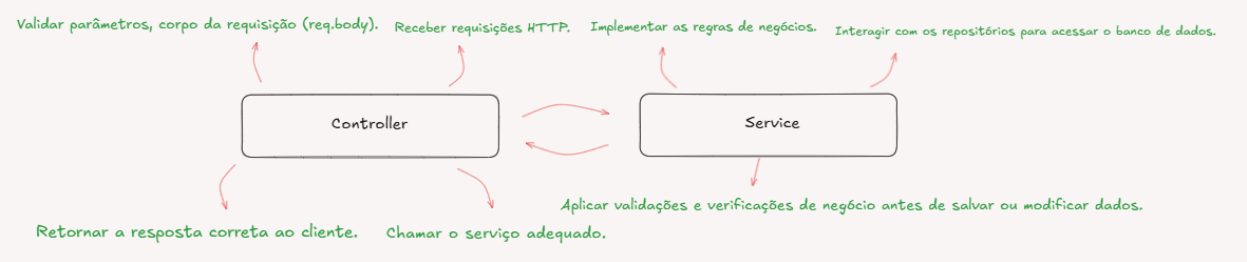
2. Controller → Service

No
GYM-OPS, seguimos o padrão de separação de responsabilidades utilizando a estrutura **Controller → Service**, onde cada camada tem uma função específica dentro da aplicação. Esse padrão melhora a organização do código, facilita a manutenção e permite a reutilização de lógica de negócios em diferentes partes do sistema.

- **Controller**
  Os

Controllers são responsáveis por **receber as requisições**, **validar os dados de entrada** e chamar os **serviços apropriados** para executar a lógica de negócios. Essa camada não contém regras de negócio, apenas direciona a solicitação para o serviço adequado e retorna a resposta.

- **Service**
  A camada de
  Service contém toda a lógica de negócios da aplicação. Ela recebe os dados do **Controller**, processa as regras do sistema e interage com os repositórios para persistência no banco de dados.



Exemplo em código :

```
//gym-ops\Back\src\modules\alunos\controller\AlunoController.ts

@injectable()
export class AlunoController {
  async create(req: Request, res: Response): Promise<Response> {
    const data = AlunoSchema.parse(req.body); //validando entrada

    const adm_id = req.user.adm_id;

    const alunoService = container.resolve(AlunoService);
```

```
    const aluno = await alunoService.create(data, adm_id); //chamando o serviço

    return res.status(201).json(aluno); // retornando resposta
  }
```

```
//gym-ops\Back\src\modules\alunos\service\AlunoService.ts

@injectable()
export class AlunoService {
  constructor(
    @inject("AlunoRepository")
    private alunoRepository: IAlunoRepository,
    @inject("PlanoRepository")
    private planoRepository: IPlanoRepository,
    @inject("UserRepository")
    private userRepository: IUserRepository,
  ) {}

  async create(data: z.infer<typeof AlunoSchema>, adm_id: number) {
    const admById = await this.userRepository.findAdmById(adm_id);


    //validação de regras de negócio

    if (!admById || admById.role != "ADM") {
      throw new AppError("Administrador inválido", 404);
    }

    const alunoByEmail = await this.alunoRepository.findByEmail(data.email, adm_

    if (alunoByEmail) {
      const error = alunoByEmail.status? "Email já cadastrado": "Usuário com emai

      throw new AppError(error, 409);
```

```
    }

    const userByCpf = await this.alunoRepository.findByCpf(data.cpf, adm_id);

    if (userByCpf) {
      const error = userByCpf.status? "Cpf já cadastrado": "Usuário com cpf já exi

      throw new AppError(error, 409);
    }

    const plano = await this.planoRepository.findById(data.plan_id, adm_id);

    if (!plano) {
      throw new AppError("Plano não encontrado", 404);
    }

    if (!data.health_notes || data.health_notes.length < 1) {
      data.health_notes = null;
    }

    const alunoData = { ...data, adm_id };

    return await this.alunoRepository.create(alunoData);
  }

  }
```

3. Repository

Utilizamos o padrão **Repository (Repositório)** para **abstrair e encapsular a lógica de acesso ao banco de dados**, garantindo que as camadas superiores (Service e Controller) não precisem interagir diretamente com a base de dados. Isso melhora

a separação de responsabilidades, facilita a manutenção e torna o código mais testável.

Além disso, **seguimos o princípio de programação orientada a interface**, garantindo que os serviços e controladores não dependam diretamente da implementação dos repositórios, mas sim de **interfaces que definem o contrato de uso**. Isso nos permite **trocar implementações facilmente**, sem impactar o restante da aplicação.

4. Dto (Data Transfer Object)

No
GYM-OPS, utilizamos o padrão **DTO** para **padronizar, validar e transportar dados** entre as diferentes camadas da aplicação. Os DTOs garantem que as informações enviadas e recebidas sigam um formato estruturado, facilitando a manutenção e reduzindo a exposição direta das entidades do banco de dados.

A validação dos DTOs é feita utilizando a biblioteca
Zod, que permite definir esquemas de validação de forma declarativa e eficiente. O **Zod** garante que os dados estejam corretos antes de serem processados, reduzindo erros e aumentando a segurança da aplicação.

Exemplo em código :

```
//gym-ops\Back\src\modules\alunos\dto\AlunoSchema.ts

import { z } from "zod";

export const AlunoSchema = z.object({
  name: z.string().min(1, { message: "O nome não pode estar vazio" }),
  date_of_birth: z
    .string()
    .refine((val) => !isNaN(Date.parse(val)), { message: "Invalid date" })
    .refine((val) => /^\d{4}-\d{2}-\d{2}$/.test(val), {
      message: "A data deve estar no formato yyyy-mm-dd",
    }),
  email: z.string().email({ message: "Email inválido" }),
  telephone: z.string().regex(/^\(\d{2}\) \d{4,5}-\d{4}$/, {
    message: "Telefone deve estar no formato (XX) XXXXX-XXXX",
  }),
  cpf: z.string().regex(/^\d{3}\.\d{3}\.\d{3}-\d{2}$/, {
    message: "CPF deve estar no formato XXX.XXX.XXX-XX",
  }),
  plan_id: z.number().int().positive({ message: "ID do plano inválido" }),
  health_notes: z
    .string()
    .max(500, { message: "Observações de saúde muito longas" })
    .optional(),
  status: z.boolean().optional().default(true),
  gender: z
    .enum(["M", "F", "O"], {
      message: "Gênero inválido",
    })
    .optional()
    .default("O"),
});
```

5. DataMapper’s

No **GYM-OPS**, utilizamos o padrão **Data Mapper** para **converter dados entre diferentes camadas da aplicação**, garantindo que a estrutura dos objetos seja apropriada para cada contexto. Esse padrão melhora a separação de responsabilidades, reduz o acoplamento entre entidades e bancos de dados, e facilita a manutenção do código.

Os **Mappers** são especialmente úteis para transformar dados vindos do banco de dados em **objetos de domínio**, ou para preparar esses objetos antes de enviá-los para a camada de apresentação (API, UI). Dessa forma, os repositórios lidam apenas com a persistência, enquanto os serviços e controllers trabalham com dados já formatados.

Exemplo em código :

```
//gym-ops\Back\src\modules\alunos\models\Aluno.ts

export class Aluno {

/*Atributos de aluno*/


  static fromDatabase(data: any): Aluno {
    return new Aluno(
      data.id,
      data.name,
      data.date_of_birth,
      data.email,
      data.telephone,
      data.cpf,
      data.plan_id,
      data.health_notes,
      data.status,
```

```
      data.gender,
      data.adm_id,
      new Date(data.created_at),
    );
  }
}
```

6. Singleton

No
GYM-OPS, o Singleton é aplicado para **garantir que o sistema utilize uma única instância do Knex.js** (biblioteca para comunicação com o banco de dados PostgreSQL), evitando a criação de múltiplas conexões desnecessárias, o que poderia impactar negativamente o desempenho da aplicação.

Antes do Singleton

```
import knex, { Knex } from "knex";
import config from "../../../../../knexfile";
import { container } from "tsyringe";

const db = knex(config.development);
container.registerInstance<Knex>("Database", db);

export default db;
```

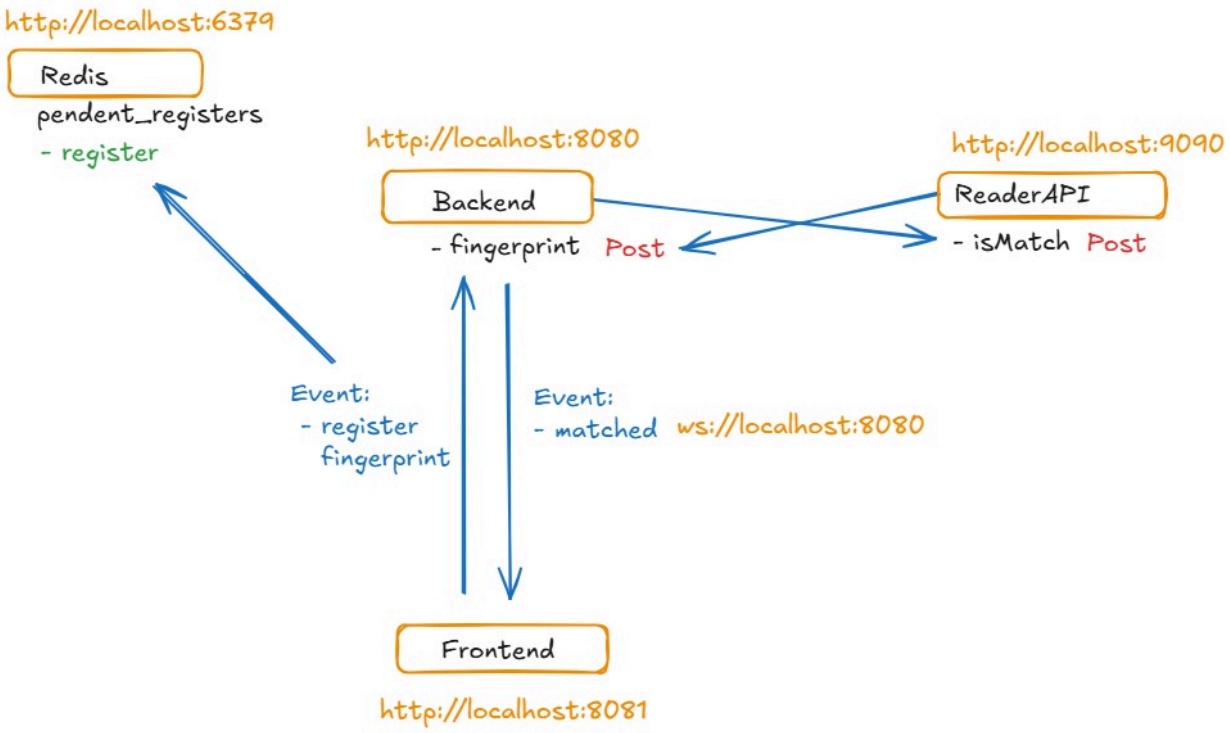
Depois do Singleton

```
//gym-ops\Back\src\shared\infra\http\config\database.ts

import knex, { Knex } from "knex";
```

```
import config from "../../../../../knexfile";
import { container } from "tsyringe";

class Database {
  private static instance: Knex;

  private constructor() {} // Impede a instanciação direta

  public static getInstance(): Knex {
    if (!Database.instance) {
      Database.instance = knex(config.development);
    }
    return Database.instance;
  }

container.registerInstance<Knex>("Database", Database.getInstance());

export default Database.getInstance();
```

7. Observer

O
GYM-OPS utiliza **WebSockets e eventos assíncronos** para implementar o padrão observer para comunicação entre **Frontend, Backend, Redis e ReaderAPI**. A imagem fornecida mostra um **fluxo de eventos** baseado no **Padrão Observer**:



Na imagem, tanto o **Backend** quanto o **Frontend** atuam como **observadores e sujeitos observáveis**, registrando interesse em receber atualizações sempre que há uma mudança de estado.

No **Frontend**, quando o estado do leitor de digitais muda de **"login"** para **"register"**, um evento é enviado ao **Backend** via **WebSocket**. O Backend, ao receber esse evento, processa a alteração de estado e toma as devidas ações.

Por outro lado, no **Backend**, quando o estado muda para **"matched"**, ele emite um evento para o **Frontend**. O **Frontend** então recebe essa informação, processa a resposta e exibe a atualização na interface do usuário.

Esse fluxo assíncrono garante uma comunicação eficiente e em tempo real entre os dois sistemas, utilizando o **Padrão Observer** para manter ambos sempre sincronizados.