

1. CRIANDO PROJETO NODE.JS

Crie a pasta do Projeto e depois o comando abaixo, para gerar o package.json.

npm init -y

Fazer com o que Node.js execute em TS

npm i typescript @types/node tsx tsup -D

Gerar o arquivo tsconfig.json com o comando abaixo e trocar o target para "es2020"

npx tsc - -init

Instalar o Fastify.js

npm i fastify

Crie uma pasta chamada **src** e um arquivo chamado **server.ts**.

```
import { app } from "./app";

app.listen({
  host: '0.0.0.0',
  port: 3333,
}).then(() => {
  console.log('🚀 HTTP Server Running!')
})
```

Crie um arquivo chamado **app.ts**, na mesma pasta src.

```
import fastify from "fastify";

export const app = fastify()
```

No arquivo package.json, crie os scripts de inicialização abaixo:

```
"scripts": {
  "start:dev": "tsx watch src/server.ts",
  "start": "node build/server.js",
  "build": "tsup src --out-dir build"
},
```

Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/2b8913b8af976120c2322bff65fafa8b970e5f1d

2. CARREGANDO VARIÁVEIS DE AMBIENTE

Instalar biblioteca para carregar as variáveis de ambientes dotenv

npm i dotenv

Instale a biblioteca zod para validar as variáveis de ambiente criadas e carregas.

npm i zod

Na raiz do projeto, crie um arquivo chamado ".env", para incluir as variáveis de ambiente que iremos criar durante o projeto.

```
NODE_ENV=dev
```

Na pasta src, crie outra pasta chamada "env" e depois um arquivo chamado "index.ts", conforme segue:

```
import 'dotenv/config'
import { z } from 'zod'

// Cria os schemas
// Definindo os valores padrões, caso não informado
const envSchema = z.object({
  NODE_ENV: z.enum(['dev', 'test', 'production']).default('dev'),
  PORT: z.coerce.number().default(3333)
})

// Valida
const _env = envSchema.safeParse(process.env)

// Verifica se deu erro na validação
if (_env.success === false) {
  console.error('❌ Invalid environment variables',
    _env.error.format())
}
```

```
    throw new Error('Invalid environment variables.')
  }

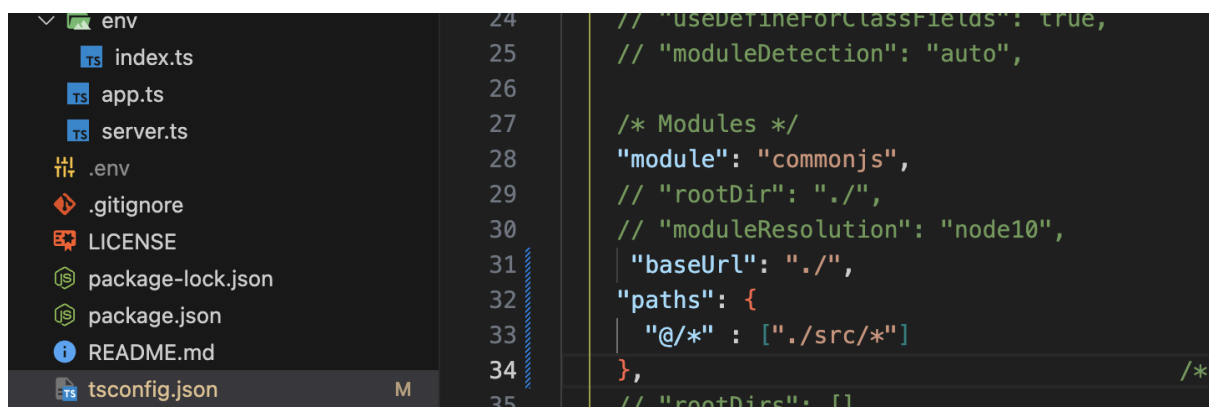
export const env = _env.data
```

Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/fa1964640e9027a21dbf2d2cb525532b168739db

3. CRIANDO ALIASES DE IMPORTAÇÃO

À medida que o projeto vai aumentando a complexidade e para reduzir a confusão nas importações dos arquivos, é interessante configurar o **baseUrl** no arquivo **tsconfig.json**.



Com essa configuração, todo arquivo passa a ser importado a partir da raiz do projeto.

Exemplo, imagine que dentro de src tenha a pasta "primeira", depois dentro a pasta "segunda" e depois dentro a pasta "terceira". Agora imagine que você deseja importar um arquivo que está antes do src, você deveria voltar uma a uma das pastas "../..". Configurando o baseUrl, você utiliza '@/nomedaPasta', porque não irá precisar retornar as pastas, mas começar sempre da raiz da importação.

4. ORM PRISMA

O ORM [Prisma](#) é uma ferramenta de mapeamento objeto-relacional (ORM) moderna e poderosa para bancos de dados. Ele facilita a interação entre a aplicação e o

banco de dados, permitindo que os desenvolvedores escrevam consultas de banco de dados usando uma sintaxe familiar de linguagem de programação, em vez de consultas SQL diretamente.

O Prisma é uma ferramenta de código aberto que oferece suporte a vários bancos de dados populares, como PostgreSQL, MySQL e SQLite. Ele permite que os desenvolvedores definam modelos de dados usando uma linguagem de definição de modelo declarativa e, em seguida, fornece métodos para realizar operações de leitura, gravação, atualização e exclusão (CRUD) nesses modelos de dados.

O Prisma também oferece suporte a recursos avançados, como transações, relacionamentos entre tabelas, migrações de banco de dados e consultas complexas. Além disso, ele é compatível com muitos frameworks e tecnologias de back-end populares, o que o torna uma escolha atraente para muitos desenvolvedores e equipes de desenvolvimento.

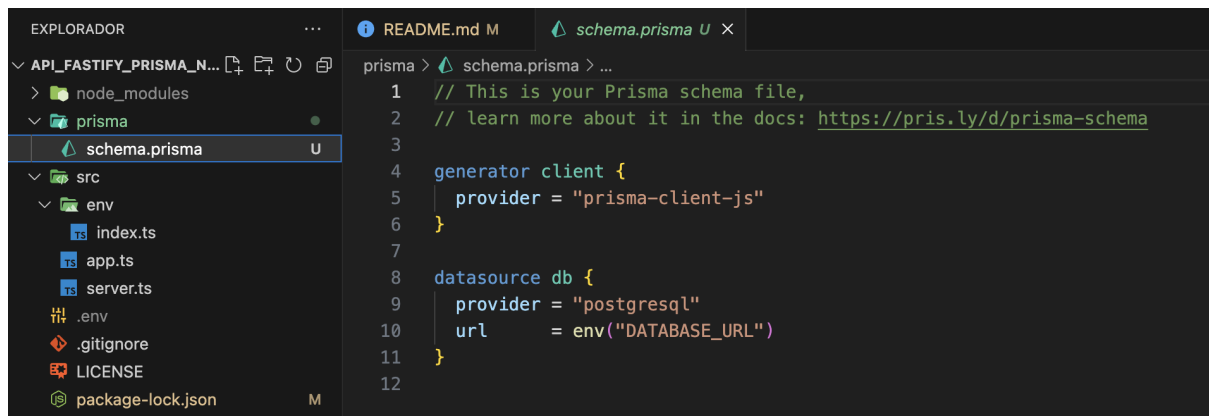
Instale o Prisma como dependência de desenvolvimento

npm i prisma -D

Inicializar o prisma.

npx prisma init

Será gerada a pasta abaixo:

A screenshot of a code editor interface. On the left, the 'EXPLORADOR' (Explorer) sidebar shows a file tree with folders like 'node_modules', 'prisma', and 'src', and files like 'index.ts', 'app.ts', 'server.ts', '.env', '.gitignore', 'LICENSE', and 'package-lock.json'. The 'prisma' folder is expanded, showing 'schema.prisma'. The main editor area shows the content of 'schema.prisma' with the following code:

```
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 generator client {
5   provider = "prisma-client-js"
6 }
7
8 datasource db {
9   provider = "postgresql"
10  url       = env("DATABASE_URL")
11 }
12
```

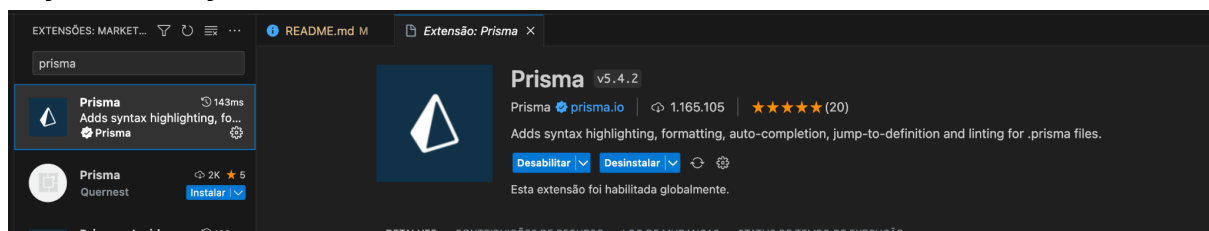
4.1 CRIANDO TABELA USERS

Vamos trabalhar com uma tabela simples de usuário porque o objetivo desta aula é utilizar o Fastify junto com o ORM Prisma. Mais detalhes sobre o Prisma, consulte o material da aula anterior.

Crie o código para criar a tabela User, logo abaixo do código gerado no schema.prisma.

```
model User {  
  id String @id @default(uuid())  
  name String  
  email String @unique  
  @@map("users")  
}
```

Faça a instalação da extensão [Prisma](#) no VScode



Prisma: Configurando extensão no VSCode

1. Instale a extensão [Prisma](#) no seu Visual Studio Code.
2. Abra a Paleta de Comandos:
 1. Se estiver no Windows ou Linux: **CTRL + SHIFT + P**
 2. Se estiver no macOS: **CMD + SHIFT + P**
3. Abra as configurações em JSON buscando por:
 1. Se o seu VSCode estiver em português: **Abrir as Configurações do usuário (JSON)**
 2. Se o seu VSCode estiver em inglês: **Open User Settings (JSON)**
4. Adicione dentro do JSON o código abaixo:

Para criar as tipagem dos dados criados, ou seja, interação com o ts, utilizando o comando abaixo:

npx prisma generate

Comando para buscar as tabelas do bd já criado e criar os modelos no JS/TS (Prisma Client).

npm prisma db pull

Instale o Cliente do Prisma

npm i @prisma/client

Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/18216c27cf94dd03294f72253d17563afc83d108

5. 'POSTGRESQL COM DOCKER

[Docker](#) é uma plataforma de software que permite que você crie, teste e implemente aplicativos rapidamente. Ele permite empacotar e distribuir aplicativos em contêineres, que são ambientes leves e portáteis que incluem tudo o que é necessário para executar um software, incluindo o código, as bibliotecas, as dependências e as variáveis de ambiente.

Os contêineres Docker são executados em cima do sistema operacional do host e compartilham o núcleo do sistema operacional, o que os torna mais eficientes em termos de recursos do que as máquinas virtuais tradicionais. Eles garantem que os aplicativos sejam executados da mesma maneira em diferentes ambientes, o que ajuda a eliminar problemas de inconsistência entre desenvolvimento, teste e produção.

Com o Docker, os desenvolvedores podem empacotar seus aplicativos juntamente com todas as dependências em um contêiner Docker, o que facilita a criação, a implantação e o gerenciamento de aplicativos em vários ambientes, como data centers locais, nuvens públicas e privadas, bem como em máquinas individuais.

Vamos começar acessando o hub com as imagens disponíveis no Docker:
<https://hub.docker.com/search?q=postgre>

A imagem do bitnami possui mais segurança do que a imagem oficial da postgresql.
<https://hub.docker.com/r/bitnami/postgresql>

Após a instalação do Docker, verifique a instalação com o comando:
docker -v

Para criar a imagem do docker:

```
docker run - -name NOME_DO_DOCKER -e POSTGRESQL_USERNAME=docker -e POSTGRESQL_PASSWORD=docker -e POSTGRESQL_DATABASE=api_node -p 5432:5432 bitnami/postgresql
```

Listar todos os containers rodando

docker ps

Listar todos os container criados em algum momento

docker ps -a

Inicializar um container docker, basta digitar o id do container ou o nome, como segue

docker start NOME_DO_DOCKER

Para parar, bastar digitar o comando

docker stop NOME_DO_DOCKER

Para excluir, basta digitar o comando

docker rm NOME_DO_DOCKER

Crie a imagem conforme código abaixo:

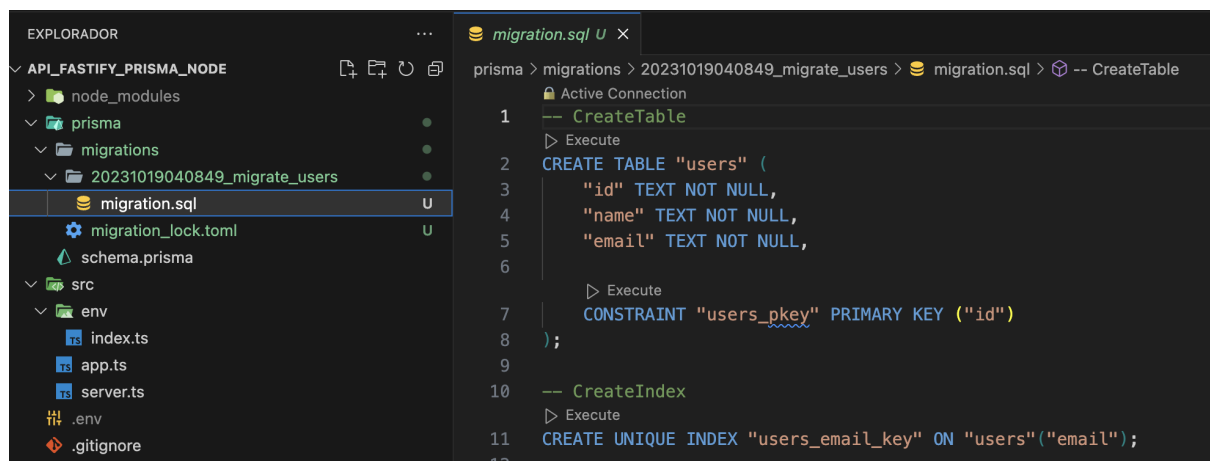
docker run --name api_prisma_node -e POSTGRESQL_USERNAME=docker -e POSTGRESQL_PASSWORD=docker -e POSTGRESQL_DATABASE=api_prisma_node -p 5432:5432 bitnami/postgresql

No arquivo .env, altere a variável de ambiente DATABASE_URL, conforme segue:

DATABASE_URL="postgresql://docker:docker@localhost:5432/api_prisma_node?schema=public"

Com o container do docker rodando, vamos criar a tabela Users no banco de dados criado. Para isso, vamos executar o comando de criação de migration do Prisma. Em seguida será questionado o nome da migration, para controle de versão. Adicionei o nome create_users

npx prisma migrate dev

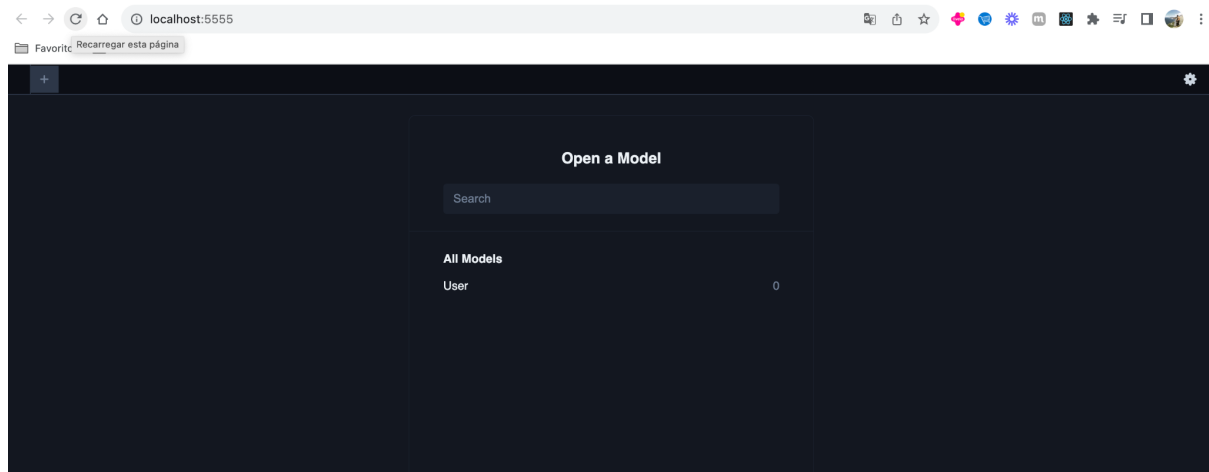


The screenshot shows a code editor with a file explorer on the left and a terminal on the right. The file explorer shows a project structure with folders like 'node_modules', 'prisma', 'migrations', and 'src'. The 'migrations' folder is expanded, showing a file named 'migration.sql'. The terminal shows the command 'prisma > migrations > 20231019040849_migrate_users > migration.sql > -- CreateTable' being executed. The output shows the SQL code for creating a table named 'users' with columns 'id', 'name', and 'email', and a primary key constraint on 'id'. The code is as follows:

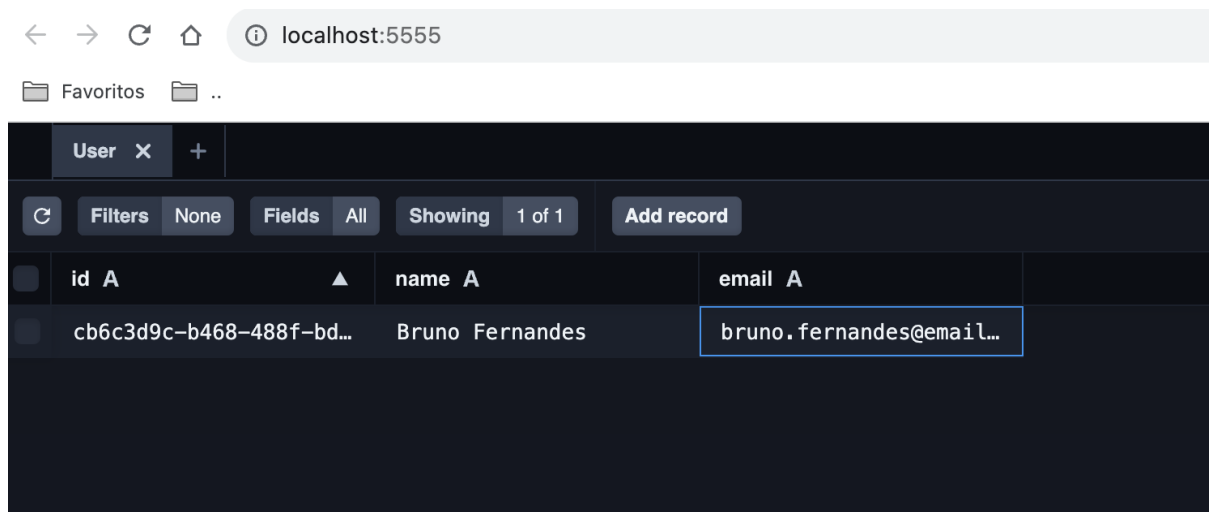
```
1 -- CreateTable
2 CREATE TABLE "users" (
3   "id" TEXT NOT NULL,
4   "name" TEXT NOT NULL,
5   "email" TEXT NOT NULL,
6
7   CONSTRAINT "users_pkey" PRIMARY KEY ("id")
8 );
9
10 -- CreateIndex
11 CREATE UNIQUE INDEX "users_email_key" ON "users"("email");
12
```

Para visualizar a tabela criada, execute o comando para inicializar o Prisma Studio

`npx prisma studio`



O Prisma Studio será automaticamente inicializado no navegador, sendo possível visualizar as tabelas criadas e ainda editar os registros das tabelas.



Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/b67dc2c9ea893923872f03b404fd9ed487df0357

5.1 Docker Compose

O Docker Compose é usado para definir os serviços de uma aplicação em um arquivo YAML e, em seguida, pode ser usado para criar e iniciar todos os serviços de uma só vez. Com o Docker Compose, você pode executar várias partes de um aplicativo como contêineres separados, em vez de colocá-los todos em um único contêiner.

Ao usar o Docker Compose, é possível definir os serviços, redes e volumes necessários para executar um aplicativo Docker completo. Isso inclui especificar variáveis de ambiente, comandos de inicialização, mapeamentos de porta e muito mais.

Crie o arquivo chamado docker-compose.yml.

```
version: '3'

services:
  api_prisma_node:
    image: bitnami/postgresql
    ports:
      - 5432:5432
    environment:
      - POSTGRESQL_USERNAME=docker
      - POSTGRESQL_PASSWORD=docker
      - POSTGRESQL_DATABASE=api_prisma_node
```

docker compose up -d

Utilize o comando acima para inicializar o container configurado. Também é possível parar o container inicializado com o comando stop.

docker compose stop

6. CONFIGURANDO REGISTROS DE LOGS

Essa etapa é opcional, porque é uma configuração que auxilia no processo de desenvolvimento, apresentando os logs das solicitações realizadas pelo Prisma.

Primeiramente, crie uma pasta chamada 'lib' e em seguida um arquivo chamado 'prisma.ts' com o seguinte código:

```
import { env } from '@env'
import { PrismaClient } from '@prisma/client'

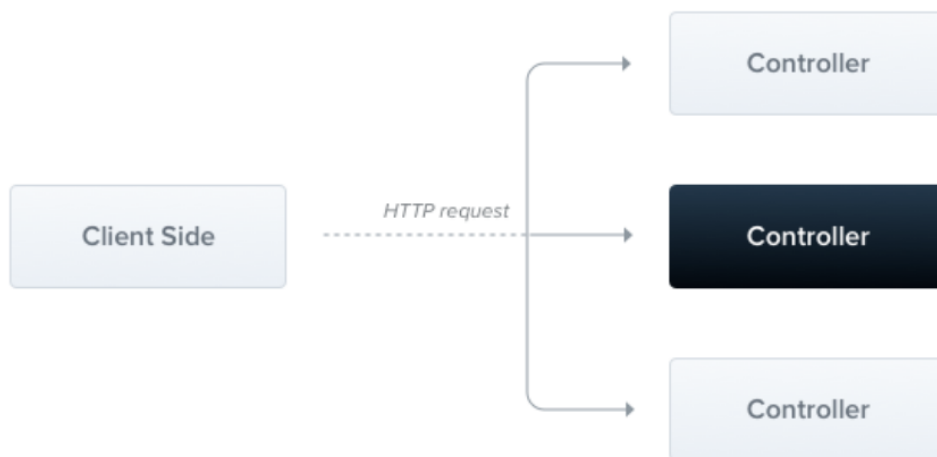
export const prisma = new PrismaClient({
  log: env.NODE_ENV === 'dev' ? ['query'] : [],
})
```

O objeto env do módulo @/env contém informações sobre a configuração do ambiente (como NODE_ENV), esse código configura o cliente Prisma para realizar o log de consultas apenas quando o aplicativo estiver em ambiente de desenvolvimento. Isso pode ser útil para depuração e otimização de consultas durante o desenvolvimento, enquanto em produção, o log de consultas é desativado para melhorar o desempenho e reduzir a quantidade de informações sensíveis expostas.

7. CONTROLLER DE USERS

Agora iremos criar um controller para receber as requisições HTTP e tratar as informações, podendo realizar ações como validar dados, buscar informações do banco de dados e enviar uma resposta ao cliente. O controller é uma parte importante da arquitetura de um servidor web, e ajuda a manter as regras de negócio separadas do restante da aplicação.

Segundo a própria documentação do [Nest.JS](#), um importante Framework JS, os controladores são responsáveis por lidar com as solicitações recebidas e retornar as respostas ao cliente.



O objetivo de um controlador é receber solicitações específicas para a aplicação. O mecanismo de roteamento controla qual controlador recebe as solicitações. Frequentemente, cada controlador possui mais de uma rota, e rotas diferentes podem executar ações diferentes.

Primeiramente iremos criar uma pasta chamada **'http'** dentro da pasta 'src'. Depois outra pasta chamada **'controllers'**, responsáveis por possuir todos os controllers criados na aplicação. Dentro da pasta http iremos criar o arquivo **'routes.ts'**, responsável por possuir/conhecer todas as rotas existentes.

Endereço: src/http/routes.ts

```
import { FastifyInstance } from 'fastify'
import { register } from '../controllers/register'

export async function appRoutes(app: FastifyInstance) {
  app.post('/users', register)
}
```

Esse trecho de código está exportando uma função `appRoutes` que, ao ser chamada com uma instância do Fastify, registra uma rota POST para `/users` que é manipulada pela função `register` do arquivo `../controllers/register`. Isso é comum em frameworks web para modularizar a definição de rotas e controladores, tornando o código mais organizado e fácil de manter.

Agora, configure adequadamente o arquivo `app.ts`.

Endereço: src/app.ts

```
import fastify from "fastify";
import { appRoutes } from '@http/routes'

export const app = fastify()

app.register(appRoutes)
```

O código configura um servidor web usando o framework Fastify, importa as rotas da aplicação e registra essas rotas na instância do Fastify para que o servidor possa lidar com as requisições correspondentes.

Por fim, iremos criar a rota de registrar usuário dentro do controllers.

Endereço: src/http/controllers/register.ts

```
import { FastifyReply, FastifyRequest } from 'fastify'
import { prisma } from '@lib/prisma'
import { z } from 'zod'

export async function register(request: FastifyRequest, reply:
FastifyReply) {
  const registerBodySchema = z.object({
    name: z.string(),
    email: z.string().email(),
  })

  const { name, email } = registerBodySchema.parse(request.body)

  await prisma.user.create({
    data: {
      name,
      email
    },
  })

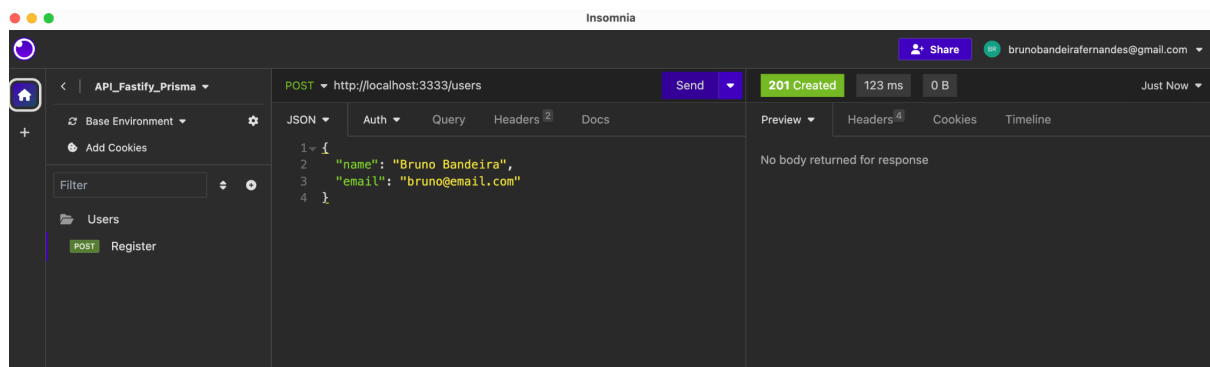
  return reply.status(201).send()
}
```

A parte do código que contém o código zod é apenas a validação das entradas dos dados. A parte que contém a const 'registerBodySchema' é um esquema de

validação definido com Zod para validar o corpo da requisição. Ele espera um objeto com propriedades name (do tipo string) e email (do tipo string e que deve ser um e-mail válido). Se não utilizou o zod como validador ou esteja utilizando o formato js, sem o zod, não precisa incluir esta parte do código.

A parte do código 'await prisma.user.create({...})' usa o Prisma para criar um novo usuário no banco de dados com os dados fornecidos na requisição.

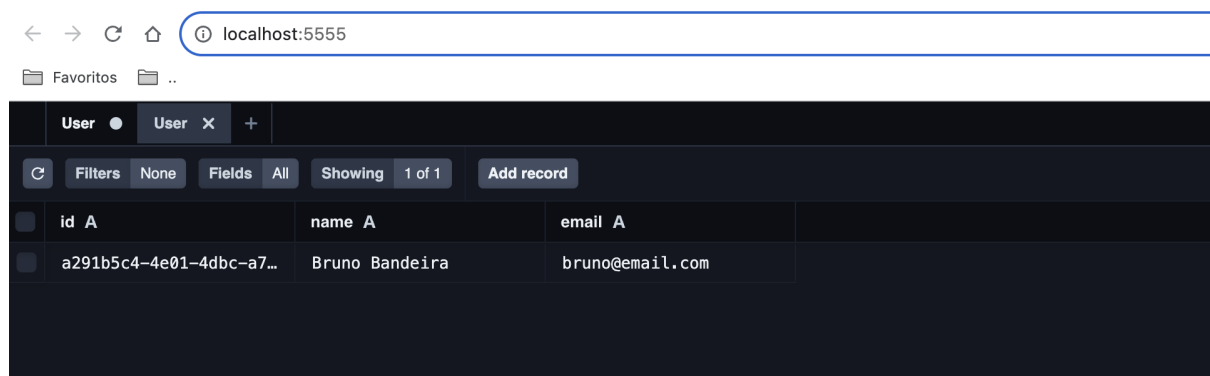
Para testar o código criado, iremos utilizar o Insomnia, criando a rota de register, conforme imagem.



No Insomnia foi criado uma pasta 'Users' e uma requisição do tipo POST para realizar o registro do usuário. A URL da requisição é <http://localhost:3333/users> e no corpo da requisição foi inserido o json abaixo:

```
{
  "name": "Bruno Bandeira",
  "email": "bruno@email.com"
}
```

Para confirmar que a requisição funcionou, consulta a tabela Users, no Prisma Studio.



Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/cbc6b173d7182b549f7b61c69f4a20c031438250

8. USE CASE DE USERS - REGISTER

"Use Case" refere-se à aplicação do padrão de arquitetura de software chamado Clean Architecture (ou Arquitetura Limpa). Clean Architecture foi proposta por Robert C. Martin e visa criar sistemas que sejam independentes de frameworks, testáveis e mantenham uma separação clara de responsabilidades.

O "Use Case" (Caso de Uso) é uma parte importante da Clean Architecture e é responsável pela implementação das regras de negócio da aplicação. Nesse contexto, um "Use Case" representa uma tarefa ou funcionalidade específica que a aplicação deve realizar.

A arquitetura limpa propõe uma estrutura em camadas, cada uma com uma responsabilidade específica. As camadas principais incluem:

- **Entidades:** representam os objetos de negócio e as regras de negócio relacionadas a eles.
- **Use Cases:** implementam as regras de negócio específicas da aplicação. Eles orquestram a execução de operações específicas, manipulando entidades e chamando repositórios.
- **Controladores (Adapters):** adaptam os dados da camada externa (por exemplo, interfaces de usuário ou APIs) para os Use Cases e vice-versa. Os controladores são responsáveis pela comunicação com o exterior da aplicação.
- **Gateways/Repositórios:** tratam da persistência e recuperação de dados. Eles são responsáveis por interagir com o banco de dados ou outros meios de armazenamento.
- **Frameworks e Drivers:** camada externa que contém os detalhes de implementação específicos do framework ou da tecnologia utilizada.

A separação clara dessas camadas permite que a lógica de negócios (contida nos Use Cases) permaneça independente do framework ou da infraestrutura utilizada. Isso facilita os testes automatizados, a manutenção e a evolução da aplicação ao longo do tempo.

Vamos começar removendo a lógica da aplicação, para registro de usuário (comando do Prisma) do controllers e adicionando em um Use-case. Neste caso, vamos criar uma pasta **src/use-cases** e o arquivo **users-register.ts**.

Endereço: src/use-cases/users-register.ts

```
import { prisma } from '@lib/prisma'

// Interface para validar/Esperar os tipos de Entrada
interface RegisterUseCaseRequest {
  name: string
  email: string
}

export async function usersRegisterUseCase({
  name,
  email,
}: RegisterUseCaseRequest) {
  // Verifica se o e-mail é único
  const userWithSameEmail = await prisma.user.findUnique({
    where: {
      email,
    },
  })

  // Error se e-mail do usuário já existe
  if (userWithSameEmail) {
    throw new Error('E-mail already exists.')
  }

  // Criando usuário com Prisma
  await prisma.user.create({
    data: {
      name,
      email,
    },
  })
}
```

O código foi comentado para facilitar o entendimento. A interface RegisterUseCaseRequest é um padrão utilizado na linguagem TS, definindo o tipo de entrada. O arquivo exporta o método usersRegisterUseCase, responsável pela regra de negócio, ou seja, por registrar o usuário no banco utilizando o ORM Prisma.

Agora precisamos ajustar o controller (register.ts) para receber o método usersRegisterUseCase.

Endereço: src/http/controllers/register.ts

```
import { FastifyReply, FastifyRequest } from 'fastify'
import { z } from 'zod'

import { usersRegisterUseCase } from '@use-cases/users-register'

export async function register(request: FastifyRequest, reply: FastifyReply) {
  // Valida os dados recebido com Zod
  const registerBodySchema = z.object({
    name: z.string(),
    email: z.string().email(),
  })
  const { name, email } = registerBodySchema.parse(request.body)

  // Executa a função do Use-Case
  try {
    await usersRegisterUseCase({
      name,
      email,
    })
  } catch (err) {
    // Se ocorrer algum erro
    // Precisaria tratar os erros com uma classe
    return reply.status(409).send()
  }

  // Em caso de sucesso
  return reply.status(201).send()
}
```


Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/e1cb326f9c8083854a8fe0e7459f080ec3019c76

9. REPOSITORY PATTERN

Apenas separar os arquivos não traz uma solução para o problema, mas é preciso ter uma lógica por trás dessa separação de camada. A ideia principal é a criação de uma arquitetura limpa, reutilizável e de fácil manutenção.

O Repository é o responsável por interagir com o banco de dados ou outros meios de armazenamento. Neste caso, nossos código do ORM Prisma estarão na pasta repository, enquanto o use-case será o responsável apenas pela regra de negócio, independente do ORM utilizado.

Endereço: src/repositories/prisma-users-repositories.ts

```
import { prisma } from '@lib/prisma'
import { Prisma } from '@prisma/client'

export class PrismaUsersRepository {
  // Create User
  // Tipando a função com o próprio Prisma - Prisma.UserCreateInput
  async create(data: Prisma.UserCreateInput) {

    // Criando usuário com Prisma
    const user = await prisma.user.create({
      data,
    })

    return user
  }
}
```

Ajustando o use case register para remover o método de inserção do usuário pelo prisma do arquivo.

Endereço: src/use-cases/users-register.ts

```
import { prisma } from '@lib/prisma'
import { PrismaUsersRepository } from '@repositories/prisma-users-repository'

// Interface para validar/Esperar os tipos de Entrada
```

```

interface RegisterUseCaseRequest {
  name: string
  email: string
}

export async function usersRegisterUseCase({
  name,
  email,
}: RegisterUseCaseRequest) {
  // Verifica se o e-mail é único
  const userWithSameEmail = await prisma.user.findUnique({
    where: {
      email,
    },
  })

  // Error se e-mail do usuário já existe
  if (userWithSameEmail) {
    throw new Error('E-mail already exists.')
  }

  // Criando usuário com Repositório do Prisma
  const prismaUsersRepository = new PrismaUsersRepository()

  await prismaUsersRepository.create({
    name,
    email
  })
}

```

A única alteração no arquivo foi a inserção das linhas

```

// Criando usuário com Repositório do Prisma
const prismaUsersRepository = new PrismaUsersRepository()

await prismaUsersRepository.create({
  name,
  email
})

```

Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/af89637ed8a7b0ee9e8d2207c7f2ce28ba6f3a9f

10. INVERSÃO DE DEPENDÊNCIA

Mesmo com a utilização do 'repositories', supondo que tenhamos inúmeros arquivos no use-case e precisarmos trocar o ORM do projeto, ainda assim precisaremos entrar em todos os arquivos do use case e trocar as dependências, isso porque o ORM Prisma ainda está contido no arquivo user-register.ts

```
/ Criando usuário com Repositório do Prisma
const prismaUsersRepository = new PrismaUsersRepository()

await prismaUsersRepository.create({
  name,
  email
})
```

Portanto, utilizaremos o conceito do SOLID para realizar uma inversão de dependências, aplicando neste caso somente o conceito D.

O SOLID é um acrônimo que representa cinco princípios de design de software que visam criar sistemas mais flexíveis, escaláveis e fáceis de manter. Estes princípios foram introduzidos por Robert C. Martin, um renomado autor e consultor de software, e são considerados fundamentais para a prática de programação orientada a objetos e design de código robusto.

- **Princípio da Responsabilidade Única (Single Responsibility Principle - SRP):** este princípio afirma que uma classe deve ter apenas uma razão para mudar. Em outras palavras, uma classe deve ter uma única responsabilidade ou tarefa. Isso promove a coesão, facilitando a manutenção e compreensão do código.
- **Princípio do Aberto/Fechado (Open/Closed Principle - OCP):** o OCP estabelece que uma classe deve estar aberta para extensão, mas fechada para modificação. Isso significa que você deve poder estender o

comportamento de uma classe sem modificar seu código-fonte. Isso é geralmente alcançado por meio de interfaces, herança e polimorfismo.

- **Princípio da Substituição de Liskov (Liskov Substitution Principle - LSP):** este princípio enfatiza que objetos de uma classe base devem ser substituíveis por objetos de suas classes derivadas sem afetar a consistência do programa. Isso garante que as subclasses respeitem o contrato estabelecido pela classe base, evitando comportamentos inesperados.
- **Princípio da Segregação de Interfaces (Interface Segregation Principle - ISP):** o ISP sugere que uma classe não deve ser forçada a implementar interfaces que ela não usa. Em vez de ter interfaces grandes e abrangentes, é preferível ter várias interfaces menores, cada uma atendendo a um conjunto específico de responsabilidades.
- **Princípio da Inversão de Dependência (Dependency Inversion Principle - DIP):** o DIP inverte a direção das dependências em um sistema. Em vez de as classes de alto nível dependerem de classes de baixo nível, ambas devem depender de abstrações. Além disso, as abstrações não devem depender de detalhes, mas os detalhes devem depender das abstrações. Isso promove a flexibilidade e a extensibilidade do código.

A adoção dos princípios SOLID contribui para a criação de código mais modular, extensível e fácil de manter. Eles são amplamente aceitos na comunidade de desenvolvimento de software e são considerados boas práticas no design de código orientado a objetos. Neste momento iremos utilizar o Princípio DIP, ou seja, nosso código não será dependente do ORM Prisma.

Agora, cada use case continuará tendo apenas um método, mas iremos reescrever o use case users-register como uma classe, tendo um construtor e o atributo responsável por receber detalhes do repositório.

```
export class UsersRegisterUseCase {  
  
  constructor(private usersRepository: any) {}  
}
```

A classe completa terá uma interface para definir os dados de entrada, a exportação da classe, um construtor que irá receber o parâmetro do repositório e a função responsável por executar o use case.

Endereço: src/use-case/users-register.ts

```
import { prisma } from '@lib/prisma'  
  
// Interface para validar/Esperar os tipos de Entrada
```

```

interface RegisterUseCaseRequest {
  name: string
  email: string
}

export class UsersRegisterUseCase {

  constructor(private usersRepository: any) {}

  async execute({ name, email }: RegisterUseCaseRequest) {

    // Verifica se o e-mail é único
    const userWithSameEmail = await prisma.user.findUnique({
      where: {
        email,
      },
    })

    // Error se e-mail do usuário já existe
    if (userWithSameEmail) {
      throw new Error('E-mail already exists.')
    }

    // Criando usuário com Repositório (repositories) do Prisma
    (prisma-users-repository)
    await this.usersRepository.create({
      name,
      email
    })
  }
}

```

Agora precisamos ajustar o controller para instanciar o ORM e realizar a chamada do use case.

```

// Prisma - como repositório definido
const usersRepository = new PrismaUsersRepository()

// Use Case - utilizando o repositório Prisma

```

```

                                const    registerUseCase    =    new
UsersRegisterUseCase(usersRepository)

    // Executando o Use Case
    await registerUseCase.execute({
        name,
        email,
    })

```

O código completa será da seguinte forma:

Endereço: src/http/controllers/register.ts

```

import { FastifyReply, FastifyRequest } from 'fastify'
import { z } from 'zod'

import { PrismaUsersRepository } from '@repositories/prisma-users-repository'
import { UsersRegisterUseCase } from '@use-cases/users-register'

export async function register(request: FastifyRequest, reply: FastifyReply) {
    // Valida os dados recebido com Zod
    const registerBodySchema = z.object({
        name: z.string(),
        email: z.string().email(),
    })

    const { name, email } = registerBodySchema.parse(request.body)

    // Executa a função do Use-Case
    try {
        // Prisma - como repositório definido
        const usersRepository = new PrismaUsersRepository()

        // Use Case - utilizando o repositório Prisma
        const registerUseCase = new UsersRegisterUseCase(usersRepository)

        // Executando o Use Case
        await registerUseCase.execute({

```

```

        name,
        email,
    })
} catch (err) {
    // Se ocorrer algum erro
    // Precisaria tratar os erros com uma classe
    return reply.status(409).send()
}

// Em caso de sucesso
return reply.status(201).send()
}

```

Note que a verificação do email ainda está contida no controller. Precisamos remover esse método do controller e atribuir ao use case. Faremos isso na próxima etapa.

Link do github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/74ed70c3aa14a51f4410f37b9abe5a3a94c2bc95

11. INTERFACE DO REPOSITÓRIO USERS

Em programação orientada a objetos (POO), uma interface é uma coleção de métodos abstratos (métodos sem implementação) que uma classe deve implementar. As interfaces fornecem um meio de definir um contrato que as classes devem seguir. Elas não contêm implementações reais dos métodos; em vez disso, deixam essa responsabilidade para as classes que as implementam.

A ideia por trás das interfaces é fornecer um meio de definir um conjunto comum de métodos que várias classes relacionadas podem implementar, independentemente de sua hierarquia de herança. Isso permite que objetos de diferentes classes sejam tratados de maneira uniforme se implementarem a mesma interface.

Vamos criar a interface para implementar a classe User.

Endereço: src/http/repositories/users-repository.ts

```

import { Prisma, User } from '@prisma/client'

export interface UsersRepository {

```

```
findByEmail(email: string): Promise<User | null>
create(data: Prisma.UserCreateInput): Promise<User>
}
```

A interface `UsersRepository` possui os métodos `findByEmail` e `create`, porém não implementa, ou seja, é apenas um contrato, informa o que a classe que implementar a interface deverá possuir.

Agora, iremos criar uma pasta 'prisma' e mover o arquivo 'prisma-users-repository.ts' para dentro da pasta. Além disso, devemos forçar o arquivo a implementar a interface, para isso adicionar o class `PrismaUsersRepository implements UsersRepository`. Por fim, iremos implementar o método `findByEmail` que estava voltando no contrato.

Endereço: `src/repositories/prisma/prisma-users-repository.ts`

```
import { prisma } from '@lib/prisma'
import { Prisma } from '@prisma/client'

import { UsersRepository } from '../users-repository'

export class PrismaUsersRepository implements UsersRepository {

  async findByEmail(email: string) {

    const user = await prisma.user.findUnique({
      where: {
        email,
      },
    })

    return user
  }

  // Create User
  // Tipando a função com o próprio Prisma - Prisma.UserCreateInput
  async create(data: Prisma.UserCreateInput) {

    // Criando usuário com Prisma
    const user = await prisma.user.create({
      data,
```



```

    })

    return user
  }
}

```

No use case de 'register.ts' é preciso excluir a dependência da busca do email do usuário e substituir pelo método implementado na interface.

```
const userWithSameEmail = await this.usersRepository.findByEmail(email)
```

O arquivo completo com as alterações ficará da seguinte forma:

Endereço: src/use-cases/users-register.ts

```

import { UsersRepository } from '@repositories/users-repository'

// Interface para validar/Esperar os tipos de Entrada
interface RegisterUseCaseRequest {
  name: string
  email: string
}

export class UsersRegisterUseCase {

  constructor(private usersRepository: UsersRepository) {}

  async execute({ name, email }: RegisterUseCaseRequest) {

    // Verifica se o e-mail é único
    const userWithSameEmail = await
this.usersRepository.findByEmail(email)

    // Error se e-mail do usuário já existe
    if (userWithSameEmail) {
      throw new Error('E-mail already exists.')
    }

    // Criando usuário com Repositório (repositories) do Prisma
(prisma-users-repository)
    await this.usersRepository.create({
      name,
      email
    })
  }
}

```

```
    })  
  }  
}
```

Link do Github:

https://github.com/brunobandeiraf/API_Fastify_Prisma_Node/commit/7bc62e703e732392fa01223bd6d9278782896d0f

12. USERS - FINDMANY