|  |
| --- |
| **AULA 1** |
| * Crie o **package.json**; * Crie uma pasta **src** e dentro crie um arquivo **server.ts**; * Instale **npm i typescript @types/node tsx tsup -D;** * Execute o **npx tsc --init**, e troque o **target** para **es2020**; * Instale **npm i fastify;** * Crie um arquivo **app.ts**; * Em **app.ts** importe o **fastify**, crie uma constante **app** com uma instância do **fastify**. Exporte a constante; * Dentro do arquivo **server.ts** importe a instância **app** e aplique o método **listen** que recebe como parâmetro um objeto uma **port: 3333** e um **host: “0.0.0.0”**. Concatene com um **then** e execute como **callback** um **console.log(‘HTTP server running’)**; * Crie o arquivo **.gitignore**. Ignore **node\_modules** e **build;** * No **package.json** crie o **script** para executar o server **“dev”: “tsx watch src/server.ts”**; * Crie um **script** de **“build”: “tsup src --out-dir build”**. Crie outro **script**, **“start”: “node build/server.js”** |

|  |
| --- |
| **AULA 2** |
| * Crie na **root** um arquivo **.env**. Coloque uma variável **NODE\_ENV=dev.** Crie o arquivo **.env.example**, com a variável **NODE\_ENV**. Adicione o **.env** no **.gitignore**; * Instale **npm i dotenv.** Crie uma pasta chamada **env** em **src** e crie dentro dele um arquivo chamado **index.ts**. Importe **‘dotenv/config’** dentro do arquivo; * Instale **npm i zod** e importe **z** de **zod** no arquivo; * Crie a variável **envSchema** que recebe um **z.object({})**. Defina **NODE\_ENV** como **enum** com os valores **dev, test** ou **production**, e adicione um valor **default** como **dev.** Adicione também a **PORT: z.coerce.number().default(3333);** * Faça a validação com **envSchema.safeParse(process.env)**, desestruture **success, error** e **data**. Crie uma condicionante caso não tenha sucesso retorne **console.error(error.format(‘Invalid environment variables’))** dê tbm um **throw new Error(‘Invalid environment variables’). Exporte const env = data**; * No **app.listen** substitua a **port**; |

|  |
| --- |
| **AULA 3** |
| * Instale o ESlint **npm i eslint @rocketseat/eslint-config -D**; * Crie um arquivo **.eslintrc.json.** Dentro do arquivo abra colchetes e digite **“extends”**: [“**@rocketseat/eslint-config/node”];** * Em **tsconfig.json** procure por **path** e jogue um objeto **“@/\*”:[”./src/\*”]** |

|  |
| --- |
| **AULA 4** |
| * Instale **npm i prisma -D.** Depois digite **npx prisma init**; * No arquivo **schema.prisma** crie um **model User{}.** Para alterar o nome da tabela **@@map(‘users’);** * Para adicionar o **id** digite o comando **id String @id @default(uuid())**. Adicione também um **name String** e um **email String @unique**; * Digite **npx prisma generate**; * Instale **npm i @prisma/client**; * No arquivo **app.ts** importe **import {PrismaClient} from ‘@prisma/client’**. Para criar a conexão com o banco basta instanciar o valor importado como uma classe, **const prisma = new PrismaClient()**; * Faça um **prisma.user.create({})**que recebe um **data**, que é um objeto que receberá as linhas da tabela como chaves; |

|  |
| --- |
| **AULA 5** |
| * No terminal digite **docker run --name** **api-solid-pg -e POSTGRESQL\_USERNAME=docker -e POSTGRESQL\_PASSWORD=docker -e POSTGRESQL\_DATABASE=apisolid -p 5432:5432 bitnami/postgresql**; * Va no arquivo **.env** e altere o **DATABASE\_URL** passando o usuário, a senha e o nome do banco de dados; * Rode o comando **npx prisma migrate dev**; * Rode **npx prisma studio** que abre uma intefarce no browser; |

|  |
| --- |
| **AULA 6** |
| * Crie um arquivo chamado **docker-compose.yml**. E adicione:      * Utilize **docker compose up** e **docker compose stop** para executar ou parar o container. Lembre que **docker rm** deleta os arquivos; |

|  |
| --- |
| **AULA 7** |
| * No arquivo **schema.prisma** crie dois novos **models** **Gym** e **CheckIn**. Todos vão ter **id String @id @default(uuid())**. No **Gym** coloque **title** e **description**, neste ultimo a **String?** pois poderá ser opcional, **phone** como opcional string, **latitude** e **longitude** como **Decimal**, renomeie **Gym** por gyms; * Renomeie **CheckIn**  para check-ins. Os campos serão **created\_at DateTime @default(now())**, **validated\_at** **DateTime?;** * No **User** coloque um **password\_hash String**, **created\_at DateTime @default(now());** * Rode o **npx prisma migrate dev**; |

|  |
| --- |
| **AULA 8** |
| * Para criar um relacionamento entre tabelas, na tabela **CheckIn** adicione o campo **user User** e dê um auto-save para que o prisma complemente; * Faça o mesmo relacionamento para **gym**; |

|  |
| --- |
| **AULA 9** |
| * No arquivo **app.ts** crie uma rota **POST** com o endereço **‘/users’**, e uma **callback** assíncrono; * Utilize o **zod** para fazer a validação em uma constante chamada **registerBodySchema**, ele receberá name, email e password; * Faça um **parse** e desestruture os itens; * Crie no root uma pasta chamada **lib** e adicione o arquivo **prisma.ts**. Aqui exporte uma instância de **PrismaClient**; * De volta a arquivo **app.ts** chame **await prisma.user.create({data:{}})** com os valores desestruturados; * Retorne um reply.status(201).send(); * No arquivo **prisma.ts**, chame dentro do parâmetro um objeto com a chave **log** e diga que se **env.NODE\_ENV** estiver em ‘dev’ então **[‘query’]** do contrário um array vazio; |

|  |
| --- |
| **AULA 10** |
| * Crie uma pasta chamada **http** em **src**, dentro dele crie outra pasta chamada **controller** e crie um arquivo chamada **register.ts**; * Dentro do arquivo crie uma função chamada **register** que receber o **callback** do método **post** criado anteriormente, e exporte ele e devolva para **app.ts**; * Nos parâmetros **request, reply** tipe com **FastifyRequest** e **FastifyReply**; * No lugar do **callback** chame a função **register**; * Dentro de **http** crie um arquivo chamado **routes.ts.** Crie uma função assíncrona chamada **appRoutes(app),** com a tipagem **FastifyInstance.** Adicione a rota de **post** já criada; * No **app.ts** faça um **app.register(appRoutes);** |

|  |
| --- |
| **AULA 11** |
| * Instale **npm i bcryptjs** e o **@types/bcryptjs** **-D**; * Importe **{hash} from ‘bcryptjs’** no arquivo **register.ts**; * Dentro da função do arquivo crie uma constante chamada **password\_hash** = **await hash(password, 6)** e substituía no lugar do **password** que está sendo passado para a tabela; * Crie uma constante **userWithSameEmail** = **await prisma.user.findUnique({where: {email}).** Se a constante der **true** dê um **reply** **409;** |

|  |
| --- |
| **AULA 12** |
| * No **src** crie uma pasta chamada **use-cases**. Dentro crie um arquivo **register.ts**; * Dentro do arquivo importe uma função assíncrona chamada **registerUseCase()** e coloque toda a funcionalidade da função **register;** * Crie uma **interface RegisterUseCaseRequest** e diga que name, email e password são do tipo strings, passe como parâmetro da função criada; * Observe que os parâmetros vem em um objeto para dizer que a tipagem é única entre eles; * No lugar do **reply** que ativa quando existir o mesmo email duas vezes dê um **throw new Error(‘Email already exists’);** * Quando for chamar a função nova no lugar da antiga passe por um **try/catch** e no erro mande o **reply 409;** |

|  |
| --- |
| **AULA 13** |
| * No **src** crie uma pasta chamada **repositories**, crie o arquivo chamado **prisma-users-repository.ts;** * Exporte uma **class PrismaUsersRepository;** * Crie o método **async create(data)** e ele receberá a parte do **register.ts** responsável por se comunicar com o banco de dados; * O parâmetro **data** recebe como tipagem **Prisma.UserCreateInput**. Dentro do método trate o **data** como um objeto sem valor. Insira toda a execução dentro de uma constante **user** e dê um **return user;** * No **register.ts** crie uma constante **prismaUsersRepository = new PrismaUsersRepository()**. E chame o método create; |

|  |
| --- |
| **AULA 14** |
| * No arquivo **register.ts**, crie a classe **RegisterUseCase** que receberá como o método **execute** toda a função que estava sendo desempenhada. * Crie um **constructor** cujos parâmetros são **private usersRepository: any** e use ele no **this**, altere o **prismaUsersRepository**  para **this.usersRepository.** * No arquivo **controllers/register.ts** dentro do **try/catch** instancie o **prismaUsersRepository** e o **usersRepository** e passe o prisma como parâmetro; * Altere o próximo código para **await registerUseCase.execute({})**; |

|  |
| --- |
| **AULA 15** |
| * Dentro da pasta **repositories** crie um arquivo chamado **user-repository.ts**; * Faça um **interface UsersRepository,** que diga que **create(data: Prisma.UserCreateInput) : Promise<User>** e aplique essa tipagem em no caso de uso (**RegisterUseCase**); * Dentro da classe **PrismaUsersRepository** utilize o comando **implements UsersRepository;** * Na pasta **repositories** crie uma nova pasta **prisma** e adicione o arquivo que contenha a classe **PrismaUsersRepository**, e faça as refatorações necessárias nas importações; * Na **interface Users** crie um **findByEmail(email: string): Promise<User|null>.** Agora no repositório de classe do prisma, crie o método **findByEmail;** * Na constante **userWithSameEmail = await this.usersRepository**.**findByEmail(email);** |

FALTA A AULA DE ERROS

|  |
| --- |
| **AULA 16** |
| * Instale o **npm i vitest vite-tsconfig-paths -D;** * Crie no root um arquivo chamado **vite.config.ts**. exporte por **default defineConfig({plugins: [tsconfigPaths()]})**; * Crie um script **“test”:”vitest run”**  e **“test:watch”:”vitest”**; * Na para **user-cases**  crie o arquivo **register.spec.ts**; |
| **AULA 17** |
| * No arquivo com a extensão **spec** importe **describe** e **it;** * Crie o **describe(‘Register Use Case’, () =>{});** * Dentro, faça o primeiro teste **it(‘should hash user password upon registration’, ()=>{}).** Como **callback** inicie instanciando uma constante **const prismaUseRepository**  e outra **registerUseCase** que recebe a anterior como parâmetro. Faça um **await registerUseCase.execute** como valores exemplo; * Dentro do **register.ts** do **user-case** faça com que o método **execute** seja adicionado a uma constante **user** e retorne **user**, retorne com um objeto**;** * Faça a **interface RegisterUseCaseResponse** como uma **user: User**, depois diga que o resultado do método **execute** retorna uma promisse do tipo da tipagem feita agora. * Volte para o **callback** do teste, e destruture **user**. * Crie uma constante chamada **isPasswordCorrectlyHashed** e chame um **await compare(password, user.password\_hash);** * Delete o parâmetro da instância do **RegisterUseRepository** e crie um objeto com os métodos **create** e **findByEmail;** |

|  |
| --- |
| **AULA 18** |
| * Dentro da pasta **repositories** crie a pasta **in-memory** e crie o arquivo **in-memory-users-repository**.**ts;** * Crie uma classe **InMemoryUsersRepository implements UsersRepository**; * Transforme os métodos em assíncronos, e substitua com o valor com o banco falso criado nos testes; * Crie uma variável **public items: User[] = [];** * Faça com quem o método **create** adicione o user criado no array; * No método **findByEmail** faça um find que busque em **this.items** o **item.email** === **email.** Se não encontrar retorne nulo, se encontrar retorne o usuário desse email; * Instancie essa classe na constante **usersRepository** no arquivo de testes. * Crie o teste **should not be able to register with same email twice**. Diferente do anterior, além do que foi feito (retirar a parte de análise do hash) crie uma constante **email** com ‘johndoe@example’. No lugar de chamar o valor duas vezes, já gere um **expect** que recebe o segundo registro como **callback** e concatene com um **rejects.toBeInstanceOf(UserAlreadyExistsError)**; * Faça um teste **should be able to register**. Crie um usuário e diga que espera que ele seja **toEqual(expect.any(String));** * Crie um script **“test:coverage”:”vitest run --coverage"**; * Coloque a pasta gerada, **coverage**, no **.gitignore**; * Instale o **npm i @vitest/ui -D;** * Crie o script **“test:ui”:”vitest run – ui;** |

|  |
| --- |
| **AULA 19** |
| * Crie o caso de uso **authenticate.ts**; * Crie a classe **AuthenticateUseCase.** Crie um **constructor** com o parâmetro **userRepository** e com a tipagem **UserRepository.** Crie o método assíncrono **execute** e deixe ele em branco; * Crie a tipagem de entrada **AuthenticateUseCaseRequest** e de saída **AuthenticateUseCaseResponse**. A primeira interface recebe **email** e **senha** tipo string. * Diga que o **execute** vai receber como parêmetros **email** e **senha** com as respectivas tipagens, e que vai devolver uma **promise** do tipo response; * Faça uma constante **user = await this.userRepository.findByEmail(email)**; * Crie na pasta de erros um arquivo chamado **invalid-credencial-errors**; * Crie um erro chamado **InvalidCredentialsError**, cuja mensagem será: “Invalid crendetials.”; * De volta a classe que estava sendo feita faça uma condicional caso **user** seja falso, jogando o novo erro criado; * Crie uma variável chamada **doesPasswordMatches** e compare a senha sem ter feito o hash dada pelos parâmetros, com a senha do usuário pego na constante **user.** Caso não bata, retorne o **InvalidCredentailsError**; * Feito tudo, retorne dentro de um objeto o **user**; * Agora diga que o Response do método é um usuário; |

|  |
| --- |
| **AULA 20** |
| * Crie o teste **authenticate.spec.ts**; * Crie o primeiro teste **should be able to authenticate.** Instancie **InMemoryUseRepository()** e adicione como parâmetro de **new AuthenticateUseCase().** Dê a constante que recebe a instância final o nome de **sut;** * Crie um usuário através do método create; * Dê um método execute e espere que o **user.id** seja igual a qualquer **String**; * Crie o teste **should not be able to authenticate with wrong email**. Pode rodar já o expect com o método **execute**  de callback, esperando que ela rejeite e o erro seja uma instância de **InvalidCredentialsError**; * Crie um teste **should not be able to authenticate with wrong password**. Crie um usuário com o **create** e depois com o **execute** como callback repita o esperado do resultado anterior; * Dentro da pasta **controllers** crie o arquivo **authenticate.ts**; * Copie o arquivo do **register.ts** e faça a fatoração necessária**.** No novo erro o status é 400; * Crie uma rota **post** para a url **sessions**; |

|  |
| --- |
| **AULA 21** |
| * Dos testes, retire a parte da instanciação das classes e adicione em um **beforeEach;** * Crie as constantes globalmente e tipe eles, e ai dentro do **beforeEach** instancie as variáveis; |

|  |
| --- |
| **AULA 22** |
| * Dentro da pasta **use-cases** crie a pasta **factories** e crie um arquivo chamado **make-register-use-case.ts**; * Crie e exporte a função **makeRegisterUseCase** e dentro coloque as instanciações feitas no **try/catch** do **controllers/register.ts** e retorne **registerUseCase**; * Dentro do **register.ts** em uma constante **registerUseCase** chame e execute a função; * Repita o processo para o **authenticate.ts**; |

|  |
| --- |
| **AULA 23** |
| * Crie um caso de uso chamado **get-user-profile.ts.** Copie o arquivo do **authenticate.ts** para agilizar; * Chame a classe de **GetUserProfileUseCase**. Faça as fatorações de nome necessárias, na interface do request vamos receber apenas **userId: string**. Coloque o **userId** como parâmetro do método **execute**; * Vá para o arquivo **user-repository.ts** e adicione outro método na interface, o **findById(id: string)**; * No arquivo **in-memory-users-repository.ts** implemente o método **findById**, reaproveite o método de encontrar pelo email, apenas substituindo o parâmetro; * Crie o error **resource-not-found-error.ts** com a mensagem ‘**Resource not found**’; * Dentro do **throw** lance o novo erro; * Crie o teste **get-user-profile.spec.ts.** Aproveite o teste do authenticate, crie um teste chamado **should be able to get user profile**  bem como faça um **should not be able to get a user profile with wrong id;** |

|  |
| --- |
| **AULA 24** |
| * Crie o caso de uso de **check-in.ts**, para facilitar copie o **authenticate.ts**, e faça a refatoração necessária com o nome **CheckInUseCase**; * Na interface vamos receber o **userId** e **gymId** e devolver **checkIn: CheckIn**; * Para a interface do repositório crie o **check-in-repository.ts**; * Crie um método **create(data: Prisma.CheckInUncheckedCreateInput): Promise<CheckIn>;** * De volta ao caso de uso, o construtor ira receber como parâmetro um **checkinrepository;** * No método **execute** coloque os parâmetros corretos; * Crie a constante **checkIn = await this.checkInRepository.create({}).** Receba os parâmetros **userId** e **gymId,** e retorne o **checkIn** como objeto; * Dentro do **in-memory** crie o **in-memory-check-in-repository.ts**, copie o do user para facilitar, e faça as alterações de nomeação necessária; * No método **create** crie a constante **checkIn** que recebe os parâmetros que a tabela **checkIn** pede. No campo **validated\_at** crie um ternário onde se o campo estiver preenchido retorne **new Date(date.validated\_at)** se não **null**; * Faça um teste **check-in.spec.ts**, copie qualquer um e faça as alterações necessárias de nomenclaturas. E peça o teste **should be able to check in;** |

|  |
| --- |
| **AULA 25** |
| * Vá para o arquivo **check-in.spec.ts** e duplique o teste que já existe nele. Altere o nome para **should not be able to check in twice in the same day**. Para isso, faça um primeiro **check-in**, e depois no **expect** execute um callback com outro pedido de **check-in** e espere que o resultado rejeite e retorne uma instância de um erro **Error**; * Para que seja possível testar com datas fictícias ou **mock dates** importe do vitest o **vi** no **beforeEach** chame o método **vi.useFakeTimers()** e no **afterEach** o método **vi.useRealTimers()**. Dentro do teste chame **vi.setSystemTime(new Date())** como parâmetro de Date como o valor que quiser; * De volta para o **CheckInRepository** crie um método chamado **findByUserIdOnDate** que recebe **userId** e **date** e retorna uma promessa que pode ser um **CheckIn** ou **null**; * Vá para o **in-memory-check-in-repository** e crie o método assíncrono **findByUserIdOnDate**. Crie uma constante **checkInOnSameDate** que recebe o resultado de um **find** dentro da array **items** onde busca se **user\_id** é igual ao **userId** do parâmetro apresentado. Se **checkInOnSameDate** der falso retorne nulo. Retorne a constante; * Vá para o **check-in** do caso de uso e dentro do método **execute** crie a constante **checkInOnSameDate** que recebe **await.this.checkInsRepository.findByUserIdOnDate(userId, new Date()).** Se o resultado da constante for verdadeiro, retorne um error; * De volta ao arquivo de testes, copie o teste fazendo um **should be able to check in twice but in different days**. A diferença para o teste anterior é que antes de executar o segundo **check in** utilize o **vi.setSystemTime** para uma data diferente da anterior, além de que receba o valor do segundo check in em uma constante **{checkIn}** e dê um **expect** que checkIn.id seja igual a **expect(any(String));** |

|  |
| --- |
| **AULA 26** |
| * Instale **npm i dayjs**; * Dentro do repositório de check in da memória crie, dentro do método **findByUserIdOnDate**, uma constante chamada **startOfTheDay = dayjs(date).startOf(‘date’).** Crie outra constante chamada **endOfTheDay** onde o **dayjs** recebe o método **endOf(‘date’);** * Na função **find** delete a condicional de igualdade de usuários e crie um corpo de função. Crie uma constante **checkInDate = dayjs(checkIn.created\_at)**. Crie outra constante **isOnSameDate = checkInDate.isAfter(startOfTheDay) && checkInDate.isBefore(endOfTheDay)**. Retorne **checkIn.user\_id && isOnSameDate;** * Na pasta **repositories** crie **gym-repository.ts** crie a interface dessa classe apenas com **findById** que retorna uma promise **Gym** ou nulo. * Crie o arquivo **in-memory-gyms-repository.ts.** Copie qualquer outro repositório para facilitar mas faças as mudanças de nomes necessárias. * No arquivo de casos do **checkIn,** o **request** na interface agora precisará de **userLatitude** e **userLongitude** que são **numbers**. E o **constructor** agora também receberá o **GymsRepository.** Crie uma constante chamada **gym** que recebe **await this.gymsRepository.findById(gymId)**, caso **gym** seja falso retorne um **throw new ResourceNotFoundError();** * Nos testes do **checkIn** agora será preciso um **gymsRepository.** Além disso, é preciso criar uma academia, como o método ainda não foi implementado faça por push; |

|  |
| --- |
| **AULA 27** |
| * Nos testes do **checkIn** copie o teste que observa a possibilidade de realizar o **checkIn** e altere para **should not be able to check in on distant gym**. Crie uma **gym** nova e pegue uma latitude e longitude aleatória superior a 100m. Faça um **expect** com um callback que tente fazer o usuário dar **checkin** na academia. Espere que dê um erro da instância **Error**; * Vá para o caso de uso do **check-in**. Adicione como parâmetros do **execute** a latitude e longitude do usuário; * Crie uma pasta na src chamada **utils**. Crie dentro um arquivo chamado **get-distance-between-coordinates.ts;** * Exporte uma interface chamada **Coordinate** que recebe **latitude** e **longitude** como **number**. Exporte, também uma função **getDistanceBetweenCoordinates**  que recebe como parâmetros **from** e **to** e ambos são **Coordinates.** Adicione a função de cálculo; * De volta ao caso de uso do **check-in** crie uma constante **distance** que recebe a função que foi exportada. Passe como coordenadas as do usuários e as do gym. Faça, depois, uma condicional onde se a **distance > MAX\_DISTANCE\_IN\_KILOMETERS,** crie a constante disso como 0.1, dispare um erro; |

|  |
| --- |
| **AULA 28** |
| * Copie o caso de uso do **register** e faça um **create-gym**. Faça a refatoração de nomes chamando **CreateGymUseCase**. Altere a interface do request para as linhas necessárias para criar a academia. No retorno passe **gym: Gym**. * No **gyms-repository.ts** adicione o método **create** que recebe **data** tipado com **GymCreateInput** que retorna uma promise **Gym;** * Crie o método create no **create-gym;** * De volta ao caso de uso corrija agora para que seja utilizado o método **create;** * Crie o teste **create-gym.spec.ts,** copie o register para facilitar. E faça a refatoração necessária. Faça o teste **should be able to create gym.** Primeiro crie uma gym; * Utilize o novo método criado para alterar os lugares o **gym** estava sendo criado por **push**; * Crie dois Erros novos **max-distance-error** e **max-number-of-check-ins-error.** |

|  |
| --- |
| **AULA 29** |
| * Crie o caso de uso **fetch-user-check-ins-history.ts.** Crie também o teste. Utilize o modelo do **check-in.** Faça as refatorações por nome necessárias. O nome da class é **FetchuserCheckInsHistoryUseCase**. * A interface de request é apenas o **userId,** a de resposta é **checkIns: checkIn[];** * Na interface do **checkInRepository** adicione o método **findManyByUserId** cujo parâmetro é uma **string userId** e o resultado é uma **promisse** de **CheckIn[];** * No **in-memory-check-in-repository** crie o método assíncrono. Dê um **return this.items.filter((item) => item.user\_id === userId);** * Na class **Fetch...** dentro do **execute** crie uma constante **checkIns** que vai fazer utilizar o método **findManyByUserId** o método retorna o checkins dentro de um objeto; * No teste, copie o teste do **check-in, should be able to fetch check-in history**. Troque o **sut** pelo **Fetch...** Faça dois **check-in** em duas academias academia.Passe um **userId** para o **await sut.execute({}).** Dê um expect com o método **toHaveLength(2)**. Espere também que **toEqual([expect.objectContaining({gym.id: gym\_01}, {gym.id: gym\_01}]);** * Crie outro teste onde **should be able to fetch paginated check-in history.** Pegue o método que cria o check-in e faça rodar um **loop** com mais de 20 valores. Se no método **execute** adicionar o parâmetro **page: 2**, o retorno deve ter tamanho 2, e as gyms retornadas tem que ser a 21 e a 22; * Vá par ao **interface do CheckInRepository** e no método **findMany** receba como segundo parâmetro **page: number**; * No **in-memory** do **check-in** receba a **page** como parâmetro. No retorno dê um **slice((page-1) \* 20, page \* 20);** |

|  |
| --- |
| **AULA 30** |
| * Crie um novo caso de uso **get-user-metrics**. Copie o **fetch** para facilitar. Interface request é só **userId**, e response só **checkInsCount: number**; * No repositório do **checkIn** crie um método **countByUserId** que recebe a **string userId** e retorna uma promessa de **number**; * No repositório **in-memory** crie o método **countByUserId**, a função retorna um filter através do **userId** e concatena com um **length**; * O **execute** do caso de uso é **checkInsCount** que recebe o método criado e retorna a constante como um objeto; * Crie o teste, Copie o **fetch** para facilitar. **Should be able to get check-ins count from metrics**. Crie dois checkIns e espere que checkInsCount seja igual a 2; |

|  |
| --- |
| **AULA 31** |
| * Crie um caso de uso chamado **search-gyms.ts**. Utilize o padrão do **create-gym.ts** para facilitar. Recebe apenas a **query: string**, **page: number.** Retorna um **gyms: Gym[];** * No repositório do de interface do **gym** crie um novo método **searchMany** lembre dos parâmetros e dos retornos, principalmente em, aqui, tratar o retorno como uma promessa; * No próprio repositório crie o método. Dê um return de um **filter** que inclua a **query**. No resultado dê **slice** igual ao anterior; * Crie o teste. Utilize o modelo do **fetch...**. Crie o teste **should be able to search gyms**. Crie duas academias através do **gymsRepository.create()**. Realize o método **execute.** Espere que o resultado seja 1, com o título especificado na query. Crie um segundo teste, **should be able to fetch paginated gyms search**. Crie várias academias; |

|  |
| --- |
| **AULA 32** |
| * Crie o caso de uso **fetch-nearby-gyms**. Utilize a estrutura do **search-gyms**. O request vai pedir, como **number**, **userLatitude** e **userLongitude**, o response continuará igual; * Crie um método na interface e no repository. O nome será **findManyNearby**, para receber será diferente, crie e exporte uma interface **FindManyNearbyParams** indicando **latitude** e **longitude** como **numbers**. No parâmetro do método coloque **params** e tipe ele com a interface criada, o retorno ainda será um array de **gyms**; * Vá implementar o método no **InMemory.** De um retorno que filtra, crie uma constante **distance = getDistanceBetweenCoordinates** e preencha seus parâmetros, e por fim retorne se a distância é menor que 10km; * Crie o teste. Utilize o **search-gyms** para facilitar. **Should be abe to fetch nearby gyms**. Cadastre duas academias, uma perto e uma longe. Espere o retorno de um array de um valor cujo objeto interno tenha o título **Near Gym;** |

|  |
| --- |
| **AULA 33** |
| * Crie o caso de uso **validate-check-in.ts**. Para facilite copie o **check-in.ts.** Irá receber um **checkInId** como string. No **check-in-repository** crie o método **findById** que recebe um id string e retorna uma promessa de checkin. No inmemory crie o método. Crie uma constante **checkin** que recebe um **find** procurando o id. Se não encontrar retorne nulo, se encontrar retorne o **checkin**; * No caso de uso quando aplicar o novo método, se o **checkin** for encontrado salve o atributo dele, **validated\_at** como **new Date()**; * De volta ao **repository** crie um método **save** que recebe um **checkIn: CheckIn** e o retorno é uma **promise** do tipo **CheckIn.** No **inmemory** implemente o método **save,** crie uma constante **checkInIndex** e utilize um **findIndex** para procurar onde o **item.id** é igual ao **checkIn.id.** Se o resultado for maior que zero **this.items[checkInIndex] = checkIn**, se não, retorne o **checkIn;** * No **execute** volte e passe o **checkIn** como parâmetro do método **save**; * Crie o teste, utilizando **check-in** como modelo. Deixe apenas o primeiro teste e transforme em **should be able to validate check-in**. Crie um **createdCheck-in** dentro de uma variável **checkIn**. Faça o **sut.execute** e jogue ele em uma constante **{checkIn}**. Espere que o retorno de **checkIn.validated\_at** seja qualquer **Date**. Espere, também **checkInsRepository**.**items[0].validated\_at** seja igual a qualquer **Date;** * Faça um segundo teste, **should not be able to validate an inexistente check-in**. Faça um **execute** de qualquer id já em callback do **expects**. Ele deve rejeitar e ser uma instância de **ResourceError;** |

|  |
| --- |
| **AULA 34** |
| * No arquivo de testes do **validate-check-in** crie o teste **should not be able to validate the check-in after 20 minutes of its creation**. Crie um **vi.setSystemTime(2023,0,1,13,40)**. Faça uma constante **createdCheckIn** criando um checkin em uma academia. Sete uma nova data utilizando **vi.advancedTimersByTime(1000 \* 60 \* 21).** Dê um expect chamando um **callback** do **sut.execute** e diga que deverá rejeitar e esperar um **Error**; * No arquivo do caso de uso, após a primeira condicional crie uma constante **distanceInMinutesFromCheckInCreation** **= dayjs(new Date()).diff(checkIn.created\_at, ‘minutes’)**. Crie uma condicional onde se a distância for maior que 20 dê um **throw error**; * Crie um error **LateCheckInValidateError;** |

|  |
| --- |
| **AULA 35** |
| * No repositório de **user** do prisma, implemente o **findById.** Que é igual ao **findByEmail** só que utilizando o **id** como parâmetro; * Crie o arquivo **prisma-check-in-repository**. Ele importa o **CheckInsRepository.** Crie uma classe com o nome do arquivo que implementa a **interface** que foi importada. Exporte todos os **métodos**. No **findById**, crie uma constante **checkIn** e dê um **await prisma.checkIn.findUnique({where: id})** e retorne o **checkIn**. No método **create**, faça uma constante **checkIn** que recebe o método **create** e como parâmetro um objeto **{data}** e retorna o **checkIn.** Para o método **save**, crie a constante, e dê um **update** que recebe dois parâmetros, **where** que recebe um objeto **id: checkInId** e **data: checkIn,** retorne **checkIn**. O método **countByUserId**, crie uma constante **count** que e dê um **count** onde user\_id: userId, e retorne **count.** No método **findManyByUserId** crie uma constante **checkIns** e dê um **findMany** onde o **user\_id: userId**, também recebe **take: 20** e **skip: (page -1) \* 20** e retorne os **checkIns**. No ultimo método, copie as constantes do **dayjs startOfTheDay** e **endOfTheDay**. Crie uma constante **checkin** e busque o **findfirst()** onde **user\_id:userId** e **created\_at: {gte: startOfTheDay.toDate(), lte: endOfTheDay.toDate().** E retorne o **checkIn;** |

|  |
| --- |
| **AULA 36** |
| * Crie o repositório do prisma para o **gym.** O método **findById** recebe uma const **gym** onde se procura um **findUnique** pela igualdade do id. No método **create** faça a constante **gym** e crie com o parâmetro **data**, retorne o **gym.** No **searchMany**, crie a constante **gym** e o peça para o prisma fazer um **findMany** onde **title: {contains: query}** faça também a paginação **take 20** e **skip (page -1)\*20.** Para o último método, crie a constante **gyms** **= await prisma.$queryRaw<Gyms[]>`SELECT \* FROM gyms WHERE preencha com a fórmula** **<= 10`,** e retorne **gym** |

|  |
| --- |
| **AULA 37** |
| * Entre os **controllers** crie um arquivo **profile.ts**. Copie o **authenticate** para facilitar. * Crie uma rota **get** com a rota **‘/me’** que chama o controller **profile.ts;** * Instale **npm i @fastify/jwt;** * No arquivo **app.ts** antes do registro das rotas, faça um **app.register(jwt)**, como segundo parâmetro passe um objeto contendo um **secret;** * Crie no .**env** um arquivo **JWT\_SECRET=ignitenode03**. Atualize o **env.example**. Vá no arquivo que possui o **envSchema** e diga que **JWT\_SECRET** é uma string. * D volta no **app.ts** coloque o valor de **secret** como **env.JWT.** * Vá para o **authenticate.ts** no **await** que executa o login do usuário, salvo o resultado em uma constante **{user}.** Abaixo, crie uma **const token** * Suba o **reply** para dentro do **try** e envie o token como um objeto através do **send**; * Dentro do **profile.ts** crie uma **await request.jwtVerify().** * Crie uma pasta **@types** e um arquivo **fastify-jwt.d.ts** e faça um * Crie a constante **getUserProfile = makeGetUserProfile().** Depois faça uma constante {user} **await getuserprofile.execute({userId: request.user.sub }).** Dê um retorno 200 com o {user} no send; * Dentro de http crie uma pasta **middlewares**, e um arquivo chamado **verify-jwt.ts.** Dentro exporte uma função VerifyJWT que recebe **request** e **reply**, dentro coloque um **try/catch**, na tentativa coloque o **await request.jwtVerify()** e no **catch** dê um reply 401 com a mensagem Unauthorized; * Apague o verify() do profile, e na rota coloque como segundo parâmetro {onRequest: [verifyJWT]}; |

|  |
| --- |
| **AULA 38** |
| * Dentro da pasta **prisma** crie a pasta **vitest-environment-prisma**. Acesse esta pasta e dê um **npm init -y** dentro dela. Crie outro arquivo chamando **prisma-test-environment.ts** e jogue no **main** do **package.json** da pasta; * No arquivo **prisma-test** importe {**Environment**} **from ‘vitest’.** Dê um **export default <Environment>**, dê o **name: prisma**, dê um **async setup(){ console.log(‘Executou’) return {teardown(){}}};** * Vá em **vite.config** e adicione a chave **test:{}** com o valor **environmentMatchGlobs: [[‘src/http/controllers/\*\*’, ‘prisma’]];** * Com o terminal dentro da pasta do **prisma** dê um **npm link**. Volte para a src e faça um **npm link vitest-environment-prisma;** |

|  |
| --- |
| **AULA 39** |
| * Crie um script **“test:2e2”: “vitest run --dir src/http”;** * No script dos testes unitários coloque um **--dir src/use-cases**; * Instale o **npm i -D npm-run-all;** * Faça o script **“test:create-prisma-environment”:”npm link ./prisma/vitest-environment-prisma”,** e **“test:install-prisma-environment”:”npm link vitest-environment-prisma”.** Crie o **“pretest:2e2”: “run-s test:create-prisma-environment test:install-prisma-environment”;** |

|  |
| --- |
| **AULA 40** |
| * Dentro do **prisma-test-environment**. No **setup()** crie uma constante **schema** e receba um **randonUUID()**. Crie uma função, fora das configurações chamada **generateDatabseURL** que recebe como parâmetro **schema**. Faça uma condicionante onde se não existir **databaseURL** dispare um erro “Please provide DATABASE\_URL environment variable. Crie uma constante **url** = **new URL(**e passe o process.env.DATABASE\_URL**).** Acesse o método **searchParams** de **url** e dê um **set(‘schema’,schema)** e return **url**.**toString()**; * Novamente no **setup** crie uma constante **databaseURL** que recebe a função criada acima. Sobrescreva **process.env.DATABASE\_URL** com a **databaseURL.** Execute **execSync(‘npx prisma migrate deploy’);** * Fora das configurações criem uma instância nova do prisma. No **teardrop** crie um **await prisma.$executeRawUnsafe(‘DROP SCHEMA IF EXISTS “${schema}’” CASCADE).** No final dê um **await.prisma.$disconnect();** |

|  |
| --- |
| **AULA 41** |
| * Instale **npm i supertest -D** e **npm i @types/supertest -D;** * Importe o **request** do **supertest**. * No **describe** dos testes chame de **Register (e2e).** Dentro faça um único teste, **should be able to register**. Dê um **const response = await request(app.server).post(‘/users’).send({name:, email:, password:,})**. Espere que **response.statuscode.toEqual(201).** * Crie um **beforeAll** dê um **await app.ready()**, e no **afterAll** dê um **await app.close()**; * Crie o teste da autenticação. Crie um usuário primeiro. Depois tente fazer o login. Espere que o **status** seja 200. Espere que o **response.body** seja igual a um **token: expect.any(String);** * Crie o teste de **profile**. **Should be able to get user profile.** Repita a criação do usuário e a autenticação, esta ultima coloque em uma constante **authResponse** e desestruture **token** de **authResponse.body**. Faça um **profileResponse,** que do request vai para a rota **get ‘/me’** e depois um **set(‘Authorization’, `Bearer ${token}`.send().** Espero que do **profileResponse** o status sem 200, e espere que o **profileResponse.body**.**user.toEqual(expect.objectContains({email: };** |

|  |
| --- |
| **AULA 42** |
| * Na pasta **controllers** crie a pasta **user** e lance dentro todos os arquivos referentes aos usuários. Mova o arquivo **routes.ts** para dentro da mesma pasta; * Crie uma pasta em **controllers** chamada **gyms**, e crie um novo arquivo de rotas. Dentro da função **gymRoutes** faça um **app.addHook(‘onRequest’, verifyJWT);** * No arquivo **app.ts** faça a alteração dos nomes dos **plugins** para **usersRoutes** e **gymsRoutes;** * Crie o controller **create.ts**. Altere o **bodySchema** para criar uma **gym**, na key **latitude** e **longitude** dê tbm um método **refine** que recebe um **value** e diga que esse **value** sempre precisa ser maior que 90 e 180, respectivamente. Faça a alteração necessária no caso de uso colocando o **makeCreateGymUseCase**. Retire do **try/catch** |

|  |
| --- |
| **AULA 43** |
| * Crie os arquivos **search.ts** e **nearby.ts** na pasta **gyms.** Copie o arquivo do **create**; * No search faça o **searchGymQuerySchemas,** no atributo **page** use um **coerce** e o **min(1)** e **default(1)**. Extraia **page** e **query** do **parse.** Crie a constante **searchGymsUseCase** que vem do make(). Dê um **execute** em uma constante **{gyms}** e retorne status 200 com **{gyms}**; * No arquivo **nearby** copie a tipagem de latitude e longitude do **create**, parseie os valores e extraia-os. E em uma constante **{gyms}** use o **execute.** Retorne um status 200 com **{gyms};** * Crie as rotas, **/gyms/search** e **/gyms/nearby** em formato de **get** e uma **post** **/gyms/create** |

|  |
| --- |
| **AULA 44** |
| * Crie a pasta **checkins** do controllers**.** Crie o arquivo **routes.ts**, e os outros arquivos, **create.ts**, **history.ts**, **validate.ts** e **metrics.ts**; * Na rota de criação, faça o **schema** do **checkIn**. Na rota o **gymId** vem nos parâmetro, então crie outro **schema** para o **gymId**.Extraia os parâmetros dos parses. Faça a constante que recebe o **usecase** e dê um **execute** em todos os parâmetros, e ainda é preciso **userId**: **request.user.sub**. Retorne um 201; * Para o **history.** Faça um **querySchema**, pegue a page através do **parse**, aloque o **make()** e o **execute** você devolve em um objeto **checkIns**, dê um retorno 200; * Para o **metrics**. Chame o caso de uso e dê um **execute** que devolve **checkinscount** e o **reply 200 tbm devolve o checkInsCount;** * Para o **validate** pegue apenas o **checkInId** nos parâmetros, e aplique no **execute**, retorne um **204**; * As rotas serão uma **post** **/gyms/:gymId/check-ins**, para **create**, uma **patch /check-ins/:checkInId/validate** para o **validate. /check-ins/history** e **check-ins/metrics** que serão **gets;** |

|  |
| --- |
| **AULA 45** |
| * Dentro da pasta **utils** crie a pasta **tests** e faça uma função que cria e faz login em um usuário, o nome é **createAndAuthenticateUser**; * Crie o teste **should be able to create a gym.**Crie o usuário e faça o login, com a função criada. Crie uma nova academia. Espere que o status code seja 201; * Crie outro arquivo para o testar o **search**, crie um usuário/login e crie duas academias. Espere que o status code seja 200, e que o **response.body.gyms** tenha apenas 1 item. E que seja igual a um array contendo um objeto que contenha o **title: JavaScrip;** |

|  |
| --- |
| **AULA 46** |
| * Dentro do arquivo **app.ts**, dentro do **app.register** adicione uma segunda chave ao **parâmetro**, chamada **sign** que recebe um objeto **expiresIn: ‘10m’**; * Entre no arquivo **authenticate.ts** da pasta **user** e abaixo da constante do **token** crie a constante **refreshToken**, idêntico ao primeiro, mas com o objeto **sign,** que expira em **‘7d’;** * Instale o **npm i @fastify/cookie;** * No arquivo **app.ts** cadastre o **app.register(fastifyCookie)**; * De volta ao **authenticate.ts** no **return** dê um **setCookie(‘refreshToken’, refreshToken,{path: ‘/’, secure: true, sameSite: true, httpOnly: true};** * De volta ao **app.ts** no **register** do **JWT** adicione outra chave chamada **cookie** que recebe um objeto: **cookieName: ‘refreshToken’, signed: false;** |

|  |
| --- |
| **AULA 47** |
| * Crie um **controller** chamado **refresh.ts**. Copie o código da rota **authenticate**  para facilitar. Faça a troca de nomes. No arquivo **routes** crie uma nova rota **patch ‘/token/refresh’.** Dentro da função **refresh** chame o **await request.jwtVerify({onlyCookie: true}.** Gere um token para o usuário; * Crie um teste para **refresh**. Copie o de **authenticate.** Crie um usuário, faça o login dele**.** Pege os **cookies**  em **response.get(‘Set-Cookie’).** Faça uma chamada para a rota enviando os **cookies**. Espere que o retorno do status seja 200, espere que no body venha um **{token: expect.any(String)}**, espere que o **response.get(‘Set-Cookie’)** seja igual a um **expect.stringContaining(‘refreshToken=’);** |

|  |
| --- |
| **AULA 48** |
| * Crie na tabela **User** uma linha **role**. Antes dos **models**, crie um **enum Role {ADMIN MEMBER}** e membro como **default**. Rode a migration; * No arquivo **authenticate**, na constante **token** e na **refreshToken** crie um primeiro parâmetro que é um objeto **{role: user.role};** * No arquivo **fastify-jwt.ts** adicione o chave **role: ‘ADMIN’|’MEMBER’** dentro de **user**; * Agora adicione a role no arquivo **refresh.ts,** desestruturando ela de **request.user**; * Crie um novo **middleware** chamado **verify-user-role.ts**. Faça uma função que retorna **role** desestruturada e diz se essa **role** é igual ou não a do parâmetro da função, caso seja diferente retorne status 401 e a mensagem **Unauthorized.** A função não precisa ser assíncrona mas o primeiro **return** precisa; * Adicione este **middleware** também na rota de validação; |

|  |
| --- |
| **AULA X** |
|  |