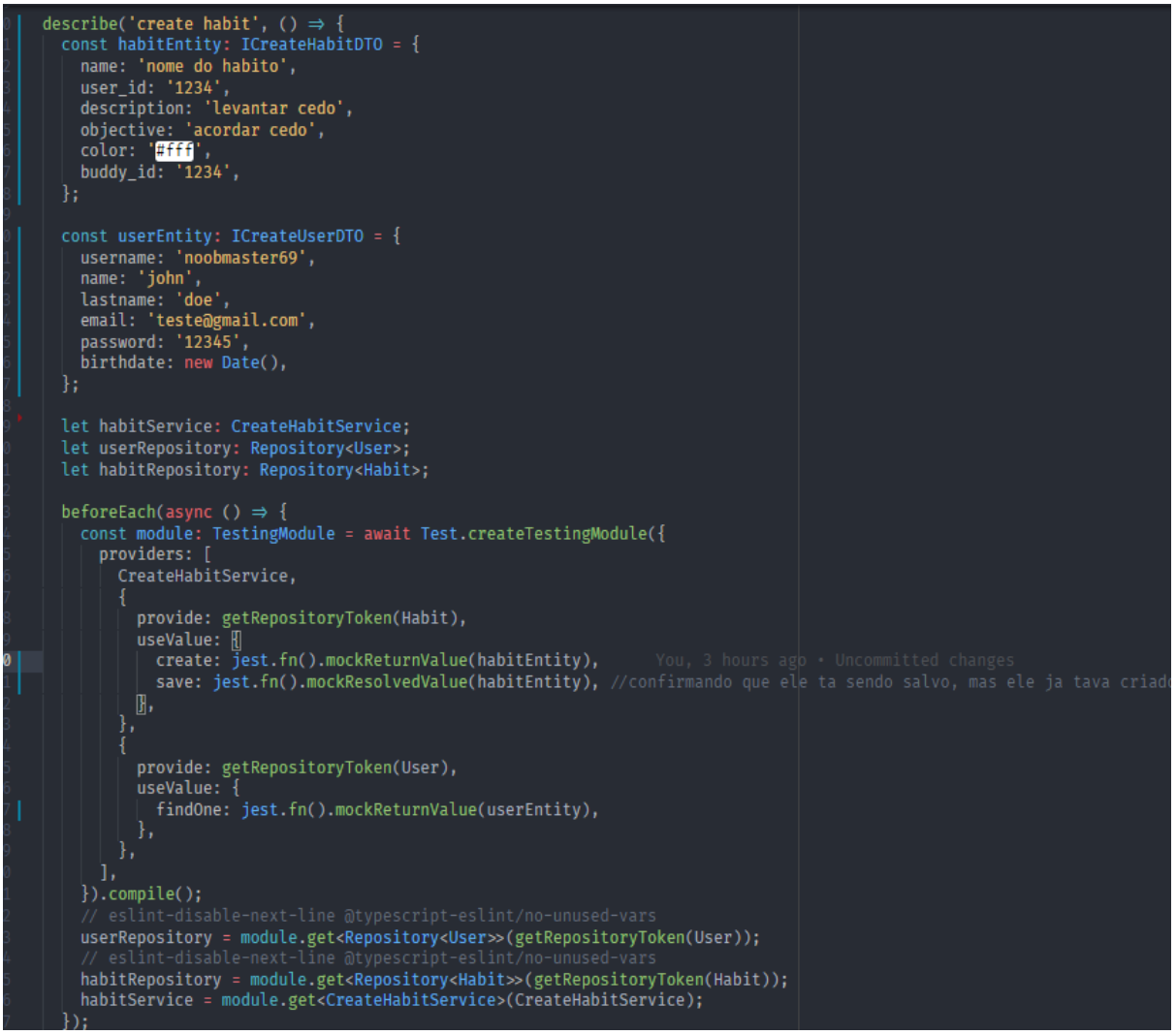
Relatório de testes unitários - Maiself

Este documento visa explicar como ocorreram os testes dentro da solução web denominada Mailsef dentro dos dois módulos existentes até então, que são os de hábitos e o de usuário. O tipo de teste que foi implementado na nossa aplicação foi o tipo de teste unitário da estrutura Jest, que vem por padrão integrado na criação de qualquer projeto Nest.Js. O foco dos arquivos de testes em todos os módulos criados foi os Services, que abstraem as regras de negócio para o Controller apenas chamar o arquivo e seu método respectivo, assim respeitando o princípio da responsabilidade única do desenvolvimento SOLID.

Também utilizamos a estratégia de mockRepository, que é é basicamente mockar (deixar estático) dados de entrada para que possíveis cenários sejam testados de acordo com o service, por que não é 100% de certeza nem nós desenvolvedores podemos garantir que tudo sempre irá funcionar como o banco de dados, o repositório do TypeOrm, então mockamos entradas para possíveis entradas de cenários e testados de acordo com a entrada.

Uma funcionalidade bastante interessante do Jest é com o comando de **test:cov,** com isso podemos ver o nível de cobertura de testes, ou seja, a medida que formos testando ele nos dá o indicativo em percentual da cobertura de testes para possíveis cenários na nossa aplicação. A primeira apresentação será referente aos testes unitários ligados ao módulo de hábitos.

Como os testes foram construídos:



```
describe('create habit', () => {
  const habitEntity: ICreateHabitDTO = {
    name: 'nome do habito',
    user_id: '1234',
    description: 'levantar cedo',
    objective: 'acordar cedo',
    color: '#fff',
    buddy_id: '1234',
  };

  const userEntity: ICreateUserDTO = {
    username: 'noobmaster69',
    name: 'john',
    lastname: 'doe',
    email: 'teste@gmail.com',
    password: '12345',
    birthdate: new Date(),
  };

  let habitService: CreateHabitService;
  let userRepository: Repository<User>;
  let habitRepository: Repository<Habit>;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        CreateHabitService,
        {
          provide: getRepositoryToken(Habit),
          useValue: {
            create: jest.fn().mockReturnValue(habitEntity),      You, 3 hours ago • Uncommitted changes
            save: jest.fn().mockResolvedValue(habitEntity), //confirmando que ele ta sendo salvo, mas ele ja tava criad
          },
        },
        {
          provide: getRepositoryToken(User),
          useValue: {
            findOne: jest.fn().mockReturnValue(userEntity),
          },
        },
      ],
    }).compile();
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    userRepository = module.get<Repository<User>>(getRepositoryToken(User));
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    habitRepository = module.get<Repository<Habit>>(getRepositoryToken(Habit));
    habitService = module.get<CreateHabitService>(CreateHabitService);
  });
```

Descrição da imagem: Teste unitário de criação de um hábito - img 1

Nesse caso está sendo demonstrado o teste de criação de um hábito.Tudo começa com o mock das entidades que iremos utilizar no teste do service, que no caso é o hábito e o usuário que possuem seus próprios DTOs. Após isso começa o escopo de um novo teste em describe, que é onde iremos definir variáveis para o service e para os repositórios utilizados no service implementado de fato, e com os seus respectivos arquivos, por exemplo o habitService é do tipo CreateHabitService.

Então, criamos o novo módulo de teste passando o service que irá ser testado no local de providers (próprio do framework), caso fosse testado um **controller** teríamos que passar o controller na propriedade controller e caso o service dependa de algum serviço externo como envio de e-mail ou autenticação JWT será preciso importar os módulos na propriedade **imports** dentro da criação do módulo de teste. Como dito anteriormente, estamos passando um service de criação para testar mas suas dependências não estão bem resolvidas como acesso a métodos de busca nos repositórios de usuário ou hábito, então o TypeOrm nos dá uma estratégia que simula todos os métodos disponíveis no TypeOrm, em getRepositoryToken passando a entidade que você deseja que tenha acesso a esses métodos. Como estamos utilizando a estratégia de mock, além de mockarmos o

tipo de dado de entrada também iremos mockar quais funções serão de uso no arquivo de teste, que são definidos no useValue.

Descrição da Imagem: Mock do service utilizado e das funções utilizadas no teste

Para ficar mais claro, a imagem acima está dizendo que usaremos o valor da função create, que definimos ela como um tipo de função do Jest que tem como retorno o objeto em formato de hábito que foi criado anteriormente, e isso acontece com todos os métodos mas cada um com suas respectivas responsibilidades. Por fim é feita a instanciação do(s) service(s) ou de repositório(s) que você deseja utilizar no teste.

```
it('should be defined', async () => {
  expect(habitService).toBeDefined();
});

it('should be able to create a new habit', async () => {
  const data: ICreateHabitDTO = {
    name: 'nome do habito',
    user_id: '1234',
    description: 'levantar cedo',
    objective: 'acordar cedo',
    color: '#fff',
    buddy_id: '1234',
  };

  const result = await habitService.execute(data);

  expect(result).toEqual(habitEntity);
  expect(userRepository.findOne).toBeCalledTimes(2);
  expect(habitRepository.create).toBeCalledTimes(1);
  expect(habitRepository.save).toBeCalledTimes(1);
});

it('should not be able a create a habit with user_id nullable', async () => {
  jest.spyOn(userRepository, 'findOne').mockRejectedValueOnce(new Error()); //

  const data: ICreateHabitDTO = {
    name: 'nome do habito',
    user_id: '1234',
    description: 'levantar cedo',
    objective: 'acordar cedo',
    color: '#fff',
    buddy_id: '1234',
  };

  expect(habitService.execute(data)).rejects.toThrowError();
  expect(userRepository.findOne).toBeCalledTimes(1);
  expect(habitRepository.create).toBeCalledTimes(0);
  expect(habitRepository.save).toBeCalledTimes(0);
});
);
```

Descrição da imagem: Implementação do teste unitário no service de criação hábito

O primeiro caso de teste que temos que serve para todo teste que é criado é que o arquivo em questão deve estar definido, o segundo caso de teste que temos estamos criando um hábito e passado ele como parâmetro do execute do service, que é o método responsável pela criação do hábito, então logo após isso fazemos algumas validações para que o caso de teste seja válido, por exemplo esperamos que o resultado da execução seja igual ao objeto hábito criado antes de tudo, esperamos que o método findOne seja chamado duas vezes, o create e o save ambos chamados uma vez, assim o cenário de criação de um hábito estará válido.

Agora vamos passar para um cenário de teste onde acontece um erro. O método jest.spyOn diz basicamente que no repositório do usuário, vamos redefinir uma vez o método **findOne** para que ele retorne um erro. Então ao executar o service esperamos que

o teste rejeite a execução e lance um erro, com isso também há as validações de quantas vezes os métodos são chamados para que esse cenário seja válido, que no caso é o findOne no usuário uma vez (a execução não passará daqui) e os outros métodos não executam nenhuma vez.

Explicando os demais testes unitários:

ViewHabitService: O service tem como intuito verificar, retornar um hábito dado um determinado id.

```
describe('view habits service', () => {
  const habitentity = {
    name: 'nome do hábito',
    description: 'hábito de levantar cedo',
    objective: 'acordar cedo',
    color: '#fff',
    buddy_id: '12345',
    frequency: [],
  };

  let viewHabitservice: ViewHabitService;      You, 2 weeks ago • test: tests habit
  let habitRepository: Repository<Habit>;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        ViewHabitService,
        {
          provide: getRepositoryToken(Habit),
          useValue: {
            findOne: jest.fn().mockReturnValue(habitentity),
          },
        },
      ],
    }).compile();

    habitRepository = module.get<Repository<Habit>>(getRepositoryToken(Habit));
    viewHabitservice = module.get<ViewHabitService>(ViewHabitService);
  });

  it('should be able to view habits', async () => {
    const result = await viewHabitservice.execute('12345', '1');
    expect(result).toEqual(habitentity);
    expect(habitRepository.findOne).toBeCalledTimes(2);
  });

  it('should not be able to view with id habit nullable', async () => {
    jest.spyOn(habitRepository, 'findOne').mockRejectedValueOnce(new Error());

    expect(viewHabitservice.execute('12345', '1')).rejects.toThrowError();
    expect(habitRepository.findOne).toBeCalledTimes(1);
  });
});
```

Descrição da imagem: Teste unitário para service de visualizar hábito

Os processo de mockar um service, uma entidade ou um repositório são iguais a todos os services que serão mostrados nesse documento, então como dito anteriormente estamos criando um módulo de teste passando o service que desejamos testar e definindo suas dependências com os métodos que são utilizados na classe de implementação, que no caso é apenas o findOne.

Para o primeiro caso de teste passamos dois id's no formato string como parâmetro do execute e esperamos que ele nos retorne um objeto hábito no formato que foi definido anteriormente (antes da criação do teste, describe). Após igualar os objetos e passar em uma validação esperamos que o findOne seja chamado duas vezes. Para o caso de teste que um id não é passado, iremos mockar novamente o erro, mas dessa vez vamos utilizar o repositório do hábito para dizer que uma vez o findOne quando chamado irá retornar um erro, assim quando os id's forem passados para a execução, o teste irá rejeitar e irá lançar um erro esperando que o findOne seja chamado uma vez.

ListHabitService: Service para listar todos os hábitos em forma de array, caso o usuário possua, dado um id de usuário que seja válido;

```
describe('list habits', () => {
  const habitEntityList = [
    {
      name: 'nome do habito',
      user_id: '1234',
      description: 'levantar cedo',
      objective: 'acordar cedo',
      color: '#fff',
      buddy_id: '1234',
    },
    {
      name: 'nome do habito',
      user_id: '1234',
      description: 'levantar cedo',      You, 2 hours ago • Uncommitted changes
      objective: 'acordar cedo',
      color: '#fff',
      buddy_id: '1234',
    },
  ];

  let listHabitService: ListHabitsService;
  let habitRepository: Repository<Habit>;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        ListHabitsService,
        {
          provide: getRepositoryToken(Habit),
          useValue: {
            find: jest.fn().mockReturnValue(habitEntityList),
          },
        },
        {
          provide: getRepositoryToken(User),
          useValue: {
            find: jest.fn(),
          },
        },
      ],
    }).compile();

    habitRepository = module.get<Repository<Habit>>(getRepositoryToken(Habit));
    listHabitService = module.get<ListHabitsService>(ListHabitsService);
  });

  it('should be defined service', async () => {
    expect(listHabitService).toBeDefined();
  });

  it('should be able to list habits', async () => {
    const result = await listHabitService.execute('1');
    expect(result.length).toEqual(habitEntityList.length);
    expect(habitRepository.find).toBeCalledTimes(1);
  });
});
```

Descrição da imagem: Teste unitário para visualizar todos os hábitos do usuário

Seguindo os mesmos passos e explicações dos testes anteriores, temos que primeiro na criação do módulo de teste dizer ao arquivo de teste o que vai ser testado que no caso é um service que é o de listar os hábitos do usuário, após isso resolvemos as dependências do service com o typeorm e dizemos que métodos ele usa. No caso de teste temos que ao passar um id, esperamos que o resultado da query para o id, retorne todos os hábitos em forma de lista e que ele seja compatível com a lista de hábitos que foi definida acima do teste, e claro, temos a validação de quantas vezes um método do repositório tem que ser chamado para aquele service.

DeleteHabitService: Service responsável por deletar um hábito dado que seja passado um id de usuário e o id do hábito a ser deletado

```
describe('delete habit', () => {
  const habityEntity: ICreateHabitDTO = {
    name: 'nome do habito',      You, seconds ago • Uncommitted changes
    user_id: '1234',
    description: 'levantar cedo',
    objective: 'acordar cedo',
    color: '#fff',
    buddy_id: '1234',
  };

  const habitSuccessuflyDelete = {
    raw: 0,
    affected: 1,
  };

  let habitRepository: Repository<Habit>;
  let deleteHabitService: DeleteHabitService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        DeleteHabitService,
        {
          provide: getRepositoryToken(Habit),
          useValue: {
            findOne: jest.fn().mockReturnValue(habityEntity),
            delete: jest.fn().mockReturnValue(habitSuccessuflyDelete),
          },
        },
      ],
    }).compile();

    habitRepository = module.get<Repository<Habit>>(getRepositoryToken(Habit));
    deleteHabitService = module.get<DeleteHabitService>(DeleteHabitService);
  });
```

Descrição da imagem: Mockando valores para a criação do módulo de teste

Neste caso de deletar um hábito, temos um objeto que guarda o valor de alteração de uma coluna que por padrão começa com raw:0 e affected: 1, que significa sucesso para remoção do hábito. Então na definição de dependências temos que ele vai usar o findOne para retornar um valor igual ao objeto criado mais acima e que o delete retorne um objeto que seja igual ao de remoção do hábito.

```
  it('should be able service defined', () => {
    expect(deleteHabitService).toBeDefined();
  });

  it('should be able a delete a service', async () => {
    const result = await deleteHabitService.execute('1', '1');

    expect(result).toMatchObject({ affected: 1, raw: 0 });
    expect(habitRepository.findOne).toBeCalledTimes(2);
  });

  it('should not be able to delete a habit with nullable id', async () => {
    jest.spyOn(habitRepository, 'findOne').mockRejectedValue(new Error());

    expect(deleteHabitService.execute('1', '')).rejects.toThrowError();
    expect(habitRepository.findOne).toBeCalledTimes(1);
    expect(habitRepository.delete).toBeCalledTimes(0);
  });

  it('should not be able to delete with raw null', async () => {
    habitSuccessuflyDelete.raw = 1;
    habitSuccessuflyDelete.affected = 0;

    jest
      .spyOn(habitRepository, 'delete')
      .mockResolvedValueOnce(habitSuccessuflyDelete);
    const result = await deleteHabitService.execute('1', '1');

    expect(result).toMatchObject({ affected: 0, raw: 1 });
    expect(habitRepository.findOne).toHaveBeenCalledTimes(2);
    expect(habitRepository.delete).toHaveBeenCalledTimes(1);

    habitSuccessuflyDelete.raw = 0;
    habitSuccessuflyDelete.affected = 1;
  });
});
```

Descrição da imagem: Testes implementados para o service de remover hábito

Neste caso temos mais fluxos que o service pode seguir e que abordamos para aumentar o índice de cobertura. O primeiro caso é o caso feliz onde é passado o id de um usuário e de um hábito que sejam válidos, então esperamos que o resultado da execução do service seja um objeto igual ao objeto de remover hábito com sucesso que é raw:0 affected:1. O segundo caso de teste é quando mockamos o erro do service passando um id nulo, então no findOne do repositório de hábito ele está marcado para retornar um erro, então quando chamamos o execute do service passando valores definimos que o teste rejeite a execução lançando um erro com as chamadas de método de findOne uma vez e para delete nenhuma vez. O outro caso é quando o objeto de sucesso para remover um hábito seja o oposto, que no caso é a falha ocasionando em raw:1 e affected:0.

UpdateHabitService: Este service tem como responsabilidade atualizar dados de qualquer hábito válido de acordo com a necessidade do usuário, que ele vai informar quais dados serão alterados.

```
describe('update habit', () => {
  const habitEntity: IUpdateHabitDTO = {
    name: 'levantar cedo',
    description: 'levantar cedo',
    objective: 'acordar cedo',
    color: '#fff',
    buddy_id: '1234',
  };

  const userEntity: ICreateUserDTO = {
    username: 'noobmaster69',
    name: 'john',
    lastname: 'doe',
    email: 'teste@gmail.com',
    password: '12345',
    birthdate: new Date(),
  };

  let updateHabitService: UpdateHabitService;
  let habitRepository: Repository<Habit>;
  let usersRepository: Repository<User>;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        UpdateHabitService,
        {
          provide: getRepositoryToken(Habit),
          useValue: {
            findOne: jest.fn().mockReturnValue(habitEntity),
            merge: jest.fn(),
            save: jest.fn().mockResolvedValue(habitEntity),
          },
        },
        {
          provide: getRepositoryToken(User),
          useValue: {
            findOne: jest.fn().mockReturnValue(userEntity),
          },
        },
      ],
    }).compile();

    usersRepository = module.get<Repository<User>>(getRepositoryToken(User));
    habitRepository = module.get<Repository<Habit>>(getRepositoryToken(Habit));
    updateHabitService = module.get<UpdateHabitService>(UpdateHabitService);
  });
```

Descrição da imagem: Mock do service a ser testado junto com suas dependências

Como qualquer outro service o que temos aqui é o mock e definição de qual service deve ser testado junto com sua definição de métodos e entidades a serem utilizadas.

```
  it('should be defined service', async () => {
    expect(updateHabitService).toBeDefined();
  });

  it('should be able to update a habit valid', async () => {
    const data: IUpdateHabitDTO = {
      name: 'nome do habito',      You, 2 weeks ago • test: tests habit module
      description: 'levantar cedo',
      objective: 'acordar cedo',
      color: '#fff',
      buddy_id: '1235',
    };

    const result = await updateHabitService.execute('1', '1235', data);
    expect(result).toEqual(habitEntity);
    expect(usersRepository.findOne).toBeCalledTimes(1);
    expect(habitRepository.findOne).toBeCalledTimes(2);
    expect(habitRepository.merge).toBeCalledTimes(1);
    expect(habitRepository.save).toBeCalledTimes(1);
  });

  it('should not be able do update a habit with no user id', async () => {
    jest.spyOn(habitRepository, 'findOne').mockRejectedValueOnce(new Error());

    const data: IUpdateHabitDTO = {
      name: 'nome do habito',
      description: 'levantar cedo',
      objective: 'acordar cedo',
      color: '#fff',
      buddy_id: '1235',
    };

    expect(
      updateHabitService.execute('1', '12345', data),
    ).rejects.toThrowError();
    expect(habitRepository.findOne).toBeCalledTimes(1);
    expect(habitRepository.merge).toBeCalledTimes(0);
    expect(habitRepository.save).toBeCalledTimes(0);
  });
});
```

Descrição da imagem: Implementação do teste ao service usado para teste

O primeiro caso de teste (após o teste definição do service que é padrão para todos os services) é o caso feliz onde o usuário consegue atualizar o seu hábito, passando o id do usuário, do hábito e novos dados para que esse hábito seja atualizado. Esperamos que o que foi atualizado seja compatível com o objeto criado acima e também que as chamadas de método sejam iguais a definidas no teste. O segundo caso de teste é o caso de erro quando o usuário não consegue atualizar o hábito, a razão é pelo id inválido. Mockamos o erro para que o findOne de repositório de hábito retorne um erro, então quando passamos os parâmetros para a execução do service esperamos que o teste lance rejeite a execução com um findOne feito uma vez e os demais métodos chamados nenhuma vez.

Testes unitários módulo de Usuário:

Para os testes desse módulo foi necessário definir mais dependências do que no módulo de hábitos por que aqui em determinados services, eles precisam de serviços externos a nossa aplicação que são por exemplo o serviço do SendGrid para envio de e-mails ou o JWT para autenticação do usuário. Assim como anteriormente explicaremos a funcionalidade do teste e por que ele existe e quais casos de teste/fluxo ele cobre.

CreateUserService: Este service é responsável pela verificação de dados passados por parâmetro e criar um novo usuário caso os dados sejam válidos, e retornar este usuário criado;

```ts
describe('Create User', () => {
  const userCreatedEntityList: Array<User> = [
    {
      name: 'namefield',
      username: 'usernamefield',
      lastname: 'lastnamefield',
      email: 'emailfield@gmail.com',
      password: 'qwe123',
      birthdate: new Date(),
      id: 'idfield',
      bodies: [],
      avatar: 'avatarfield',
      created_at: new Date(),
      updated_at: new Date(),
    },
    {
      name: 'namefield2',
      username: 'usernamefield2',
      lastname: 'lastnamefield2',
      email: 'emailfield2@gmail.com',
      password: 'qwe1232',
      birthdate: new Date(),
      id: 'idfield2',
      bodies: [],      Eric Rodrigues, 5 days ago • refactor: remove the depender
      avatar: 'avatarfield2',
      created_at: new Date(),
      updated_at: new Date(),
    },
  ];

  const emailCreateUserSend = {
    to: userCreatedEntityList[0].email,
    from: 'no-reply@maiself.com.br',
    subject: 'Welcome to Maiself',
    templateId: 'd-edce0598398f458692d26ae47ae5dbda',
    dynamicTemplateData: {
      first_name: userCreatedEntityList[0].name,
    },
  };

  const hashProvider = () => {
    return {
      generateHash: jest
        .fn()
        .mockReturnValue(
          '8A085DFC3E5BEF71F611D372D8C0040E9A525F08B9B53DE9F0804946218E0FB8',
        ),
      compareHash: jest.fn(),
    };
  };

  let usersRepository: Repository<User>;
  let createUserService: CreateUserService;
```

Descrição da imagem: Mock dos dados para realizar testes no CreateUserService

Como dito anteriormente, alguns services do usuário irão depender de serviços externos e no caso o de criação de usuário é um deles. Primeiro de tudo estamos mockando dados como uma lista de usuários e seus dados necessários para criação, seguidamente estamos mockando um template de e-mail a ser enviado ao usuário recém cadastrado e registrado no banco de dados, após isso temos também o mock do hashProvider que consiste no encriptador da senha na nossa solução.

```
beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
    imports: [
      SendGridModule.forRoot({
        apiKey:
          'SG.J8xI-wBDSc2eP-m0Cal9Gw.wmeCNr-O95f6j-DdM8q2994dTkVBrslj2n1Gn6XS_A',
      }),
    ],
    providers: [
      CreateUserService,
      {
        provide: getRepositoryToken(User),
        useValue: {
          findOne: jest.fn(),
          create: jest.fn().mockReturnValue(userCreatedEntityList[0]),
          save: jest.fn().mockResolvedValue(userCreatedEntityList[0]),
        },
      },
      {
        provide: SendGridService,
        useValue: {
          send: jest.fn().mockResolvedValue(emailCreateUserSend),
        },
      },
      { provide: 'BCryptHashProvider', useFactory: hashProvider },
    ],
  }).compile();

  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  usersRepository = module.get<Repository<User>>(getRepositoryToken(User));
  createUserService = module.get<CreateUserService>(CreateUserService);
});

it('Should be able defined create user service', () => {
  expect(createUserService).toBeDefined();
});

it('Should be able create user', async () => {
  const data: ICreateUserDTO = {
    name: 'namefield',
    username: 'usernamefield',
    lastname: 'lastnamefield',
    email: 'emailfield@gmail.com',
    password: 'qwe123',
    birthdate: new Date(),
  };

  const result = await createUserService.execute(data);

  expect(result).toEqual(userCreatedEntityList[0]);
  expect(usersRepository.create).toHaveBeenCalledTimes(1);
  expect(usersRepository.save).toHaveBeenCalledTimes(1);
  expect(usersRepository.findOne).toHaveBeenCalledTimes(2);
});
```

Descrição da imagem: Criação módulo de testes e casos de teste

Na criação do módulo de teste temos que configurar o sendGrid para que o e-mail seja enviado mais tarde caso tudo ocorra bem, e na parte de providers temos a definição de

dependências que acontece em todos os services e já foi descrito e comentado anteriormente. No segundo caso de teste válido temos o caminho para o usuário ser cadastrado, então passamos os dados de um usuário e esperamos que ele seja igual ao usuário na posição 0 da lista de usuários criada anteriormente, após isso temos as verificações de quantas vezes os métodos dos repositórios são chamados. Em seguida serão mostrados os casos onde a criação não acontece:

```
  it('Should not be able create user what exists', async () => {
    jest.spyOn(usersRepository, 'findOne').mockRejectedValueOnce(new Error());

    const data: ICreateUserDTO = {
      name: 'namefield',
      username: 'usernamefield',
      lastname: 'lastnamefield',
      email: 'emailfield@gmail.com',
      password: 'qwe123',
      birthdate: new Date(),
    };

    expect(createUserService.execute(data)).rejects.toEqual(
      new HttpException(
        'Sorry, this operation could not be performed, please try again.',
        HttpStatus.BAD_REQUEST,
      ),
    );
    expect(usersRepository.findOne).toHaveBeenCalledTimes(1);      ericrodriguesfer,
    expect(usersRepository.create).toHaveBeenCalledTimes(0);
    expect(usersRepository.save).toHaveBeenCalledTimes(0);
  });

  it('Should be able create other user', async () => {
    jest
      .spyOn(usersRepository, 'create')
      .mockReturnValueOnce(userCreatedEntityList[1]);

    jest
      .spyOn(usersRepository, 'save')
      .mockResolvedValueOnce(userCreatedEntityList[1]);

    const data: ICreateUserDTO = {
      name: 'namefield2',
      username: 'usernamefield2',
      lastname: 'lastnamefield2',
      email: 'emailfield2@gmail.com',
      password: 'qwe1232',
      birthdate: new Date(),
    };

    const result = await createUserService.execute(data);

    expect(result).toEqual(userCreatedEntityList[1]);
    expect(usersRepository.create).toHaveBeenCalledTimes(1);
    expect(usersRepository.save).toHaveBeenCalledTimes(1);
    expect(usersRepository.findOne).toHaveBeenCalledTimes(2);
  });

  it('Should not be able create other user what exists', async () => {
    jest.spyOn(usersRepository, 'findOne').mockRejectedValueOnce(new Error());

    const data: ICreateUserDTO = {
      name: 'namefield2',
      username: 'usernamefield2',
      lastname: 'lastnamefield2',
      email: 'emailfield2@gmail.com',
      password: 'qwe1232',
      birthdate: new Date(),
    };

    expect(createUserService.execute(data)).rejects.toEqual(
      new HttpException(
        'Sorry, this operation could not be performed, please try again.',
        HttpStatus.BAD_REQUEST,
      ),
    );
    expect(usersRepository.findOne).toHaveBeenCalledTimes(1);
    expect(usersRepository.create).toHaveBeenCalledTimes(0);
    expect(usersRepository.save).toHaveBeenCalledTimes(0);
  });
});
```

Descrição da imagem: Casos de testes de sucesso e falha ao criar mais de um usuário

DeleteUserService: Service responsável pela remoção de um usuário do banco de dados, semelhante ao remover um hábito do banco, como o processo é semelhante em definição de dependências e nos casos, segue a imagem:

```
let usersRepository: Repository<User>;
let deleteUserService: DeleteUserService;

beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
    providers: [
      DeleteUserService,
      {
        provide: getRepositoryToken(User),
        useValue: {
          findOne: jest.fn().mockReturnValue(userCreatedEntity),
          delete: jest.fn().mockReturnValue(successfulDelete),
        },
      },
    ],
  }).compile();

  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  usersRepository = module.get<Repository<User>>(getRepositoryToken(User));
  deleteUserService = module.get<DeleteUserService>(DeleteUserService);
});

it('Should be able defined create delete user service', () => {
  expect(deleteUserService).toBeDefined();
});      ericrodriguesfer, 3 weeks ago • feat: adjust import unnecessary of DeleteResult f…

it('Should be able delete user', async () => {
  const result = await deleteUserService.execute('1');

  expect(result).toEqual(true);
  expect(usersRepository.findOne).toHaveBeenCalledTimes(1);
  expect(usersRepository.delete).toHaveBeenCalledTimes(1);
});

it('Should not be able delete user', async () => {
  successfulDelete.raw = 1;
  successfulDelete.affected = 0;

  jest
    .spyOn(usersRepository, 'delete')
    .mockResolvedValueOnce(successfulDelete);
  const result = await deleteUserService.execute('123');

  expect(result).toEqual(false);
  expect(usersRepository.findOne).toHaveBeenCalledTimes(1);
  expect(usersRepository.delete).toHaveBeenCalledTimes(1);

  successfulDelete.raw = 0;
  successfulDelete.affected = 1;
});

it('Should not be able delete user, because user not exists, action throws exception', async () => {
  jest.spyOn(usersRepository, 'findOne').mockRejectedValueOnce(new Error());

  expect(deleteUserService.execute('123')).rejects.toEqual(
    new HttpException(
      'Sorry, this operation could not be performed, please try again.',
      HttpStatus.BAD_REQUEST,
    ),
  );
  expect(usersRepository.findOne).toHaveBeenCalledTimes(1);
  expect(usersRepository.delete).toHaveBeenCalledTimes(0);
});
});
```

Descrição da imagem: Casos de testes de sucesso e falha ao remover usuário

UpdateUserService: Novamente, como o processo de criação do módulo de teste e definição de dependências é padrão e se assemelha com a atualização de um hábito (mas

muda os dados a serem atualizados), segue a imagem no teste de atualizar um usuário nos

casos de sucesso ou falha:

```
let usersRepository: Repository<User>;
let updateUserService: UpdateUserService;

beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
    providers: [
      UpdateUserService,
      {
        provide: getRepositoryToken(User),
        useValue: {
          findOne: jest.fn().mockReturnValue(userUpdate),
          merge: jest.fn().mockReturnValue(userUpdated),
          save: jest.fn().mockReturnValue(userUpdated),
        },
      },
      { provide: 'BCryptHashProvider', useValue: hashProvider },
    ],
  }).compile();

  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  usersRepository = module.get<Repository<User>>(getRepositoryToken(User));
  updateUserService = module.get<UpdateUserService>(UpdateUserService);
});

it('Should be able defined create update user service', () => {
  expect(updateUserService).toBeDefined();
});

it('Should be able update user', async () => {
  const data: IUpdateUserDTO | User = {
    name: 'namefield2',
    lastname: 'lastnamefield2',
    username: 'usernamefield',
    email: 'emailfield@gmail.com',
    password: 'passwordfield',
    birthdate: new Date(),
  };

  const result = await updateUserService.execute('1', data);
    ericrodriguesfer, 2 weeks ago • test: service update user tested with succ
  expect(result).toEqual(userUpdated);
  expect(usersRepository.findOne).toBeCalledTimes(1);
  expect(usersRepository.merge).toBeCalledTimes(1);
  expect(usersRepository.save).toBeCalledTimes(1);
});

it('Should not be able update user, because user not exists', async () => {
  jest.spyOn(usersRepository, 'findOne').mockRejectedValueOnce(new Error());

  const data: IUpdateUserDTO | User = {
    name: 'namefield2',
    lastname: 'lastnamefield2',
    username: 'usernamefield',
    email: 'emailfield2@gmail.com',
    password: 'passwordfield',
    birthdate: new Date(),
  };

  expect(updateUserService.execute('1', data)).rejects.toEqual(
    new HttpException(
      'Sorry, this operation could not be performed, please try again.',
      HttpStatus.BAD_REQUEST,
    ),
  );
  expect(usersRepository.findOne).toBeCalledTimes(1);
  expect(usersRepository.merge).toBeCalledTimes(0);
  expect(usersRepository.save).toBeCalledTimes(0);
});
});
```

Descrição da imagem: Casos de testes de sucesso e falha ao atualizar usuário

Testes unitários módulo de Amizade:

Módulo referente a opção que um usuário tem ao criar um hábito, onde o mesmo pode adicionar ao seu hábito que está sendo criado, um outro usuário da plataforma para que ela possa acompanhar esse dado hábito em questão, no qual esse segundo usuário foi atribuído como “amigo” neste dado hábito.

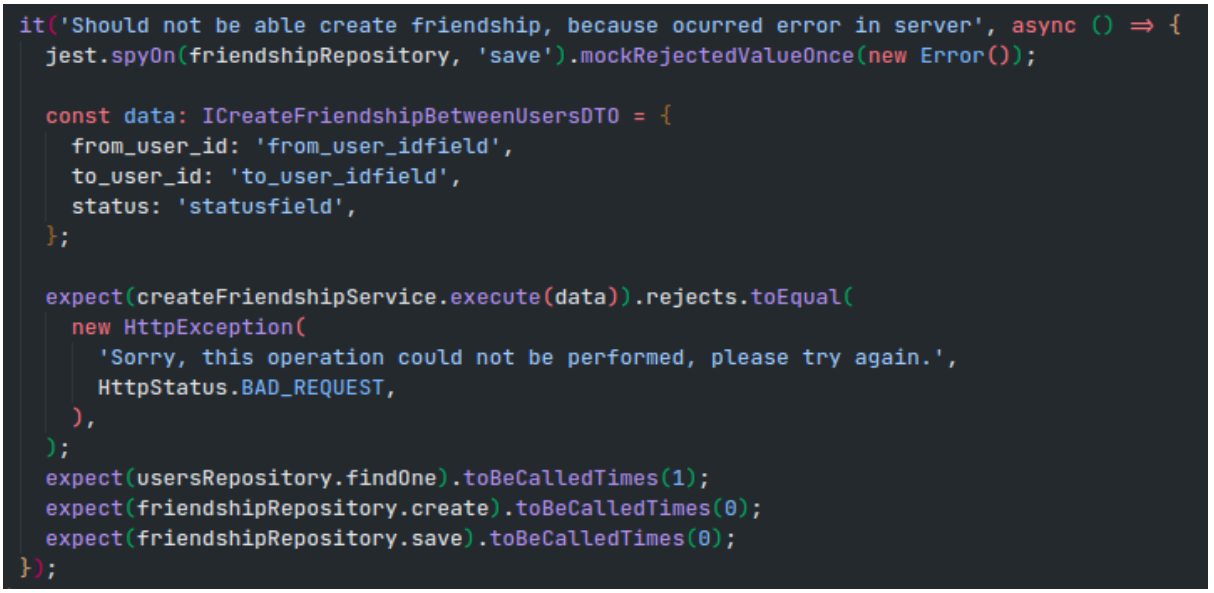
CreateFriendShipService: Implementação de casos de testes para a criação de uma amizade em um dado hábito, do usuário corrente logado, com um de sua preferência para ser seu acompanhante nesse hábito.

```ts
describe('Create Friendship', () => {
  const userCreated: User = {
    id: 'idfield',
    email: 'emailfield',
    password: 'passwordfield',
    name: 'namefield',
    lastname: 'lastnamefield',
    username: 'usernamefield',
    birthdate: new Date(),
    bodies: [],
    avatar: 'avatarfield',
    created_at: new Date(),
    updated_at: new Date(),
    fullName: 'fullNamefield',
  };

  const friendshipCreated: Friendship = {
    id: 'idfield',
    from_user_id: 'from_user_idfield',
    to_user_id: 'to_user_idfield',
    status: 'statusfield',
    created_at: new Date(),
    updated_at: new Date(),
    fromUser: userCreated,
    toUser: userCreated,
  };

  let usersRepository: Repository<User>;
  let friendshipRepository: Repository<Friendship>;
  let createFriendshipService: CreateFriendshipBetweenUsersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [      You, 43 minutes ago · test: concluded testing for
        CreateFriendshipBetweenUsersService,
        {
          provide: getRepositoryToken(User),
          useValue: {
            findOne: jest.fn().mockReturnValue(userCreated),
          },
        },
        {
          provide: getRepositoryToken(Friendship),
          useValue: {
            create: jest.fn().mockReturnValue(friendshipCreated),
            save: jest.fn().mockReturnValue(friendshipCreated),
          },
        },
      ],
    }).compile();
```

Descrição da imagem: Definições e mocks utilizados para os testes de criação de amizade

```
it('Should be able create friendship in habit', async () => {
  const data: ICreateFriendshipBetweenUsersDTO = {
    from_user_id: 'from_user_idfield',
    to_user_id: 'to_user_idfield',
    status: 'statusfield',
  };

  const result = await createFriendshipService.execute(data);

  expect(result).toEqual(friendshipCreated);
  expect(usersRepository.findOne).toBeCalledTimes(2);
  expect(friendshipRepository.create).toBeCalledTimes(1);
  expect(friendshipRepository.save).toBeCalledTimes(1);
});

it('Should not be able create friendship, because user friend not exists', async () => {
  jest.spyOn(usersRepository, 'findOne').mockRejectedValueOnce(new Error());

  const data: ICreateFriendshipBetweenUsersDTO = {
    from_user_id: 'from_user_idfield',
    to_user_id: 'to_user_idfield',
    status: 'statusfield',
  };

  expect(createFriendshipService.execute(data)).rejects.toEqual(
    new HttpException(
      'Sorry, this operation could not be performed, please try again.',
      HttpStatus.BAD_REQUEST,
    ),
  );
  expect(usersRepository.findOne).toBeCalledTimes(1);
  expect(friendshipRepository.create).toBeCalledTimes(0);
  expect(friendshipRepository.save).toBeCalledTimes(0);
});
```

Descrição da imagem: Casos de teste de sucesso e falha na criação de uma amizade via hábito



```
it('Should not be able create friendship, because ocurred error in server', async () => {
  jest.spyOn(friendshipRepository, 'save').mockRejectedValueOnce(new Error());

  const data: ICreateFriendshipBetweenUsersDTO = {
    from_user_id: 'from_user_idfield',
    to_user_id: 'to_user_idfield',
    status: 'statusfield',
  };

  expect(createFriendshipService.execute(data)).rejects.toEqual(
    new HttpException(
      'Sorry, this operation could not be performed, please try again.',
      HttpStatus.BAD_REQUEST,
    ),
  );
  expect(usersRepository.findOne).toBeCalledTimes(1);
  expect(friendshipRepository.create).toBeCalledTimes(0);
  expect(friendshipRepository.save).toBeCalledTimes(0);
});
```

Descrição da imagem: Caso de erro onde ocorre falha na criação de amizade via hábito, por erro ocorrido no servidor