### Você será capaz de:

- Entender e aplicar conceitos de testes de integração / contrato;
- Criar testes de integração para API's REST;
- Testar endpoint com `middleware` de autenticação JWT.

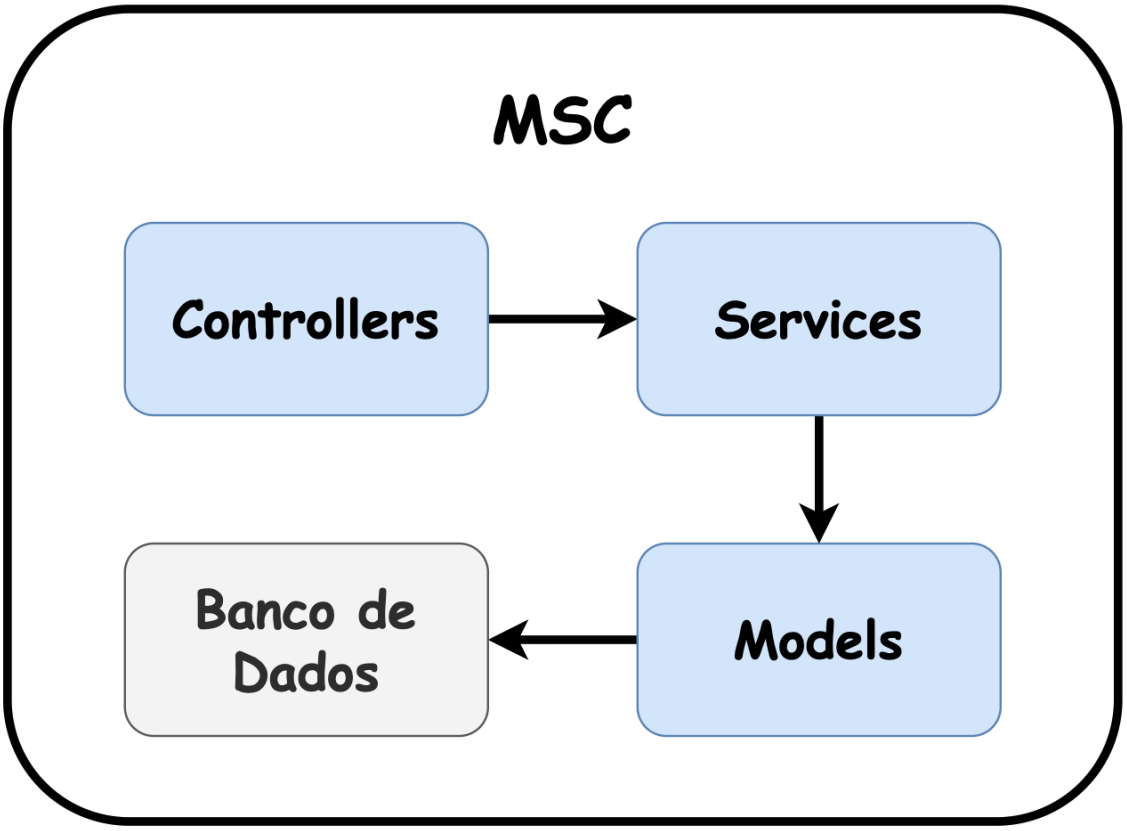
---

## Por que isso é importante?

Existem diversas maneiras diferentes de se testar uma aplicação com testes automatizados, cada uma com suas particularidades, prós e contras.
Ao longo da sua jornada como pessoa desenvolvedora, você irá lidar com esses diferentes tipos de teste, de acordo com a realidade do time e do projeto que você atuar.
Dessa forma, ter um leque amplo de conhecimentos sobre os tipos de testes mais comuns e como testar os principais cenários e padrões *(como API's REST com autenticação `JWT` , por exemplo)* será importante para o seu desenvolvimento e crescimento na área.

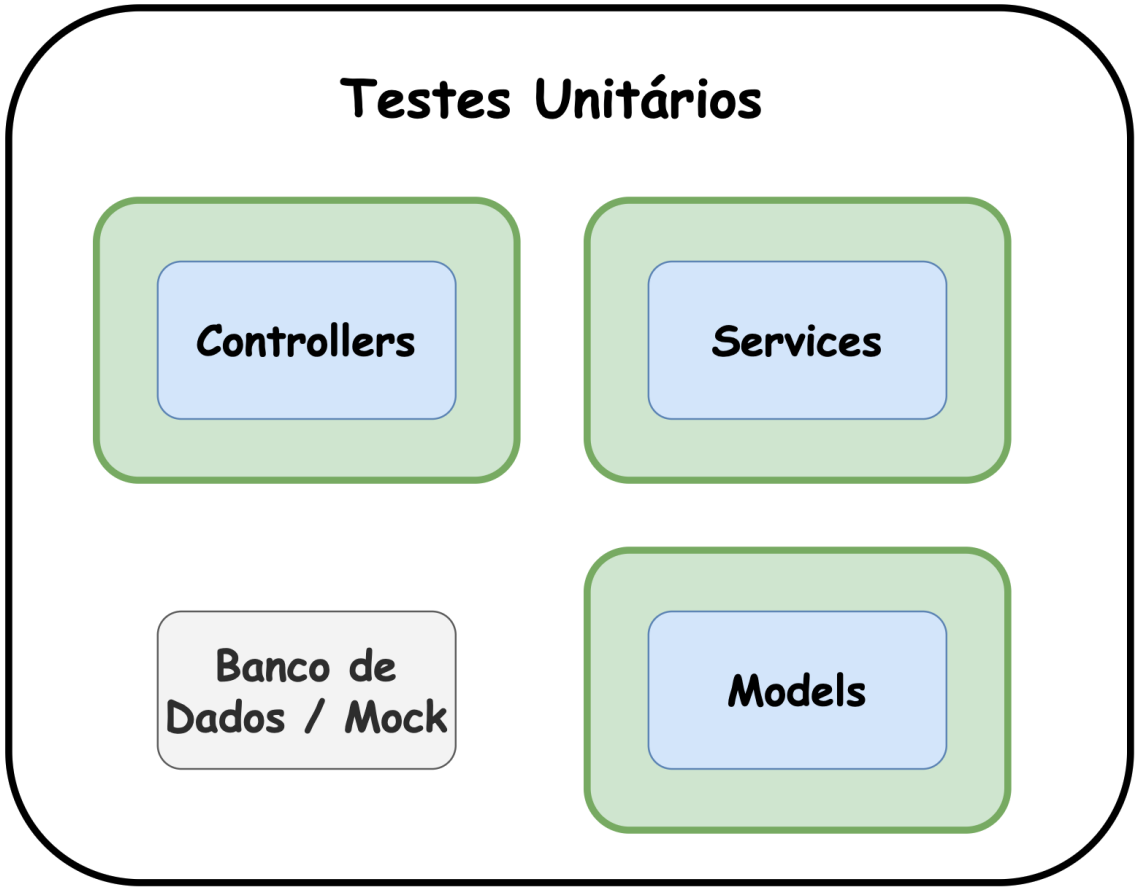
### Testes de Integração (Integration Tests)

Aprendemos anteriormente sobre testes unitários e como podemos testar cada unidade do nosso código de maneira individual. Por exemplo, ao utilizarmos o padrão MSC podemos definir cada camada como sendo uma unidade:




Arquitetura MSC

Dessa forma, podemos testar cada camada, ou seja, cada unidade de maneira separada, isolando uma parte da outra e escrevendo testes individuais para cada uma:




Testes Unitários

Seguindo nosso aprendizado sobre tipos de testes, aprenderemos como aplicar testes de integração .

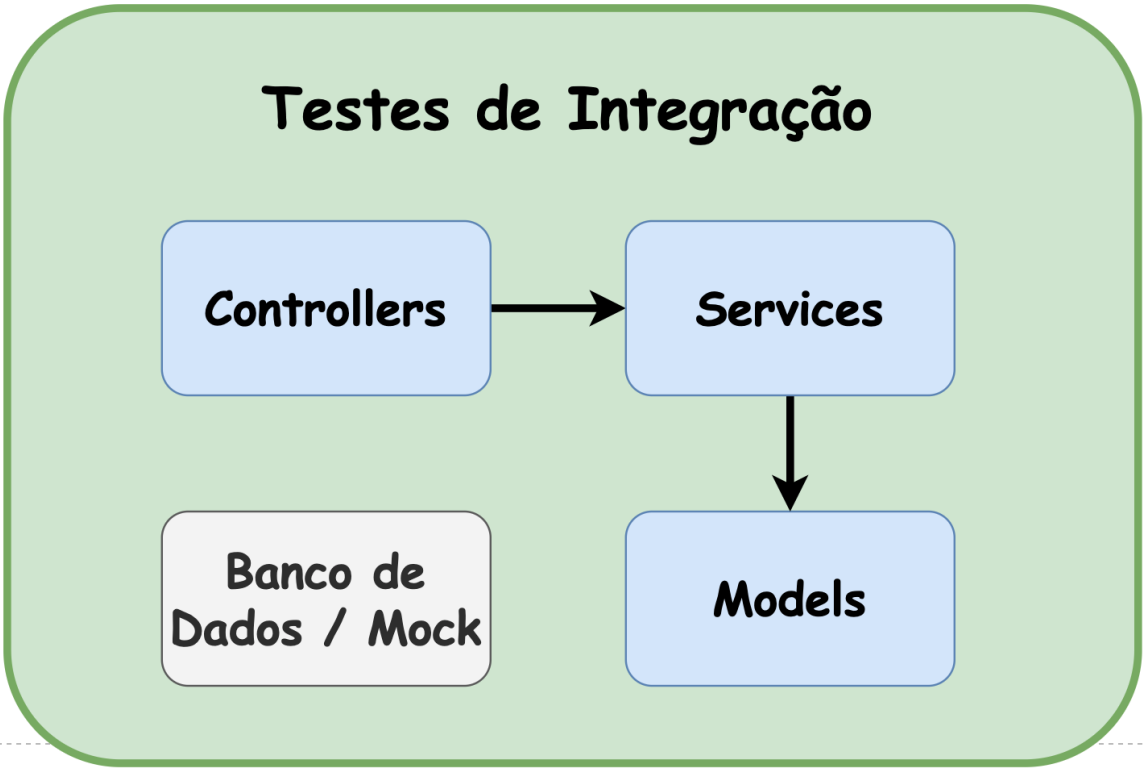
Os testes de integração, ou *integration tests* , servem para verificar se a comunicação entre os componentes de um sistema estão ocorrendo conforme o esperado.

Diferente dos testes unitários, onde isolamos cada unidade como fizemos com cada camada do MSC, nos testes de integração o comportamento deverá ser testado com a integração entre todas as unidades.

Ambos os tipos de testes são importantes, porém, cada um tem um objetivo diferente.


Testes unitários VS Testes de Integração

Da mesma forma como definir uma unidade é subjetivo, não existe um nível de granularidade específico de integração para ser testada, sendo possível adaptar esse conceito de acordo com os objetivos desejados. Nossa integração partirá do recebimento do objeto da requisição ( `request` ), seguindo com o controlador ( `controller` ), o serviço ( `service` ) e terminando no modelo ( `model` ). Iremos apenas isolar a comunicação do `model` com o Banco de dados para evitarmos operações de IO. Podemos representar isso da seguinte maneira:




Testes de Integração

## Contratos

Agora que já sabemos o nível de integração que iremos testar, ou seja, quais partes serão cobertas pelos nossos testes, precisamos saber o que exatamente queremos testar e de qual forma .
Iremos definir isso a seguir, mas antes, vamos refletir sobre um conceito importante.
Sempre quando consumimos ou fornecemos um serviço, como por exemplo uma API REST, precisamos ter os comportamentos pré definidos. Esses comportamentos são definidos de acordo com as regras de entrada e saída de dados da API .
Por exemplo, ao chamar um endpoint `GET /users/:userId` passamos um ID de usuário e esperamos receber os dados referentes aquele usuário com um código http `200 - OK` . Caso o usuário não seja encontrado, esperamos receber um status http `404 - Not Found` , por exemplo.
Perceba que temos diversos padrões definidos e comportamentos esperados. Dessa forma, podemos testar exatamente se esses comportamentos estão sendo cumpridos por nossas API's, retornando uma resposta compatível com o cenário.
Em testes, esse conceito é chamado de contratos . Por exemplo, ao se alugar um imóvel, é necessário assinar um contrato onde está definido tudo aquilo que foi combinado entre as partes. Nos testes de contratos de API, a ideia é bem semelhante, o contrato define aquilo que foi previamente acordado, ou seja, como a API deverá se comportar em um determinado cenário.
Para ficar ainda mais nítido, vamos utilizar novamente o endpoint `GET /users/:userId` . Podemos dizer que o contrato dele é, quando a pessoa usuária existe, retornar a seguinte resposta:

- Código HTTP: `200 - OK` ;
- Body:

Copiar

```
{
    "id": "123",
    "name": "jane",
    "fullName": "Jane Doe",
    "email": "janedoe@trybemail.com"
}
```

Esse conceito trabalha muito bem junto com os testes de integração, pois podemos testar se cada contrato está sendo cumprido após o processamento de todas as camadas.

## Escrevendo testes

### Baixando o repositório base

Chega de teoria e vamos ver como podemos fazer isso na prática. Vamos utilizar o mesmo projeto do conteúdo sobre `JWT` . Para isso, podemos baixar o repositório com o código base já com o JWT implementado, conforme fizemos na aula, neste link . O gabarito do dia está disponível neste link , na branch para `block-28-3`
*Nota 1: Caso já possua o projeto na sua máquina, você também pode criar uma nova branch (exemplo `tests` ) para aplicar o conteúdo de hoje*
*Nota 2: Valide a constante `MONGO_DB_URL` , no arquivo `./src/models/connection.js` (linha 3). Essa é a `URL` da sua instância local do MongoDB e ficará disponível assim que você executar o serviço do mongo (Relembre no Bloco 23 - Introdução ao MongoDB em "Mãos à obra, vamos executar!"). Localmente, a `URL` padrão é `mongodb://127.0.0.1:27017` .*

### Preparando o ambiente

1. Primeiro, modifique a estrutura do arquivo baixado, e coloque no seguinte formato;

Copiar

```
├── README.md
├── assets
├── src
│   ├── api
│   │   ├── routes.js
│   │   └── server.js
│   ├── controllers
│   │   ├── createUser.js
│   │   ├── login.js
│   │   └── posts.js
│   └── models
│       ├── connection.js
│       └── user.js
└── tests
├── package-lock.json
└── package.json
```

2. Em seguida, faça a instalação dos pacotes que já conhecemos anteriormente, para utilizarmos em ambiente de desenvolvimento, para realizarmos testes:

Copiar
```
npm i -D mocha chai sinon
```

Aqui, também é necessário a inicialização de um script de testes no `package.json` :

Copiar
```
...
"scripts": {
    ...
    "test": "mocha ./tests/**/*$NAME*.test.js --exit",
},
...
```

### Escrevendo um teste base

Agora vamos iniciar escrevendo testes para a rota de criação de usuários. Conforme definido, ao criar um usuário com sucesso o endpoint deverá responder:

- com o status http `201 - OK` ;
- com um objeto `JSON` , contendo a propriedade `message` com o valor `"Novo usuário criado com sucesso"` .

Essa definição é o contrato da nossa API. Podemos transformá-lo em teste convertendo-o para asserções/ afirmações, igual já fizemos anteriormente com o `mocha` e o `chai` :
tests/createUsers.test.js

Copiar
```
const chai = require('chai');

const { expect } = chai;

describe('POST /api/users', () => {
  describe('quando é criado com sucesso', () => {
    let response = {};

    it('retorna o código de status 201', () => {
      expect(response).to.have.status(201);
    });
```

```
    it('retorna um objeto', () => {
      expect(response.body).to.be.a('object');
    });

    it('o objeto possui a propriedade "message"', () => {
      expect(response.body).to.have.property('message');
    });

    it('a propriedade "message" possui o texto "Novo usuário criado
com sucesso"',
      () => {
        expect(response.body.message)
          .to.be.equal('Novo usuário criado com sucesso');
      });
  });
});
```

Agora que temos nosso contrato expresso em código precisamos validar se nossa aplicação está obedecendo aquilo que está definido nele.

### Testando nossa API

Iremos testar toda nossa API de maneira integrada, ou seja, queremos testar que dado um valor de entrada, o mesmo será processado pelas diversas partes da API, então, nos dar um retorno conforme estabelecido pelo nosso "contrato". Diferente de como fizemos antes testando cada unidade da API com os testes unitários por camada.
Para nos ajudar com esse desafio, utilizaremos o plugin Chai HTTP com esse plugin poderemos simular uma request a nossa API.
Primeiro precisamos instalar esse novo pacote, para isso, execute:

Copiar
```
npm install -D chai-http
```

E então no nosso teste iremos adicionar o seguinte trecho, adicionando o plugin a instância do chai:

Copiar
```
// const chai = require('chai');
const chaiHttp = require('chai-http');

chai.use(chaiHttp);

// const { expect } = chai
```

```
// ...
```

Adicionado o plugin ao chai, poderemos consumir nosso server em `express` através dele, sem que haja a necessidade de subirmos a api manualmente. Para isso basta importarmos nossa api e então passamos ela como parâmetro ao método `request` do chai.
Nesse caso, uma boa prática para a arquitetura da API, é fazer a separação do conjunto da definição das rotas e regras de `middlewares` *(Em um arquivo `app.js` , por exemplo. Que vai ser consumido pelo `chaiHttp` )* , do servidor propriamente dito, que consome essas regras *(Esse continuaria em `server.js` , para utilizarmos em contextos de não-teste)* :

Copiar

```
// ./src/api/app.js
const express = require('express');
const bodyParser = require('body-parser');
const routes = require('./routes');

const app = express();

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

const apiRoutes = express.Router();
apiRoutes.get('/api/posts', routes.getPosts);
apiRoutes.post('/api/users', routes.createUsers);
apiRoutes.post('/api/login', routes.login);

app.use(apiRoutes);

/*
    Detalhe para a exportação do `app`, já que
    precisaremos dele nos testes com `chaiHttp`
*/
module.exports = app;
```

Copiar

```
// ./src/api/server.js
const PORT = process.env.PORT || 8080;
const app = require('./app');

app.listen(PORT, () => console.log(`Conectado na porta ${PORT}`));
```

Após essa separação, voltando em `createUsers.test.js` , podemos testar nossos end-points utilizando a referência deles contida em `./src/api/app.js` :

Copiar

```
// const chai = require('chai');
// const chaiHttp = require('chai-http');
const server = require('../src/api/app');

// chai.use(chaiHttp);

// const { expect } = chai;

// describe('POST /api/users', () => {
//     describe('quando é criado com sucesso', () => {
//         let response = {};

        before(async () => {
            response = await chai.request(server);
        });

        /*
            Veremos adiante o exemplo completo 😉
        */
//     });
//});
```

Após chamarmos o método `request` passando o nosso server, podemos chamar diretamente nossos end-points, simulando chamadas HTTP. Vejamos alguns exemplos disso:

Copiar

```
/*
    Podemos chamar um `GET` que deve consumir nossa api,
    sem que pra isso precisemos subir ela manualmente
*/
const response = await chai.request(server)
  .get('/exemplo');

/*
    Da mesma forma, podemos chamar um `POST` passando um
    `body` e/ou um `header`, por exemplo:
*/
const response = await chai.request(server)
  .post('/favorite-foods')
```

```
  .set('X-API-Key', 'foobar')
  .send({
      name: 'jane',
      favoriteFood: 'pizza'
  });
```

Dessa forma, o plugin nos ajudará a consumir nossa API em nossos testes de maneira muito simples, veja como ficará nosso teste após o `refactor` completo:

Copiar

```
const chai = require('chai');
const chaiHttp = require('chai-http');

const server = require('../src/api/app');

chai.use(chaiHttp);

const { expect } = chai;

describe('POST /api/users', () => {
    describe('quando é criado com sucesso', () => {
        let response = {};

        before(async () => {
            response = await chai.request(server)
                .post('/api/users')
                .send({
                    username: 'jane',
                    password: 'senha123'
                });
        });

        it('retorna o código de status 201', () => {
            /*
                Perceba que aqui temos uma asserção
                específica para o status da `response` 😬
            */
            expect(response).to.have.status(201);
        });

        it('retorna um objeto', () => {
            expect(response.body).to.be.a('object');
        });
```

```
        it('o objeto possui a propriedade "message"', () => {
            expect(response.body).to.have.property('message');
        });

        it('a propriedade "message" possui o texto "Novo usuário
criado com sucesso"',
            () => {
                expect(response.body.message)
                    .to.be.equal('Novo usuário criado com sucesso');
            }
        );
    });
});
```

Antes de executarmos nossos testes, precisamos fazer mais um ajuste. Apesar de estarmos fazendo testes de integração, lembre-se que só queremos testar até nosso `model` , sem deixar que nossa aplicação de fato vá até o banco de dados, isolando o IO.
Para isso, utilizaremos novamente a estratégia de utilizar uma instância do nosso banco de dados em memória. Logo, vamos instalar novamente este pacote.

Copiar

```
npm install -D mongodb-memory-server@6
```

Nosso teste ficará assim:

Copiar

```
// const chai = require('chai');
// const chaiHttp = require('chai-http');
const sinon = require('sinon');
const { MongoClient } = require('mongodb');
const { MongoMemoryServer } = require('mongodb-memory-server');

// const server = require('../src/api/app');

// chai.use(chaiHttp);

// const { expect } = chai;

// describe('POST /api/users', () => {
//    describe('quando é criado com sucesso', () => {
//        let response = {};
        const DBServer = new MongoMemoryServer();
```

```
//         before(async () => {
            const URLMock = await DBServer.getUri();
            const connectionMock = await
MongoClient.connect(URLMock,
                { useNewUrlParser: true, useUnifiedTopology: true }
            );

            sinon.stub(MongoClient, 'connect')
                .resolves(connectionMock);

//             response = await chai.request(server)
//                 .post('/api/users')
//                 .send({
//                     username: 'jane',
//                     password: 'senha123'
//                 });
//         });

        after(async () => {
            MongoClient.connect.restore();
            await DBServer.stop();
        });

//         it('retorna o código de status 201', () => {
//             expect(response).to.have.status(201);
//         });

//         it('retorna um objeto', () => {
//             expect(response.body).to.be.a('object');
//         });

//         it('o objeto possui a propriedade "message"', () => {
//             expect(response.body).to.have.property('message');
//         });

//         it('a propriedade "message" possui o texto "Novo usuário
criado com sucesso"',
//             () => {
//                 expect(response.body.message)
//                     .to.be.equal('Novo usuário criado com
sucesso');
//             }
```

```
//         );
//     });
//});
```

E então podemos rodar nossos testes e, se nossa API estiver respeitando o contrato definido, teremos sucesso:

```
POST /api/users
  quando é criado com sucesso
    ✓ retorna o código de status 201
    ✓ retorna um objeto
    ✓ o objeto possui a propriedade "message"
    ✓ a propriedade "message" possui o texto "Novo usuário criado com sucesso"


4 passing (311ms)
```

Testes - Criação de usuário

---

## Pensando testes para outros contextos

Como e qual teste preciso fazer? Pode não parecer em um primeiro momento, mas como dito anteriormente, a testagem de sistemas complexos fica muito mais simples se pensarmos nos contratos que as situações exigem.
Uma outra forma de medir o nível entre escopo e interação na idealização de um teste, é buscar uma especificidade para que possamos transforma-lo em requisito. Nesse sentido, considere a seguinte pergunta:
Se precisasse fazer o teste manualmente, qual seria o meu processo/ "teste de mesa" ?
Antes de continuar , experimente fazer esse exercício individualmente, pensando o uso do `JWT` em uma API, em um contrato onde caso se tenha um login válido, deva ser possível trazer dados de `posts` (com status `200 - OK` ).
Dependendo da resposta, podemos identificar os tipos de teste que precisaremos fazer:

Copiar

```
## Exemplo de resposta nesse cenário

Utilizaria o `postman`, onde:
   - Faria um login válido na rota `POST /api/login` pra conseguir um `token`;
```

```
   - Aguardaria um status `200 - OK`, acompanhado de um JSON com o
`token`;
   - Testaria a rota `GET /api/posts`, passando esse `token` no
`header`:
     `authorization`;
   - Aguardaria um status `200 - OK`, acompanhado de um JSON com os
`posts`.
```

Agora vamos identificar aqui aquilo que poderia ser um teste unitário e aquilo que caracteriza um teste de integração :

Copiar

```
## Analisando linha a linha

1. Utilizaria o `postman`, onde:
<!--
    Aqui já notamos que o teste requer uma estrutura que depende de
um servidor
    rodando. Esse teste, por tanto, leva em consideração a
`integração` de outros
    elementos, como a definição de um server, rotas e `controllers`;
-->

2. Faria um login válido na rota `POST /api/login`* pra conseguir um
`token`**
<!--
    [*] Se estivéssemos testando isoladamente um `model` que, ao
receber os
    parâmetros de email e password, pode se comportar de uma forma
ou outra,
    esse poderia ser um `teste unitário`;

    [**] Se estivéssemos testando isoladamente o `service` que gera
nosso
    `token`, ou seja, se estamos testando a capacidade de trabalhar
com uma
    função (ou `middleware`) que utiliza internamente o método
`.sign()` do `jwt`
    (que por sua vez, não precisa de um teste unitário por ser uma
biblioteca
    já testada), para encriptar dados aleatórios ou 'mocks', esse
poderia ser um
    `teste unitário`.
```

    Se estamos no entanto, esperando que com base em um conjunto de dados válidos,
    recebamos uma informação específica (através do consumo de uma api), esse é,
    muito provavelmente, um `teste de integração`. Isso, porque esse teste precisa
    que vários componentes da sua api estejam funcionando corretamente: `server`,
    `controller`, `service` e `model`.
-->

3. Aguardaria um status `200 - OK`, acompanhado de um JSON com o `token`;
<!--
    Se estivermos testando isoladamente um `controller`, podemos assumir que esse
    trará um resultado ou outro, o que poderia ser um `teste unitário`.

    Aqui, porém, esse comportamento pressupõe uma ação anterior, ou seja, ele é
    disparado uma vez que a pessoa usuária aciona o login. Sendo parte de um
    `teste de integração`, pois pressupõe a etapa anterior e suas dependências.
-->

4. Testaria a rota `GET /api/posts`, passando esse `token` no `header`:
 `authorization`;
<!--
    Como no item nº2, poderíamos separar testes individuais para cada competência
    nessa pipeline do express, ou seja, poderíamos ter `testes unitários` para,
    por exemplo:
        - Middleware: `auth`, que validaria tokens;
        - Service: `getUser`, que validaria emails e senhas;
        - Model: `findUser`, onde traríamos dados de pessoas usuárias no banco;
        - Service: `getAllPosts`, onde testaríamos alguma validação ou regra;

```
        - Model: `findPosts`, onde traríamos dados de posts do
banco;
        - Controller: `getPosts`, testando dados de retorno;

    Pensando o todo, esse teste depende dos demais, pois depende do
`token` para
    funcionar corretamente. Aqui novamente, sendo parte de um `teste
de
    integração`.
-->

5. Aguardaria um status `200 - OK`, acompanhado de um JSON com os
`posts`.
<!--
    Caso aqui não estejamos testando um `controller`, então esse
passo só faria
    sentido como uma asserção/afirmação ao final de um `teste de
integração`.
-->
```

---

## Cobertura de testes

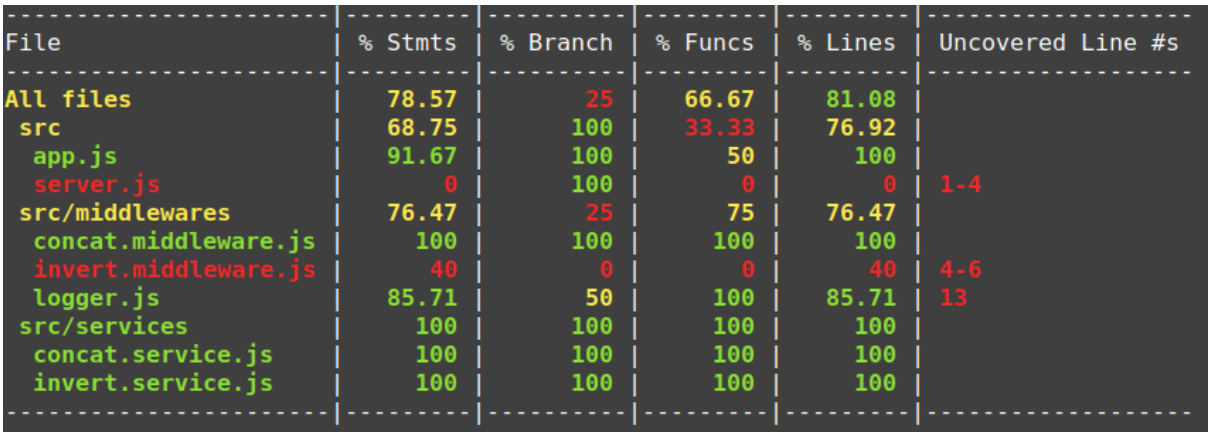
Uma forma de acompanhar o quão bem estamos conseguindo exercitar a testagem do nosso sistema pode ser feita através de relatórios de cobertura . Boa parte das suites de teste das linguagens de programação possui uma forma de gerar um relatório desse tipo, no caso do `NodeJS` , conseguimos gerar esses relatórios, tanto para os testes feitos no `jest` quanto no `mocha` (utilizando uma ferramenta chamada `nyc` ).
Esses relatórios checam, se para um escopo de arquivos definidos (aqui podemos pensar o conteúdo da nossa aplicação, excluindo bibliotecas e arquivos de configuração), os seus testes são capazes de rodar todas as linhas dos arquivos relacionados , o que gera uma porcentagem total de cobertura para aquele escopo.

### Critérios relevantes

De forma geral (também para outras linguagens de programação), suites de testes geram relatórios de cobertura segundo alguns critérios básicos , os mais relevantes para nosso contexto são:

- Cobertura de Funções / *Function Coverage* : Cada função/sub-rotina do script foi acionado/chamado?
- Cobertura de Afirmações / *Statement Coverage* : Cada afirmação/definição/comando do script foi executado?
- Cobertura de Ramificações / *Branch Coverage* : Cada `situação` de ramificação do código *(aqui podemos assumir um script condicional, como um `if { /*situação A*/ } else { /*situação B*/ }` )* foi executada?

No nosso contexto, ambas ferramentas ( `jest` e `nyc` ) vão utilizar relatórios do `Instanbul` , por tanto em uma situação de exemplo, um relatório gerado em uma das nossas ferramentas, deve retornar uma tabela semelhante a essa:



| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
|---|---|---|---|---|---|
| All files | 78.57 | 25 | 66.67 | 81.08 | |
| src | 68.75 | 100 | 33.33 | 76.92 | |
| app.js | 91.67 | 100 | 50 | 100 | |
| server.js | 0 | 100 | 0 | 0 | 1-4 |
| src/middlewares | 76.47 | 25 | 75 | 76.47 | |
| concat.middleware.js | 100 | 100 | 100 | 100 | |
| invert.middleware.js | 40 | 0 | 0 | 40 | 4-6 |
| logger.js | 85.71 | 50 | 100 | 85.71 | 13 |
| src/services | 100 | 100 | 100 | 100 | |
| concat.service.js | 100 | 100 | 100 | 100 | |
| invert.service.js | 100 | 100 | 100 | 100 | |

Exemplo de um relatório de cobertura com `jest`/`nyc`

- `File` (Arquivo): Retorna a estrutura do escopo analisado, cada linha é referente a pasta ou arquivo específico, no nosso caso, a cobertura esta analisando todos arquivos `*.js` contidos em `./src` , que fica na raiz do projeto;
- `Stmts` (Statements/Afirmações): Retorna os percentuais da *cobertura de afirmações executadas* que citamos anteriormente, no nosso caso, é possível assumir que o arquivo `middlewares/invert.middleware.js` não executou todas as suas definições/afirmações . Note ainda, que em `Uncovered Line #s` *(Linhas não-cobertas)* , o relatório identifica quais as linhas do arquivo não foram executadas, no nosso caso, as linhas de 4 a 6 não foram executadas em nenhum momento quando esse arquivo foi referenciado nos nossos testes (via `require()` , ou via parâmetro de configuração, o que veremos mais a frente);
- `Branch` (Ramo): Retorna o percentual de *situações de ramificação cobertos* . Se observarmos no arquivo `logger.js` , existe um percentual de `50%` de situações não-cobertas (ou seja, situações que

não foram testadas em nenhum momento), o relatório ainda aponta a linha `13` como a linha não-coberta, aqui podemos assumir que essa linha faz parte do resultado de um script condicional (como um `if{}else` ). Se no arquivo não houverem situações de ramificação, o retorno é `100%` .
Detalhe , o relatório vai considerar uma `branch` , mesmo que não haja nenhuma situação de `else` para ela, ex:

Copiar

```
const debug = true;

module.exports = (req, res, next) => {
  if(debug){
    res.on('finish', () => {
      console.log({
        method: req.method,
        endPoint: req.originalUrl,
        status: res.statusCode
      })
    });
  }
  /*
    No caso desse `if`, não existe cobertura pra uma situação onde `debug`
    é falso, então, ainda que um teste cubra 100% desse código, o retorno
    em `% Branch` para esse arquivo, será 50%;
  */

  next();
}
```

- `Funcs` (Functions/Funções): Retorna o percentual de *funções executadas* nos arquivos. Em `middlewares/invert.middleware.js` e `server.js` , podemos assumir que nenhuma das funções desses arquivos foi executada nos nossos testes. Em `server.js` , ainda, é possível identificar que o arquivo não foi nem mesmo referenciado nos testes, já que nenhuma definição do mesmo foi executada (Coluna `% Stmts` );
- `Lines` (Linhas): Retorna o percentual de linhas executadas nos arquivos, no caso de `All files` , esse valor representa o total de

cobertura da sua suite de testes , que no nosso caso representa `81,08%` de cobertura total, dado os problemas apresentados.

### Como gerar uma cobertura de testes no meu ambiente?

Como visto acima, tanto no o `jest` quanto no `nyc` , é possível gerar um relatório de cobertura padrão. E a depender da forma da utilização de cada um, é possível ainda trazer relatórios em diferentes formatos (como em `html` , por exemplo).
A princípio, vamos passar pelos comandos básicos para execução e a descrição de cada um, após isso, passaremos pela API de exemplo que utilizamos hoje, gerando um relatório de cobertura utilizando o `nyc` :

#### Comando básico

No `jest` , utilizamos o parâmetro `--coverage` (como em `jest --coverage` ), assim, podemos pensar a seguinte configuração de `scripts` no `package.json` :

Copiar

```
...
"scripts": {
    ...
    "test": "jest ./tests",
    "test:coverage": "npm test -- --coverage",
    ...
},
...
```

Dessa forma, conseguimos ter um script próprio para gerar esse relatório, que rodamos com `npm run test:coverage` .
No `mocha` , antes, temos que instalar a biblioteca `nyc` *(que é a `cli` - interface de linha de comando, do `Instanbul` )* :

Copiar

```
npm i -D nyc
```

Após isso, a utilização também é bastante simples, utilizaremos o `nyc` , passando como parâmetro o comando que utilizaremos para os testes em `mocha` , exemplo: `nyc mocha ./tests --recursive` . Dessa forma, conseguimos fazer uma configuração de `scripts` similar ao do `jest` :

Copiar

```
...
"scripts": {
    ...
```

```
    "test": "mocha ./tests --recursive",
    "test:coverage": "nyc npm test",
    ...
},
...
```

Personalizando o escopo de cobertura

Por padrão, os `reporters` vão fazer a cobertura dos arquivos que são referenciados nos seus testes. Para trazer a porcentagem de cobertura dentro de um escopo fixo você pode:
No `jest` , de duas formas:

- Utilizando um arquivo de configuração `jest.config.js` (que deve ser referenciado via cli com o parâmetro `--config=<seuArquivoDeConfig>` ). Esse arquivo pode receber uma propriedade `collectCoverageFrom` , contendo o padrão a ser respeitado;
- Utilizando o mesmo comando, via cli: `--collectCoverageFrom` , da seguinte forma:

Copiar

```
...
"scripts": {
    ...
    "test": "jest ./tests",
    "test:coverage": "npm test -- --coverage
--collectCoverageFrom='src/**/*.js'",
    ...
},
...
```

No `nyc` , de duas formas:

- Utilizando um arquivo de configuração `nyc.config.js` na raiz do projeto. Esse arquivo pode receber uma propriedade `include` , contendo o padrão a ser respeitado;
- Utilizando o mesmo comando, via cli: `--include` , da seguinte forma:

Copiar

```
...
"scripts": {
    ...
    "test": "mocha ./tests --recursive",
    "test:coverage": "nyc --include='src/**/*.js' npm run test",
```

```
    ...
},
...
```

- É possível ainda, via cli, utilizar o parâmetro `--all` para coletar a cobertura de todos os arquivos (mesmo os que não tem referência nos testes).

Notem aqui, que estamos colocando nosso código fonte dentro de uma pasta `./src` , para que não seja necessário criar uma lista de exclusão de cobertura (para pasta `node_modules` ou a própria pasta `tests` , por exemplo), nesse sentido, também é importante manter a pasta `tests` na raiz, pelo mesmo motivo;

### Rodando um teste de cobertura no projeto atual

Seguindo os passos anteriores, basta adicionar um `script` no nosso `package.json` contendo o escopo de cobertura:

Copiar

```
...
"scripts": {
    ...
    "test": "mocha ./tests/**/*$NAME*.test.js --exit",
    "test:coverage": "nyc --include='src/**/*.js' npm run test",
    ...
},
...
```

Após isso, basta rodar o comando `npm run test:coverage` ;

```
----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |   67.92 |       10 |   54.55 |   73.47 |
 api            |     100 |      100 |     100 |     100 |
  app.js        |     100 |      100 |     100 |     100 |
  routes.js     |     100 |      100 |     100 |     100 |
 controllers    |   43.48 |       10 |   33.33 |      50 |
  createUser.js |      75 |       50 |     100 |   85.71 | 12
  login.js      |   16.67 |        0 |       0 |      20 | 4-16
  posts.js      |   66.67 |      100 |       0 |   66.67 | 21
 models         |   76.47 |      100 |    62.5 |   81.25 |
  connection.js |      75 |      100 |   66.67 |      75 | 13-14
  user.js       |   77.78 |      100 |      60 |    87.5 | 9
----------------|---------|----------|---------|---------|-------------------
```

Cobertura do projeto de exemplo `jwt-base`

Pronto! Agora já é possível identificar quais pontos ainda precisam de uma cobertura de testes apropriada no seu projeto!

---

## Agora, a prática

Vamos juntar tudo o que aprendemos até aqui e exercitar mais ainda nosso aprendizado!
Iremos dar seguimento ao projeto visto durante a aula. Para isso, certifique-se de tê-lo clonado deste link .
Utilizando o processo de TDD, você irá implementar, a partir de testes, um endpoint para busca de dados de um usuário a partir do seu ID: `GET /api/users/:userId` . Cada exercício conterá um dos requisitos a ser implementado.
Lembre-se de utilizar os conceitos visto até aqui:

- Utilize TDD , ou seja, inicie um requisito escrevendo as asserções necessárias para validar aquele cenário, em seguida implemente o código necessário e por fim, faça os ajustes necessários para que o teste fique compatível com sua implementação.
- Nos testes, isole o IO utilizando a técnica de subir o banco de dados em memória.
- Utilize o plugin do `chai` de requests HTTP para consumir seus endpoint diretamente em seus testes.

Exercício 1 : O endpoint deverá ser autenticado, exigindo o envio de um token no header da requisição. Caso não seja passado um token, o endpoint deverá retornar:

- Código de status `400 - Not Found` ;
- Mensagem de erro no body da response com o texto `Token não encontrado ou informado` .

Lembre-se de utilizar o `middleware` de autenticação para validação do JWT.
Exercício 2 : O usuário poderá ver somente os seus próprios dados. Ou seja, ao receber uma request, deverá ser comparado se o ID vindo no parâmetro é o mesmo do armazenado no token. Para isso, utilize o `middleware` de autenticação para recuperar o ID dentro do token. Caso não seja, a API deverá retornar:

- Código de status `401 - Unauthorized` ;
- Mensagem no body da response com o texto `Acesso negado` .

Exercício 3 : Caso o usuário esteja autenticado corretamente e esteja solicitando os dados de seu próprio usuário, o sistema deverá retornar:

- Os dados da pessoa usuária em um objeto no corpo ( `body` ) da resposta ( `response` );
- Código de status `200 - OK` .

---

# Soluções

Utilizando o processo de TDD, você irá implementar, a partir de testes, um endpoint para busca de dados de um usuário a partir do seu ID: `GET /api/users/:userId` . Cada exercício conterá um dos requisitos a ser implementado.
Exercício 1 : O endpoint deverá ser autenticado, exigindo o envio de um token no header da requisição. Caso não seja passado um token, o endpoint deverá retornar:

- Código de status 400;
- Mensagem de erro no body da response com o texto `Token não encontrado ou informado` .

api/server.js

Copiar

```
// const express = require('express');
// const bodyParser = require('body-parser');
// const routes = require('./routes');
// const validateJWT = require('./auth/validateJWT');

// const port = process.env.PORT || 8080;

// const app = express();

// app.use(bodyParser.urlencoded({ extended: false }));
// app.use(bodyParser.json());

// const apiRoutes = express.Router();
// apiRoutes.get('/api/posts', validateJWT, routes.getPosts);
apiRoutes.get('/api/users/:userId', validateJWT,
routes.findUserById);
// apiRoutes.post('/api/users', routes.createUsers);
```

```
// apiRoutes.post('/api/login', routes.login);

// app.use(apiRoutes);

// app.listen(port);
// console.log('conectado na porta ' + port);

// module.exports = app;
```

tests/findUserById.js

Copiar

```
const chai = require('chai');

const chaiHttp = require('chai-http');
chai.use(chaiHttp);

const { expect } = chai;

const server = require('../api/server');

const EXAMPLE_ID = '605de6ded1ff223100cd6aa1'

describe('GET /api/users/:userId', () => {
    describe('Quando não é passado um JWT para autenticação', () =>
{
        let response;
        before(async () => {
            response = await chai.request(server)
                .get(`/api/users/${EXAMPLE_ID}`);
        });

        after(() => {
            //
        });

        it('retorna código de status "400"', () => {
            expect(response).to.have.status(400);
        });

        it('retorna um objeto no body', () => {
            expect(response.body).to.be.an('object');
        });
```

```
        it('objeto de resposta possui a propriedade "error"', () =>
{
            expect(response.body).to.have.property('error');
        });

        it('a propriedade "error" possui a mensagem "Token não
encontrado ou informado"', () => {
            expect(response.body.error).to.be.equal('Token não
encontrado ou informado');
        });
    });
});
```

Exercício 2 : O usuário poderá ver somente os seus próprios dados. Ou seja, ao receber uma request, deverá ser comparado se o ID vindo no parâmetro é o mesmo do armazenado no token. Para isso, utilize o middleware de autenticação para recuperar o ID dentro do token. Caso não seja, a API deverá retornar:

- Código de status 401;
- Mensagem no body da response com o texto `Acesso negado` .

controllers/findUserById.js

Copiar

```
module.exports = async (req, res) => {
  if (req.params.userId !== req.user._id) {
    res.status(401).json({ error: 'Acesso negado' });
  }
};
```

tests/findUserById.js

Copiar

```
const chai = require('chai');
const sinon = require('sinon');

const chaiHttp = require('chai-http');
chai.use(chaiHttp);

const { expect } = chai;

const server = require('../api/server');

const { MongoClient } = require('mongodb');
const { MongoMemoryServer } = require('mongodb-memory-server');
```

```
const EXAMPLE_ID = '605de6ded1ff223100cd6aa1'

describe('GET /api/users/:userId', () => {
    // describe('Quando não é passado um JWT para autenticação', ()
=> {
    //     let response;
    //     before(async () => {
    //         response = await chai.request(server)
    //             .get(`/api/users/${EXAMPLE_ID}`);
    //     });

    //     after(() => {
    //         //
    //     });

    //     it('retorna código de status "400"', () => {
    //         expect(response).to.have.status(400);
    //     });

    //     it('retorna um objeto no body', () => {
    //         expect(response.body).to.be.an('object');
    //     });

    //     it('objeto de resposta possui a propriedade "error"', ()
=> {
    //         expect(response.body).to.have.property('error');
    //     });

    //     it('a propriedade "error" possui a mensagem "Token não
encontrado ou informado"', () => {
    //         expect(response.body.error).to.be.equal('Token não
encontrado ou informado');
    //     });
    // });

    describe('Quando o usuário solicita informações de outro
usuário', () => {
        let response;
        const DBServer = new MongoMemoryServer();

        before(async () => {
            const connectionMock = await DBServer.getUri()
```

```
            .then(URLMock => MongoClient.connect(
                URLMock,
                { useNewUrlParser: true, useUnifiedTopology: true }
            ));

            sinon.stub(MongoClient, 'connect')
                .resolves(connectionMock);

            connectionMock.db('jwt_exercise')
                .collection('users')
                .insertOne({
                    _id: EXAMPLE_ID,
                    username: 'fake-user',
                    password: 'fake-password',
                    name: 'fake-name',
                    birthdate: '01/01/1960',
                    biography: 'fake-biography',
                })

            const token = await chai.request(server)
                .post('/api/login')
                .send({
                    username: 'fake-user',
                    password: 'fake-password'
                })
                .then((res) => res.body.token);

            const OTHER_EXAMPLE_ID = '565de6ded1ff223100cd6aa2'

            response = await chai.request(server)
                .get(`/api/users/${OTHER_EXAMPLE_ID}`)
                .set('authorization', token);
        });

        after(async () => {
            MongoClient.connect.restore();
            await DBServer.stop();
        });

        it('retorna código de status "401"', () => {
            expect(response).to.have.status(401);
        });
```

```
        it('retorna um objeto no body', () => {
            expect(response.body).to.be.an('object');
        });

        it('objeto de resposta possui a propriedade "error"', () =>
{
            expect(response.body).to.have.property('error');
        });

        it('a propriedade "error" possui a mensagem "Acesso
negado"', () => {
            expect(response.body.error).to.be.equal('Acesso
negado');
        });
    });
});
```

Exercício 3 : Caso o usuário esteja autenticado corretamente e esteja solicitando os dados de seu próprio usuário, o sistema deverá retornar:

- Os dados da pessoa usuária em um objeto no body da response;
- Código de status 200.

models/user.js

Copiar

```
// const connect = require('./connection');
const ObjectId = require('mongodb').ObjectId;

// const registerUser = async (username, password, name, birthdate,
biography) =>
//   connect().then((db) =>
//     db.collection('users').insertOne({ username, password, name,
birthdate, biography })
//   ).then(result => result.ops[0].username );

// const findUser = async (username) =>
//   connect().then((db) => db.collection('users').findOne({
username }));

const findUserById = async (userId) =>
  connect().then((db) => db.collection('users').find({ _id:
ObjectId(userId) }));
```

```
// module.exports = { registerUser, findUser, findUserById };
```

controllers/findUserById.js

Copiar

```
const Model = require('../models/User');

// module.exports = async (req, res) => {
  // if (req.params.userId !== req.user._id) {
  //   res.status(401).json({ error: 'Acesso negado' });
  // }

  const {
    name,
    username,
    birthdate,
    biography
  } = await Model.findUserById(req.params.userId);

  res.status(200).json({
    name,
    username,
    birthdate,
    biography,
  });
// };
```

tests/findUserById.js

Copiar

```
// const chai = require('chai');
// const sinon = require('sinon');

// const chaiHttp = require('chai-http');
// chai.use(chaiHttp);

// const { expect } = chai;

// const server = require('../api/server');

// const { MongoClient } = require('mongodb');
// const { MongoMemoryServer } = require('mongodb-memory-server');

// const EXAMPLE_ID = '605de6ded1ff223100cd6aa1'
```

```
// describe('GET /api/users/:userId', () => {
//     describe('Quando não é passado um JWT para autenticação', ()
=> {
//         let response;
//         before(async () => {
//             response = await chai.request(server)
//                 .get(`/api/users/${EXAMPLE_ID}`);
//         });

//         after(() => {
//             //
//         });

//         it('retorna código de status "400"', () => {
//             expect(response).to.have.status(400);
//         });

//         it('retorna um objeto no body', () => {
//             expect(response.body).to.be.an('object');
//         });

//         it('objeto de resposta possui a propriedade "error"', ()
=> {
//             expect(response.body).to.have.property('error');
//         });

//         it('a propriedade "error" possui a mensagem "Token não
encontrado ou informado"', () => {
//             expect(response.body.error).to.be.equal('Token não
encontrado ou informado');
//         });
//     });

//     describe('Quando a pessoa usuária solicita informações de
outra pessoa usuária', () => {
//         let response;
//         const DBServer = new MongoMemoryServer();

//         before(async () => {
//             const connectionMock = await DBServer.getUri()
//             .then(URLMock => MongoClient.connect(
//                 URLMock,
```

```
//                 { useNewUrlParser: true, useUnifiedTopology: true
}
//             ));

//             sinon.stub(MongoClient, 'connect')
//                 .resolves(connectionMock);

//             connectionMock.db('jwt_exercise')
//                 .collection('users')
//                 .insertOne({
//                     _id: EXAMPLE_ID,
//                     username: 'fake-user',
//                     password: 'fake-password',
//                     name: 'fake-name',
//                     birthdate: '01/01/1960',
//                     biography: 'fake-biography',
//                 })

//             const token = await chai.request(server)
//                 .post('/api/login')
//                 .send({
//                     username: 'fake-user',
//                     password: 'fake-password'
//                 })
//                 .then((res) => res.body.token);

//             const OTHER_EXAMPLE_ID = '565de6ded1ff223100cd6aa2'

//             response = await chai.request(server)
//                 .get(`/api/users/${OTHER_EXAMPLE_ID}`)
//                 .set('authorization', token);
//         });

//         after(async () => {
//             MongoClient.connect.restore();
//             await DBServer.stop();
//         });

//         it('retorna código de status "401"', () => {
//             expect(response).to.have.status(401);
//         });

//         it('retorna um objeto no body', () => {
```

```
//             expect(response.body).to.be.an('object');
//         });

//         it('objeto de resposta possui a propriedade "error"', () =>
{
//             expect(response.body).to.have.property('error');
//         });

//         it('a propriedade "error" possui a mensagem "Acesso
negado"', () => {
//             expect(response.body.error).to.be.equal('Acesso
negado');
//         });
//     });

    describe('Quando a pessoa usuária é encontrada com sucesso', ()
=> {
        let response;
        const DBServer = new MongoMemoryServer();

        before(async () => {
            const connectionMock = await DBServer.getUri()
                .then(URLMock => MongoClient.connect(
                    URLMock,
                    { useNewUrlParser: true, useUnifiedTopology:
true }
                ));

            sinon.stub(MongoClient, 'connect')
                .resolves(connectionMock);

            connectionMock.db('jwt_exercise')
                .collection('users')
                .insertOne({
                    _id: EXAMPLE_ID,
                    username: 'fake-user',
                    password: 'fake-password'
                })

            const token = await chai.request(server)
                .post('/api/login')
                .send({
                    username: 'fake-user',
```

```
                    password: 'fake-password'
                })
                .then((res) => res.body.token);

            response = await chai.request(server)
                .get(`/api/users/${EXAMPLE_ID}`)
                .set('authorization', token);
        });

        after(async () => {
            MongoClient.connect.restore();
            await DBServer.stop();
        });

        it('retorna código de status "200"', () => {
            expect(response).to.have.status(200);
        });

        it('retorna um objeto no body', () => {
            expect(response.body).to.be.an('object');
        });
    });
// });
```