UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Felipe Rosa da Silva

Testes *backend* automatizados no Sistema Online de Distribuição de Disciplinas

Uberlândia, Brasil

2022

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Felipe Rosa da Silva

Testes *backend* automatizados no Sistema Online de Distribuição de Disciplinas

Trabalho de conclusão de curso apresentado à Faculdade de Gestão e Negócios da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Gestão da Informação.

Orientador: Bruno Augusto Nassif Travençolo

Universidade Federal de Uberlândia – UFU

Faculdade de Gestão e Negócios

Bacharelado em Gestão da Informação

Uberlândia, Brasil

2022

Agradecimentos

Gostaria de agradecer aos professores e amigos que compartilharam seus conhecimentos e contribuíram para o desenvolvimento deste trabalho, em especial ao colega Matheus dos Santos Mendes pela contribuição no projeto.

Resumo

Todo e qualquer tipo de software está propenso a vários tipos de erros que afetam o seu funcionamento, principalmente durante o seu desenvolvimento. Diante disso, é imprescindível que os sistemas passem por diversos testes, para que os problemas sejam evidenciados e corrigidos pelos desenvolvedores. A fim de tornar esse trabalho mais efetivo, é importante considerar as tecnologias existentes para auxiliar, como a automação, e ter um planejamento bem estruturado, para abranger a maior quantidade de cenários possíveis no sistema em específico. O foco deste trabalho é discorrer sobre os testes de software, mostrando os tipos, a sua importância e as vantagens de se fazer a automação. Além disso, é demonstrado alguns exemplos práticos do teste automatizado feito no *backend* do Sistema Online de Distribuição de Disciplinas (SODD) da Faculdade de Computação (FACOM) da Universidade Federal de Uberlândia.

Palavras-chave: Teste de Software, Automação, Qualidade, Sistema Online de Distribuição de Disciplinas.

Lista de ilustrações

Figura 1 – Fluxograma de execução do teste de software. Figura adaptada de Trevisan Junior (2007). 10
Figura 2 – Exemplo do comando “Describe” utilizado para iniciar o desenvolvimento dos testes. 15
Figura 3 – Código do caso de teste “Criar uma turma”. 17
Figura 4 – Código do caso de teste que valida a funcionalidade de criar uma turma com duplicidade. 18
Figura 5 – Código do caso de teste que valida a funcionalidade de criar uma turma com o campo “ch” vazio. 19
Figura 6 – Código do caso de teste que valida a funcionalidade de criar uma turma com o campo `"codigo_disc"` de uma disciplina inexistente. 20
Figura 7 – Código do caso de teste que valida a funcionalidade de criar uma turma com o campo “semestre” maior que 2. 21
Figura 8 – Mensagem que aparece no terminal quando a automação encontra algum erro. 22
Figura 9 – Resumo consolidado de todos os casos de testes apresentados no terminal após a execução. 22
Figura 10 – Resultado individual de sucesso ou falha de cada caso de teste apresentado no terminal após a execução. 23
Figura 11 – Método “afterAll”. 23

Sumário

1 INTRODUÇÃO **6**
1.1 Objetivos **7**
1.1.1 Objetivo Geral 7
1.1.2 Objetivos Específicos 7
1.2 Organização do Trabalho **7**

2 REFERENCIAL TEÓRICO **8**
2.1 Tipos de teste de software **8**
2.2 Fases do teste de software **9**

3 FERRAMENTAS E MÉTODOS UTILIZADOS **12**
3.1 Software **12**
3.2 Ferramentas **12**
3.3 Jest **13**

4 RESULTADOS **14**

5 CONCLUSÃO **24**

REFERÊNCIAS **25**

1 Introdução

Hoje em dia, a tecnologia está por toda parte e em quase tudo que usamos, como no comércio, produção de alimentos, nos estudos, etc. Com isso, o desenvolvimento de softwares foi se intensificando cada vez mais e a necessidade de serem mais eficientes e com o mínimo de defeitos logo apareceu. Dessa forma, como erros durante a codificação dos sistemas acontecem em praticamente todos os casos, surgiu a prática de testá-los metodicamente, a fim de resolver esse problema.

Para realizar testes em um programa como um todo, é necessário simular e avaliar todas as funcionalidades e recursos que o usuário final terá acesso, o que requer muito tempo, além de um planejamento e conhecimento para se encontrar possíveis erros ou melhorias. Dessa maneira, para maximizar o rendimento desse trabalho e ele se tornar mais significante, foram criados os testes automatizados, que proporcionam alguns benefícios, como a diminuição do custo, do tempo e, principalmente, de ser possível retestar quantas vezes precisar sem ter todo o "trabalho braçal" (COLLINS; LOBÃO, 2010).

Por outro lado, muitos *bugs* só são encontrados em cenários específicos e que geralmente não são detectados pela automação, e sim pelos testes manuais, já que a flexibilidade é maior e que não necessariamente segue o fluxo ideal da funcionalidade do sistema (BARBOSA; TORRES, 2011). Por isso que ambas as formas de se realizar os testes são úteis e importantes durante o desenvolvimento de um software.

Dentre todos os tipos de testes, o único que é mais comumente utilizado no automatizado é o teste funcional, já que ele é feito seguindo o fluxo ideal da usabilidade do sistema. Atrelando isso às vantagens proporcionadas por ele ser feito de forma automática, fica muito rápido e prático identificar problemas grotescos ou já identificados anteriormente, que podem ser gerados após atualizar a versão do sistema, algo que pode acontecer durante o desenvolvimento.

Em relação aos objetivos do trabalho, o intuito inicial é apresentar uma parte teórica dos testes de software e, por fim, demonstrar alguns exemplos práticos da automação. É importante destacar que dentre as diferentes formas e finalidades de fazer a automação, o teste *backend* será o foco deste trabalho, que é o teste voltado a avaliar a codificação do sistema. Com isso, as telas que aparecem para o usuário final (*frontend*) não serão utilizadas nesse caso.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo deste trabalho é a implantação de testes automatizados do software Sistema Online de Distribuição de Disciplinas (SODD) da Faculdade de Computação (FACOM)-UFU.

1.1.2 Objetivos Específicos

- Estudar as práticas recentes de desenvolvimento de testes automatizados de software.
- Identificar funcionalidades no *backend* do SODD para serem testadas.

1.2 Organização do Trabalho

Este trabalho dispõe de uma introdução falando sobre os testes de software e da automação desse processo, além de algumas vantagens de utilizá-los. Posteriormente, vem o referencial teórico, que explica sobre os tipos e as fases do teste, ressaltando a importância do planejamento e da aplicação dos métodos de forma organizada. Na sequência, são exibidas as ferramentas e métodos utilizados nos testes automatizados realizados no sistema e uma descrição do software testado. Depois são apresentados os resultados dos testes automatizados feitos sobre uma funcionalidade em específico do sistema, com algumas imagens para ilustrar os códigos. Por último, é realizado a conclusão do trabalho.

2 Referencial teórico

O teste de software consiste em avaliar um sistema por meio da sua utilização a fim de verificar se ele possui todas as funcionalidades e requisitos previamente especificados sem apresentar nenhum erro ou defeito (DIAS NETO, 2014). Sendo assim, o objetivo é de procurar por possíveis falhas ou melhorias e evidenciá-las para que possam ser corrigidas e o software atingir seu nível de excelência. Segundo Dias Neto (2014), esse processo das atividades de teste tem uma característica "destrutiva" e não "construtiva", já que ele busca por todas as falhas possíveis antes de apresentar ao usuário final.

Durante o processo de testes de um software, não é feito apenas o uso do sistema de forma aleatória em busca de erros. Existem várias atividades que são importantes nesse momento, tais como: planejamento e controle, documentação dos casos de teste, escolha das condições de teste, avaliar e documentar os resultados atingidos.

2.1 Tipos de teste de software

Da mesma forma que é de extrema importância ter um planejamento na hora de realizar os testes em um software, também é fundamental saber que existem vários tipos de testes. Nesse sentido, é interessante estudar qual deles será aplicado, já que a sua utilidade e efetividade pode variar de acordo com a necessidade e a particularidade do sistema. Dentre eles, os mais utilizados são (LUFT, 2012):

- O teste de instalação e configuração serve para avaliar e verificar se o sistema funciona em diferentes tipos de software e hardware, de acordo com as configurações especificadas.
- O teste de usabilidade é voltado para analisar a aparência visual e funcional do sistema, a fim de torná-lo mais intuitivo e mais fácil de se utilizar.
- O teste de volume tem o objetivo de avaliar o comportamento do software com uma grande quantidade de dados.
- O teste de segurança é para confirmar que todos os métodos de proteção utilizados estejam funcionando corretamente, impedindo acessos indevidos às funcionalidades do sistema.
- O teste de performance é aplicado para verificar se o sistema apresenta um bom desempenho das suas funcionalidades. É importante lembrar que esse teste é feito

em um aparelho que tenha os requisitos de configurações de software e hardware especificados.

- O teste funcional, como o próprio nome já diz, é aquele que busca garantir que as funcionalidades do sistema estejam corretas. Em outras palavras, ele valida se o sistema que foi desenvolvido apresenta todos os requisitos funcionais planejados anteriormente. Porém, esse tipo de teste não se importa em "verificar como ocorrem internamente os processamentos no software, mas se o algoritmo inserido no software produz os resultados esperados" (BARTIÉ, 2002, p. 105).

2.2 Fases do teste de software

Na hora de realizar os testes do sistema como um todo, é interessante pensar em certos níveis de profundidade do teste, a fim de tornar o processo mais organizado e sequencial. Nesse sentido, é possível separar os testes em diferentes fases, definindo quando será realizado cada caso de teste, para buscar a maior variedade de cenários possíveis (LUFT, 2012).

O teste de unidade é a fase em que cada componente do código do sistema é testado separadamente, sendo assim, a preocupação está na menor unidade do projeto do software.

Posteriormente, vem o teste de integração, na qual as unidades que foram validadas isoladas, agora são agrupadas de acordo com a arquitetura do sistema para serem testadas. Nesse sentido, o objetivo é encontrar possíveis erros de integração entre os componentes previamente testados.

Depois da integração do programa, é feito o teste de sistema, que tem o intuito de testar se o software interage corretamente com o banco de dados, com o hardware, entre outros, e que satisfaz seus pré-requisitos.

O teste de aceitação é feito pelo usuário final, ou seja, o software é disponibilizado para ele utilizar e validar se as funcionalidades presentes atendem os requisitos solicitados ou sugerir melhorias em alguns aspectos.

O teste de regressão é feito após uma atualização no sistema e consiste em fazer o reteste das funcionalidades, para garantir que o que estava funcionando anteriormente não foi afetado pela manutenção. Conforme Trevisan Junior (2007) ilustra com um fluxograma um processo genérico de teste, é possível identificar as fases de teste e a sequência em que ela é realizada. Além disso, ele mostra que em cada fase é feito um ciclo entre o teste e as correções aplicadas (Figura 1).

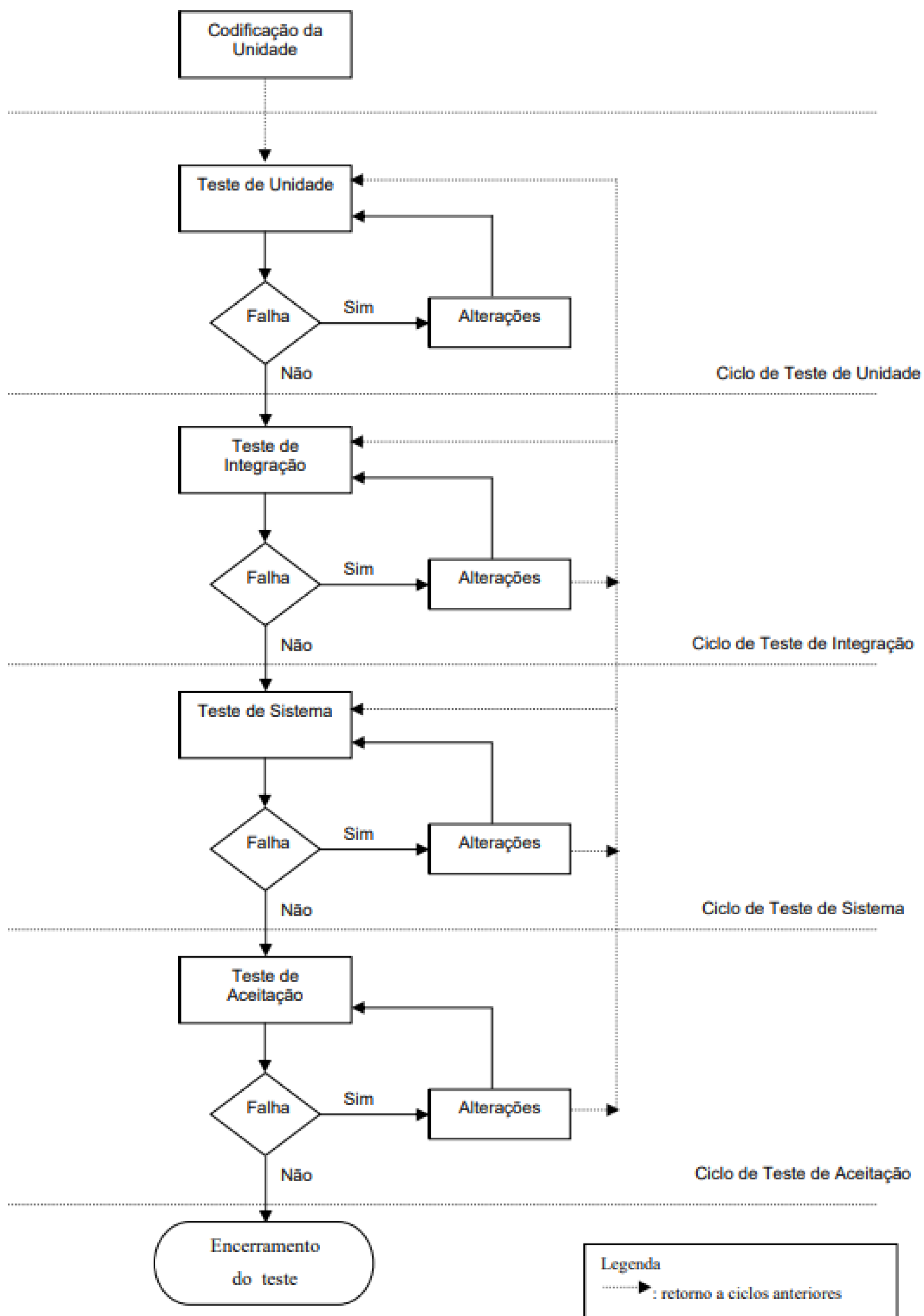

Figura 1 – Fluxograma de execução do teste de software. Figura adaptada de Trevisan Junior (2007).

Outro ponto importante de se observar é que, após as alterações feitas, deve ser levado em consideração o reteste dos ciclos anteriores, que seria o teste de regressão. Dessa

forma, é possível verificar se as modificações realizadas para corrigir os erros encontrados em cada fase, não geraram problemas que são detectados em fases passadas. Por outro lado, o processo de teste como um todo começa a ficar bastante oneroso, já que os erros e correções podem acontecer inúmeras vezes. Nesse sentido, é notório que a existência de uma automação para realizar grande parte desse trabalho quantas vezes for necessário e de forma rápida é fundamental.

3 Ferramentas e métodos utilizados

Este capítulo apresenta o software que será testado e as ferramentas utilizadas na automação.

3.1 Software

O Sistema Online de Distribuição de Disciplinas (SODD) é um sistema feito para auxiliar na gestão dos cursos da FACOM da Universidade Federal de Uberlândia, em relação à divisão da carga horária de cada professor. Sendo assim, ele permite o cadastro dos professores, disciplinas, turmas, cursos, semestres, horários a fim de proporcionar a funcionalidade de distribuição dos professores nas disciplinas de cada turma.

Na sua primeira proposta, o SODD foi desenvolvido utilizando a linguagem Java por Locatelli (2015). Porém, depois disso, outros desenvolvedores começaram a incrementar e dar manutenção no sistema, sem necessariamente seguir um padrão. Com isso, na medida que o projeto foi avançando, começou a se tornar inviável dar sequência no desenvolvimento, pelas formas e técnicas diferentes que foram utilizadas na construção do sistema. Além disso, inicialmente não havia nenhuma proposta relacionada a testes de software, que foi surgir posteriormente com Fernandes (2020), utilizando o Cucumber, Selenium WebDriver e o JUnit para a criação da automação.

Nesse sentido, Mendes (2022) veio com uma nova proposta de implementação para o SODD, visando utilizar novas tecnologias e manter um desenvolvimento padronizado, para que as futuras novas funcionalidades e manutenções sejam mais fáceis de serem executadas. Além disso, Mendes (2022) também implantou os testes automatizados no projeto, na qual seu padrão de desenvolvimento foi utilizado neste trabalho.

No momento em que este trabalho foi feito, o software em questão ainda está em desenvolvimento pelos alunos do curso de Sistema de Informação da UFU, porém grande parte do *backend* já está pronto, o que permite a realização dos testes que serão apresentados.

3.2 Ferramentas

Conforme mencionado anteriormente, este trabalho será dedicado apenas no teste *backend* e, com isso, é necessário gerar a aplicação. Nesse sentido, as ferramentas utilizadas foram o Node.js, responsável pela execução do ambiente do lado do servidor, que usa o JavaScript como linguagem, o PostgreSQL e o PgAdmin para as funcionalidades do banco

de dados, o Yarn e o NPM que são gerenciadores de pacotes, utilizados para auxiliar e facilitar as configurações iniciais, que foram todas feitas por alguns comandos, tais como:

- Comando para configurar as variáveis de ambiente:

```
1 npm install -g win-node-env
```

- Comando para baixar todas as dependências necessárias para o projeto:

```
1 yarn
```

- Comando para criar toda a estrutura do banco de dados:

```
1 yarn typeorm migration:run
```

- Comando para criar um usuário de administrador na tabela “users”:

```
1 yarn seed:admin
```

- Comando para rodar o servidor e deixá-lo pronto para usar:

```
1 yarn dev
```

3.3 Jest

Inicialmente criado pelo Facebook, o Jest é um framework de testes de código aberto que trabalha com JavaScript e TypeScript. Projetado para garantir a correção de qualquer código em JavaScript com uma simplicidade tanto na sua configuração quanto no desenvolvimento, o Jest passou a ser bastante conhecido para criação dos testes automatizados. Com isso, hoje ele é utilizado em grandes empresas, como o Twitter, Spotify, Instagram, entre outros. Além disso, o Jest conta com uma boa documentação encontrado facilmente na internet ou em seu site oficial (FACEBOOK, 2022).

4 Resultados

Foi desenvolvido uma automação de testes no intuito de validar algumas funcionalidades relacionadas a criar uma Turma no sistema em questão. Em cada repositório, é descrito todos os casos de teste e os códigos que serão executados, que são responsáveis pelas validações.

A forma em que são feitos os testes automatizados utilizando o Jest, primeiro é criado uma função com a descrição do contexto dos testes, que está no comando *describe*. Posteriormente, dentro dele será a codificação de todos os casos de teste desse cenário, feito utilizando o comando *it*, na qual são feitas todas as validações necessárias. Além disso, é possível utilizar outras funções, como o *beforeAll*, que é executado antes de todos os casos de testes.

Na Figura 2 conseguimos verificar o *describe*, que engloba os métodos de teste, e o *beforeAll*, que nesse caso foi utilizado para criar a conexão com o banco de dados antes da execução dos casos de teste. É importante ressaltar que por se tratar de testes, foi usado um banco de dados de teste criado conforme o modelo descrito nas *migrations*, com a estrutura semelhante ao da aplicação, para não haver nenhuma interação com o banco de dados real. Isso garante que não tenham registros já cadastrados anteriormente que possam interferir nos testes e que os registros feitos pelos casos de testes não poluem o banco de dados da aplicação. Além disso, foi preciso criar um usuário administrador para que a sessão fosse autenticada, para possibilitar a conexão.

```
let connection: Connection;

jest.setTimeout(600 * 60);

describe("Testar as funcionalidades do cadastro de uma Turma", () => {
  beforeAll(async () => {
    connection = await createConnection();
    await connection.runMigrations();

    const id = uuidV4();
    const password = await hash("admin", 8);

    await connection.query(
      `INSERT INTO usuarios(id, name, email, password, "isAdmin", created_at)
      values('${id}', 'admin', 'sodd_tcc@outlook.com', '${password}', 'true', 'now()')`
    );

    const curso = new Curso();
    curso.codigo = "BCC";
    curso.nome = "Ciencia da Computacao";
    curso.unidade = "UFU";
    curso.campus = "udi";
    curso.permitir_choque_periodo = false;
    curso.permitir_choque_horario = false;

    await connection.manager.save(curso);

    const disciplina = new Disciplina();
    disciplina.codigo = "GSI505";
    disciplina.nome = "Lógica para Computação";
    disciplina.ch_teorica = 4;
    disciplina.ch_pratica = 0;
    disciplina.ch_total = 4;
    disciplina.curso = "BCC";
    disciplina.temfila = true;
    disciplina.periodo = 2;

    const disciplina2 = new Disciplina();
    disciplina2.codigo = "GSI508";
    disciplina2.nome = "Estrutura de dados 1";
    disciplina2.ch_teorica = 4;
    disciplina2.ch_pratica = 0;
    disciplina2.ch_total = 4;
    disciplina2.curso = "BCC";
    disciplina2.temfila = true;
    disciplina2.periodo = 2;

    await connection.manager.save(disciplina);
    await connection.manager.save(disciplina2);
  });
```

Figura 2 – Exemplo do comando “Describe” utilizado para iniciar o desenvolvimento dos testes.

Para cadastrar uma Turma, é necessário que haja um registro de uma Disciplina,

na qual seu campo “código” é utilizado como chave estrangeira. Para que seja criado uma Disciplina, é necessário que já exista um Curso cadastrado, pelo mesmo motivo. Nesse sentido, foi criado essa massa de teste, que consiste em criar os registros que são pré-requisito para realizar os casos de teste que virão (Figura 2).

Na sequência, foram desenvolvidos os casos de teste pelo método *it*. Por segurança, em todos os testes é necessário criar uma sessão com o usuário inserido anteriormente no banco de dados e utilizar o *token* para permitir o acesso. Dessa forma, é possível tentar criar o registro desejado do teste em questão. No exemplo feito na Figura 3, foi criada uma Turma e validado o seu sucesso a partir da utilização dos comandos:

```
1 expect(response.status).toBe(201);
```

Valida se a requisição teve como retorno o código HTTP 201, que significa sucesso no processo.

```
1 expect(responseResult.codigo_disc).toBe("GSI505");
```

Valida se o registro foi cadastrado com o campo “`codigo_disc`” igual a “GSI505”, o mesmo inserido na criação.

```
1 expect(responseResult.turma).toBe("M");
```

Valida se o registro foi cadastrado com o campo “turma” igual a “M”, o mesmo inserido na criação.

```
1 expect(responseResult.ch).toBe(6);
```

Valida se o registro foi cadastrado com o campo “ch” igual a “6”, o mesmo inserido na criação.

```
1 expect(responseResult.ano).toBe(2022);
```

Valida se o registro foi cadastrado com o campo “ano” igual a “2022”, o mesmo inserido na criação.

```
1 expect(responseResult.semestre).toBe(1);
```

Valida se o registro foi cadastrado com o campo “semestre” igual a “1”, o mesmo inserido na criação.

```
it("Criar uma turma.", async () => {
    const responseToken = await request(app).post("/sessions").send({
      email: "sodd_tcc@outlook.com",
      password: "admin",
    });

    const { token } = responseToken.body;

    const response = await request(app)
      .post("/turmas")
      .send({
        codigo_disc: "GSI505",
        turma: "M",
        ch: 6,
        ano: 2022,
        semestre: 1,
      })
      .set({
        Authorization: `Bearer ${token}`,
      });

    const responseResult = response.body as Turma;

    expect(response.status).toBe(201);
    expect(responseResult.codigo_disc).toBe("GSI505");
    expect(responseResult.turma).toBe("M");
    expect(responseResult.ch).toBe(6);
    expect(responseResult.ano).toBe(2022);
    expect(responseResult.semestre).toBe(1);
  });
```

Figura 3 – Código do caso de teste “Criar uma turma”.

Na sequência, será apresentado outros exemplos da automação desenvolvida com uma breve explicação e, por fim, o resultado das execuções:

Tentar criar uma Turma com os campos “`codigo_disc`” e “turma” com duplicidade (Figura 4). Como a Turma possui chaves compostas, que são esses dois campos, o sistema não deve cadastrar o registro, já que eles já foram utilizados no primeiro teste (Figura 3), que foi cadastrado com sucesso. Portanto, não é esperado um retorno de sucesso no comando *expect*.

```
it(
"Tentar criar uma turma com os campos codigo_disc e turma com duplicidade.
O sistema deve retornar erro."
, async () => {
    const responseToken = await request(app).post("/sessions").send({
      email: "sodd_tcc@outlook.com",
      password: "admin",
    });

    const { token } = responseToken.body;

    const response = await request(app)
      .post("/turmas")
      .send({
        codigo_disc: "GSI505",
        turma: "M",
        ch: 6,
        ano: 2022,
        semestre: 1,
      })
      .set({
        Authorization: `Bearer ${token}`,
      });

    expect(response.status).not.toBe(201);
  });
```

Figura 4 – Código do caso de teste que valida a funcionalidade de criar uma turma com duplicidade.

Tentar criar uma Turma com o campo “ch” vazio. Por se tratar de um campo obrigatório, o sistema não deve cadastrar o registro com o devido campo vazio. Sendo assim, não é esperado um retorno de sucesso no comando *expect* (Figura 5).

```
it(
"Tentar criar uma turma com o campo ch vazio. O sistema deve retornar
erro."
, async () => {
    const responseToken = await request(app).post("/sessions").send({
      email: "sodd_tcc@outlook.com",
      password: "admin",
    });

    const { token } = responseToken.body;

    const response = await request(app)
      .post("/turmas")
      .send({
        codigo_disc: "GSI505",
        turma: "A",
        ch: null,
        ano: 2022,
        semestre: 1,
      })
      .set({
        Authorization: `Bearer ${token}`,
      });

    expect(response.status).not.toBe(201);
  });
```

Figura 5 – Código do caso de teste que valida a funcionalidade de criar uma turma com o campo "ch" vazio.

Tentar criar uma Turma com o campo "`codigo_disc`" de uma Disciplina inexistente. O sistema não deveria cadastrar o registro, já que esse campo é uma chave estrangeira. Conforme mostrado na Figura 6, o campo em questão foi preenchido por "ABC", que não é o código de nenhuma Disciplina cadastrada na massa de teste. Com isso, o esperado no comando *expect* é o de não receber o status de sucesso.

```
it(
"Tentar criar uma turma com o campo codigo_disc de uma disciplina inexistente.
O sistema deve retornar erro."
, async () => {
    const responseToken = await request(app).post("/sessions").send({
      email: "sodd_tcc@outlook.com",
      password: "admin",
    });

    const { token } = responseToken.body;

    const response = await request(app)
      .post("/turmas")
      .send({
        codigo_disc: "ABC",
        turma: "A",
        ch: 4,
        ano: 2022,
        semestre: 1,
      })
      .set({
        Authorization: `Bearer ${token}`,
      });

    expect(response.status).not.toBe(201);
  });
```

Figura 6 – Código do caso de teste que valida a funcionalidade de criar uma turma com o campo `"codigo_disc"` de uma disciplina inexistente.

Tentar criar uma Turma com o campo “semestre” maior do que dois (Figura 7). Como em um ano só existem no máximo dois semestres, o sistema não deveria aceitar o registro nesse teste. Com isso, não é esperado um retorno de sucesso no comando *expect*.

```
it(
"Tentar criar uma turma com o campo semestre maior que 2. O sistema deve
 retornar erro."
, async () => {
    const responseToken = await request(app).post("/sessions").send({
      email: "sodd_tcc@outlook.com",
      password: "admin",
    });

    const { token } = responseToken.body;

    const response = await request(app)
      .post("/turmas")
      .send({
        codigo_disc: "GSI505",
        turma: "A",
        ch: 4,
        ano: 2022,
        semestre: 3,
      })
      .set({
        Authorization: `Bearer ${token}`,
      });

    expect(response.status).not.toBe(201);
  });
```

Figura 7 – Código do caso de teste que valida a funcionalidade de criar uma turma com o campo “semestre” maior que 2.

Ao executar o teste apresentado pela Figura 7, o Jest acusou um erro, na qual o *expect* espera algo diferente do código HTTP de sucesso, que não é o que acontece, conforme mostrado na Figura 8. Portanto, podemos concluir que o registro foi cadastrado e que foi encontrado uma falha, mostrando que o software não está preparado para esse tipo de bloqueio.

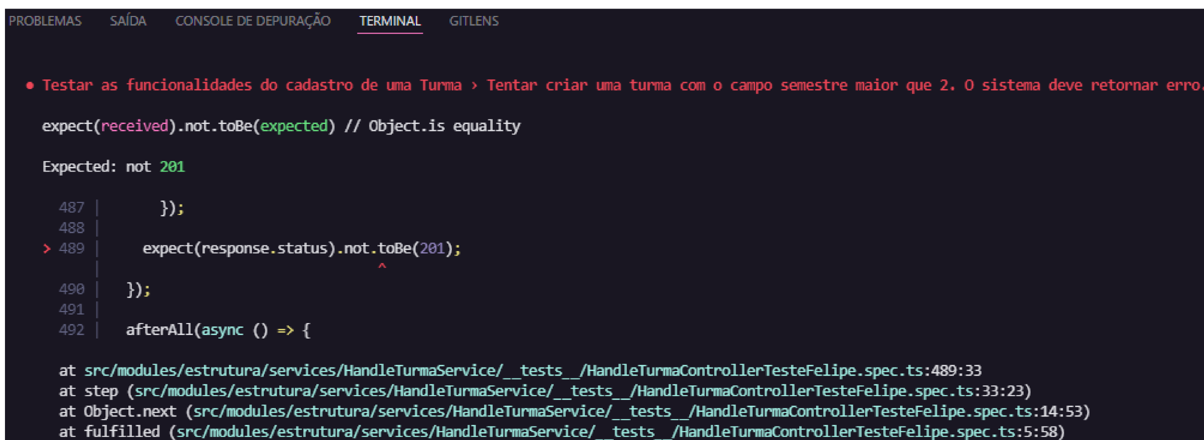

Figura 8 – Mensagem que aparece no terminal quando a automação encontra algum erro.

No total, foram feitos 17 testes para validar a funcionalidade de criar uma Turma, sendo que 5 deles acharam algum tipo de erro e 12 deles passaram com sucesso (Figura 9 e 10). Para rodar todos os testes, foi executado no terminal o comando abaixo, também presente na Figura 10:

```
1 NODE_ENV=test yarn jest HandleTurmaControllerTesteFelipe.spec.ts --detectOpenHandles
```

O resultado deles e dos outros testes descritos acima, é exibido no terminal após eles serem executados (Figura 9 e 10). Em caso de falha, a forma em que é apresentado está ilustrado na Figura 8.

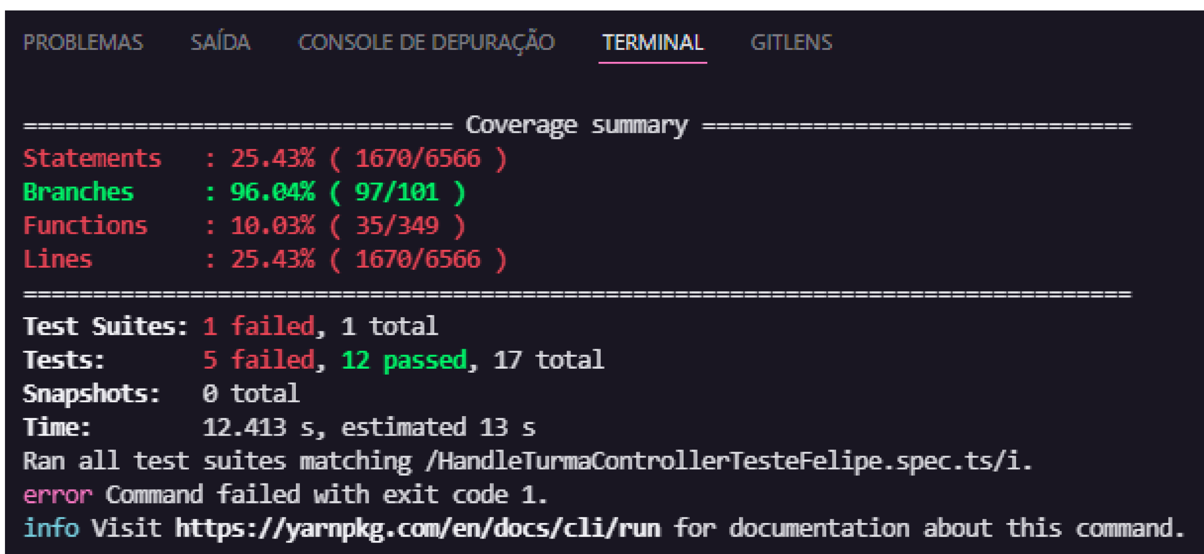

Figura 9 – Resumo consolidado de todos os casos de testes apresentados no terminal após a execução.

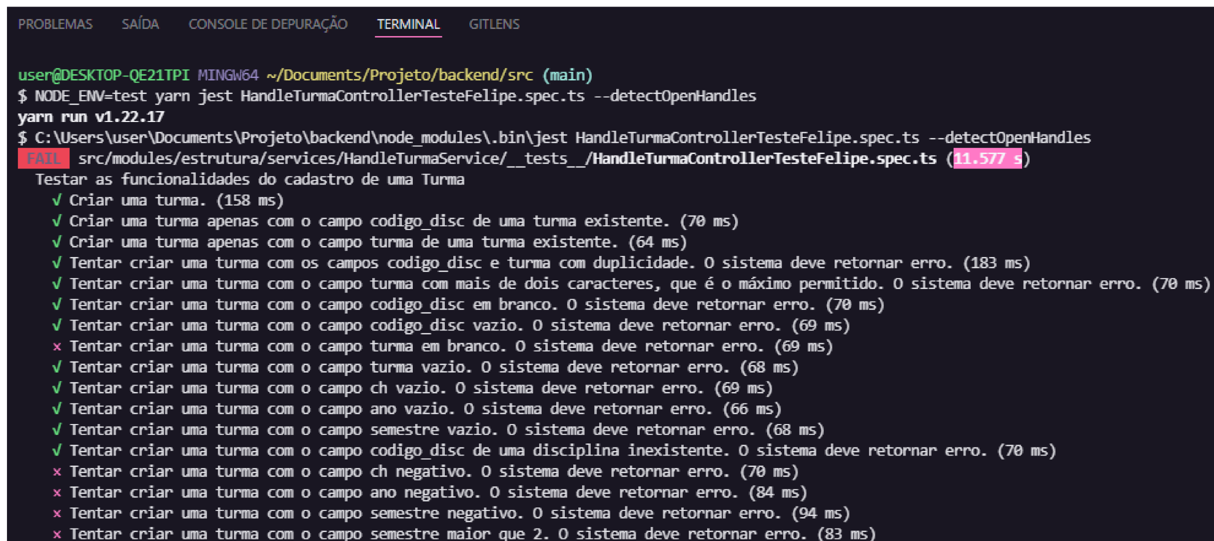

Figura 10 – Resultado individual de sucesso ou falha de cada caso de teste apresentado no terminal após a execução.

Por fim, depois da codificação de todos os testes, foi utilizado o *afterAll* para apagar o banco de dados e fechar a conexão (Figura 11). Por mais que este comando esteja no final do código, ele é sempre executado depois de todos os outros.

Figura 11 – Método “afterAll”.

5 Conclusão

Conforme foi apresentado neste trabalho, os testes de software estão presentes em todo processo de desenvolvimento de sistemas, tendo em vista que ele tem o propósito de melhorar a qualidade do produto final. Ademais, foi exemplificado como são feitos os testes *backend* automatizados utilizando o Jest, realizados no Sistema Online de Distribuição de Disciplinas (SODD).

Além de mostrar a importância de se realizar testes em um sistema, principalmente de forma organizada e estruturada, foi possível identificar por meio dos testes automatizados realizados que o software em questão (SODD) demonstrou que algumas funcionalidades estavam de acordo com o esperado e outras não. Com isso, podemos afirmar que os testes apresentados atingiram seu objetivo, já que eles garantiram o funcionamento de certos requisitos e também identificou problemas para serem corrigidos antes de chegar ao usuário final.

Sendo assim, esse trabalho contribuiu no desenvolvimento do projeto da nova implementação realizada no SODD. Além disso, ficou claro que é possível aprofundar nos testes e encontrar vários outros casos de testes em uma funcionalidade. Com isso, ainda existem vários outros a serem criados, que podem ser feito por outros alunos da UFU que queiram cooperar no projeto.

Referências

BARBOSA, F. B.; TORRES, I. V. O teste de software no mercado de trabalho. *Tecnologias em Projeção*, v. 02, p. 49–52, 2011. Disponível em: <http://revista.faculdadeprojecao.edu.br/index.php/Projecao4/article/view/82/70>. Citado na página 6.

BARTIÉ, A. *Garantia da qualidade de software*. Gulf Professional Publishing, 2002. ISBN 9788535211245. Disponível em: <https://books.google.com.br/books/about/Garantia_da_qualidade_de_software.html?hl=pt-BR&id=O57Us2kUh4oC&redir_esc=y>. Citado na página 9.

COLLINS, E. F.; LOBÃO, L. M. de A. Experiência em automação do processo de testes em ambiente Ágil com scrum e ferramentas opensource. *Anais do IX Simpósio Brasileiro de Qualidade de Software*, p. 303–310, 2010. Disponível em: <https://sol.sbc.org.br/index.php/sbqs/article/view/15438/15281>. Citado na página 6.

DIAS NETO, A. C. Introdução a teste de software. *Engenharia de Software Magazine*, v. 01, p. 54–59, 2014. Disponível em: <https://www.researchgate.net/publication/266356473_Introducao_a_Teste_de_Software>. Citado na página 8.

FACEBOOK. *Facebook - Documentação Jest*. 2022. <https://jestjs.io/pt-BR/>. [Online; Acessado em 25 de Março de 2022]. Citado na página 13.

FERNANDES, G. S. *Automação de testes com Selenium WebDriver aplicada no Sistema de Distribuição de Disciplinas*. 2020. Trabalho de Conclusão de Curso (Graduação em Gestão da Informação – Universidade Federal de Uberlândia, Uberlândia). Citado na página 12.

LOCATELLI, J. A. *Sistema Online para Distribuição de Disciplinas*. 2015. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação – Universidade Federal de Uberlândia, Uberlândia). Citado na página 12.

LUFT, C. C. *Teste de software: uma necessidade das empresas*. 2012. <https://bibliodigital.unijui.edu.br:8443/xmlui/bitstream/handle/123456789/1307/TCC%20-%20Cristiane%20Carline%20Luft.pdf?sequence=1&isAllowed=y>. [Online; Acessado em 17 de Fevereiro de 2022]. Citado 2 vezes nas páginas 8 e 9.

MENDES, M. dos S. *Proposta de Nova Implementação do Sistema Online de Distribuição de Disciplinas*. 2022. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação – Universidade Federal de Uberlândia, Uberlândia). Citado na página 12.

TREVISAN JUNIOR, E. *Processos de teste de software*. 2007. <http://lyceumonline.usf.edu.br/salavirtual/documentos/1081.pdf>. [Online; Acessado em 17 de Fevereiro de 2022]. Citado 3 vezes nas páginas 4, 9 e 10.