Implementação de Testes Unitários para o Sistema EducAPI: Um Relato de Experiência

Enos F. T. da Silva[1]

Universidade Federal da Paraíba (UFPB)
Caixa Postal 58297 – 000 – Rio Tinto – PB – Brasil

enos.francisco@dcx.ufpb.br

Abstract. *The implementation of unit tests can be a complex task, especially when the classes to be tested involve frameworks. Moreover, another challenge is to build tests that are easy to maintain. In this context, the problem of implementing unit tests with quality in systems based on the Spring framework arises. Considering this problem, this article aims to present an experience report on the implementation of unit tests in the EducAPI System, which uses this framework. In order to do this, we describe the test implementation process, the difficulties encountered and the lessons learned in this process.*

Resumo. *A implementação de testes unitários pode ser uma tarefa complexa, principalmente quando as classes a serem testadas envolvem frameworks. Além disso, um outro desafio é construir testes que sejam fáceis de manter. Nesse contexto surge a problemática de implementar testes unitários com qualidade em sistemas baseados no framework Spring. Diante dessa problemática, este artigo visa apresentar um relato de experiência sobre a implementação de testes unitários no Sistema EducAPI, o qual utiliza esse framework. Para isto, foi descrito o processo de implementação de testes, as dificuldades encontradas e as lições aprendidas nesse processo.*

1. Introdução

Com o avanço da tecnologia, os sistemas precisam ser produzidos e mantidos de forma rápida, eficaz e com qualidade. Um dos métodos para assegurar a qualidade é o uso de testes. Embora existam diversos métodos de se testar sistemas, em determinadas situações, como por exemplo, ao criar testes unitários para sistemas baseados em frameworks como o Spring, os testes podem ficar difíceis de manter. Nesse contexto, o presente trabalho busca atacar a problemática de implementar testes unitários de qualidade em sistemas baseados no *framework Spring*.

O objetivo deste trabalho é apresentar um relato de experiência sobre o processo de construção de testes unitários no sistema EducAPI[2], um sistema construído utilizando o framework Spring.

---

[1] Trabalho de conclusão de curso, sob orientação da professora Ayla Débora Dantas de Souza Rebouças submetido ao Curso de Licenciatura em Ciência da Computação do Centro de Ciências Aplicadas e Educação (CCAE) da Universidade Federal da Paraíba, como parte dos requisitos necessários para obtenção do grau de LICENCIADO EM CIÊNCIA DA COMPUTAÇÃO.

[2] "Sistema EducAPI." https://github.com/a4s-ufpb/EducAPI. Acessado em 29 jun. 2021.

Para atingir o objetivo deste trabalho, as atividades iniciais consistiram em três etapas. A primeira teve como finalidade a compreensão do sistema EducAPI e dos principais conceitos relativos ao framework Spring e à área de testes automáticos por meio da análise de artigos e documentações de ferramentas. A segunda etapa teve um caráter exploratório e nela buscamos atentar para as diferentes formas de realizar a automação de testes unitários na linguagem Java e em sistemas baseados em Spring com base no projeto EducAPI. Na terceira, deu-se início ao processo de construção dos testes de forma iterativa e incremental, buscando desenvolvê-los utilizando boas práticas e padrões de forma que fossem fáceis de manter. Posteriormente passou-se à documentação do processo e das principais lições aprendidas neste documento de TCC.

As demais seções deste trabalho foram organizadas conforme descrito a seguir: A Seção 2 possui uma breve descrição dos conceitos e ferramentas utilizadas ao longo do artigo. A Seção 3 contém a descrição do processo de implementação dos testes. A Seção 4 descreve os desafios encontrados ao longo da implementação dos testes do EducAPI, bem como as principais lições aprendidas. Por fim, a Seção 5 apresenta as considerações finais e propostas de trabalhos futuros.

2. Fundamentação Teórica

Nesta seção apresentamos uma breve descrição dos conceitos e ferramentas utilizadas ao longo do artigo de forma a facilitar a compreensão do trabalho realizado. Inicialmente, apresentaremos o sistema testado, que foi o EducAPI e explicaremos alguns detalhes sobre o Framework Spring utilizando esse sistema como exemplo. A seguir, falaremos sobre automação de testes, incluindo informações sobre testes unitários, o framework de testes JUnit e o uso de *Mocks*. Por fim, apresentaremos o conceito de anotações e alguns exemplos de anotações presentes no EducAPI e nos testes desenvolvidos.

2.1 O Sistema EducAPI e o *Framework* Spring

A seguir apresentamos o sistema para o qual foram desenvolvidos os testes apresentados neste trabalho, e que é intitulado Sistema EducAPI. Descrevemos nesta subseção seus objetivos, seus módulos e o framework *Spring*, que serviu como base para sua construção.

O EducAPI é um sistema desenvolvido como parte das atividades do projeto Apps4Society[3] e como uma evolução do trabalho de Silva, Alves e Rebouças (2017, p. 418). O seu objetivo é gerenciar uma base de dados multimídia que pode ser utilizada por diferentes aplicativos ou sistemas Web, com foco em alfabetização, como por exemplo, aplicativos para associar imagens a palavras ou que explorem atividades para completar sílabas ou letras de palavras.

[3] "Apps4Society." https://apps4society.dcx.ufpb.br/. Acessado em 21 abr.. 2021.

Esta base de dados oferecida pelo EducAPI contém desafios representados por palavras e que estão atrelados a contextos (temas). Um exemplo de contexto poderia ser “ANIMAIS SELVAGENS” e exemplos de desafios relacionados a este contexto poderiam ser “LEÃO”, “TIGRE” e “ONÇA”. Tanto os desafios quanto os contextos podem ter associados a eles áudios, imagens ou vídeos.

Os usuários cadastrados no sistema (e.g. professores) podem cadastrar, editar, deletar ou consultar desafios e contextos. Esses contextos e desafios cadastrados poderão ser acessados tanto pelo usuário que os cadastrou, quanto por outros usuários.

A Figura 1 é uma representação da abordagem proposta para a utilização do sistema. Nela se ilustram professores utilizando uma interface de software para gerir os contextos e que se comunica com o serviço provido pelo EducAPI, que é responsável por disponibilizar e armazenar as informações relativas a contextos e desafios cadastrados por diferentes usuários. A Figura também ilustra que usuários de diferentes perfis (e.g. crianças ou adultos) podem acessar o sistema através de softwares que utilizam os serviços do EducAPI para as mais diversas finalidades, como por exemplo, atividades de associação de imagens a palavras, jogos da forca, jogos de memorização, dentre outros.

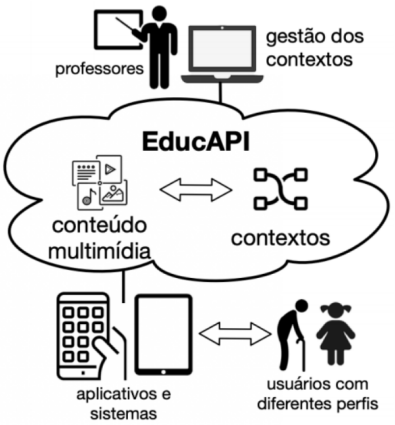

Figura 1: Abordagem Proposta com o EducAPI

Fonte: https://apps4society.dcx.ufpb.br/?page_id=942

O EducAPI é composto por três módulos, sendo eles, o módulo de Usuário (*User*), o de Contexto (*Context*), e o de Desafio (*Challenge*). Neste trabalho de conclusão de curso focamos no módulo de Usuário (*User*), que é o módulo responsável pelo controle de acesso e gerência de usuários.

O sistema, desenvolvido em Java, fornece uma API REST (*Application Programming Interface* que segue o padrão de arquitetura REST) para permitir o acesso dos usuários aos seus serviços. Para construir esse sistema, foi utilizado o framework Spring, que será explicado a seguir.

O Spring é um *framework* que fornece um modelo abrangente de programação e configuração para sistemas baseados em Java [Spring Framework, 2021]. Alguns dos

projetos e funcionalidades relacionadas ao Spring são o Spring Boot, Spring Web MVC, Spring Data, Spring Security, dentre outros. O serviço Web disponibilizado via API REST no EducAPI utilizou o Framework Spring para sua construção. Na abordagem proposta pelo Spring, as solicitações HTTP (do inglês HyperText Transfer Protocol, na tradução livre: protocolo de transferência de hipertexto) que a API REST recebe são tratadas por um *resource controller* (controlador de recursos), que é baseado no elemento *Controller* do padrão arquitetural MVC [Fowler et al. 2002]. Nesse padrão são propostos três elementos: o Model, o View e o Controller. A seguir serão explicados esses elementos na implementação do EducAPI utilizando a linguagem Java e o *Spring*.

O *Model* é uma representação do domínio da aplicação e é responsável pelas regras de negócios. No EducAPI, um *model* apresenta basicamente duas classes. Uma das classes é a classe `User`, onde são definidas as características que um usuário pode ter, como nome, e-mail, senha dentre outras. A outra classe é a classe `UserService`, onde são definidas as regras de negócios referentes à classe `User`.

O *View* corresponde à interface com o usuário, e é responsável por receber e apresentar informações. No EducAPI essa responsabilidade é destinada aos diferentes serviços que se comunicam com a API REST por meio de diferentes dispositivos, como por exemplo, um aplicativo para que um professor se cadastre, cadastre contextos e desafios. Um exemplo de uma dessas interfaces é o aplicativo EducAPI Manager[4], o qual corresponderia na Figura 1 à gestão de contextos.

O *Controller* é o elemento que recebe as informações inseridas no *View*, manipula o *Model* e retorna as informações atualizadas ao *View*. No EducAPI, um exemplo de classe que tem essa função durante a gestão de usuários é a classe `UserResource`.

A Figura 2 ilustra alguns dos elementos do MVC com foco na gestão de usuários do EducAPI, que foi o módulo considerado para os testes apresentados neste trabalho.

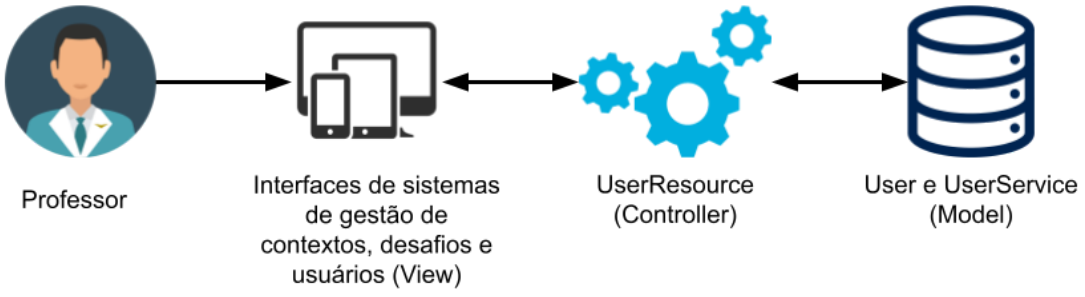

Figura 2: Representação do MVC no EducAPI no módulo de Usuário

Fonte: O autor

[4] "EducAPI-Manager - GitHub." 26 mai.. 2021, https://github.com/a4s-ufpb/EducAPI-Manager. Acessado em 16 jun.. 2021.

Conforme ilustrado pela Figura 2, elementos como a classe `UserResource`, que atua como *Controller* gerenciam requisições HTTP baseadas na arquitetura REST (do inglês *Representational State Transfer*, na tradução livre: Transferência de Estado Representacional). Esta classe, por exemplo, recebe e responde as requisições (*requests*) HTTP relativas ao cadastro, atualização, consultas e remoção de dados de usuários do sistema. Os dados recebidos das requisições são mapeados em objetos Java por meio de anotações do Spring como a anotação `@RequestController` presente nesta classe.

O código do serviço REST do EducAPI[5] pode ser acessado através do repositório do projeto Apps4Society no GitHub.

2.2 Automação de Testes, JUnit e *Mocks*

Nesta seção apresentamos brevemente o que é a automação de testes, a diferença entre o teste automatizado e teste manual, o objetivo do teste automatizado e seus benefícios, além de ferramentas e conceitos relacionados à automação de testes e criação de testes de unidade, que é o foco deste trabalho.

A automação de teste consiste no uso de ferramentas de execução de testes de modo a automatizar o processo de execução de testes. Tem como objetivo aumentar a confiança no sistema, reduzir o trabalho manual repetitivo, como por exemplo, a execução de testes de regressão e configuração de ambiente, além de oferecer maior consistência com relação a testes manuais e aumentar a confiança dos desenvolvedores para possíveis alterações e melhorias no código [ISTQB 2018].

Diferente do teste manual, em que uma pessoa responsável por garantia de qualidade executa os testes manualmente interagindo com o software, o teste que foi automatizado pode ser executado rapidamente e é menos propício a erros humanos. Logo, um bom conjunto de testes automatizados gera uma maior confiança para implementar novas funcionalidades no sistema. Além disso, a presença de testes automáticos também é fundamental para processos de refatoração de código, que segundo Fowler e Beck (2018) consistem em modificar um sistema de software de modo que não seja alterado o comportamento externo do código, embora melhore sua estrutura interna.

Embora existam diferentes níveis de testes, como por exemplo unidade, serviço e UI (*user interface*), neste trabalho focamos na automação de testes unitários pois segundo Mike Cohn (2009), os testes unitários são mais rápidos de serem executados e testam individualmente as partes do sistema. Sendo assim, uma vez que falham, permitem que se encontre mais rapidamente a causa do problema.

O teste unitário, conhecido também como teste de componente ou módulo, concentra-se em componentes que são testáveis separadamente. Por meio dele é possível reduzir o risco de defeitos, verificar se o componente funciona conforme o esperado e gerar uma maior confiança no sistema [ISTQB 2018].

Por isolar um componente do sistema, é geralmente executado de forma automatizada e possibilita o isolamento de um defeito. Dessa forma, o desenvolvedor pode ter um melhor entendimento do defeito e de sua solução, de modo que uma vez

[5] "EducAPI - GitHub." https://github.com/a4s-ufpb/EducAPI. Acessado em 28 jun.. 2021.

que seja desenvolvida uma solução para um defeito, o teste pode orientar se o problema foi resolvido.

Para poder testar isoladamente cada componente, em alguns casos na programação orientada a objetos, faz-se necessária a utilização de *mocks,* também conhecidos como *mock objects*[6]. *Mock* é o termo utilizado no desenvolvimento para nos referirmos a objetos que simulam o comportamento de objetos reais de forma controlada.

Por exemplo, ao implementar um teste unitário onde se invoca um método de um componente do sistema que está sendo testado, utilizamos *mocks* para isolar este componente de componentes externos.

Para implementar testes automáticos na linguagem Java, utilizamos comumente o framework de testes JUnit [Tudose 2020], um framework iniciado por Kent Beck e Erich Gamma em 1995. Por meio do JUnit, é possível efetuar verificações (*assertions*) sobre o resultado de operações realizadas em um sistema sob teste.

Para criar *mocks* em testes JUnit é comum a utilização do *framework* Mockito. Um outro *framework* também utilizado é o MockMvc.

O Mockito é um framework que permite a interceptação de chamadas a métodos substituindo a execução desses métodos e retornando dados configurados nos testes [Mockito Framework]. Por exemplo, pode-se configurar um objeto *mock* para retornar determinado valor quando um método específico for invocado neste objeto.

Nos testes apresentados neste trabalho, o Mockito foi utilizado para isolar as classes testadas de componentes externos com os quais estas classes interagiam.

Utilizamos também o Spring MVC *Test framework*[7], também conhecido como MockMvc, que faz parte do *framework* Spring Test. Ele fornece suporte para testar aplicativos Spring MVC, por meio da simulação de requisições (*requests*) HTTP.

Para permitir a boa qualidade dos testes, é importante observar padrões documentados na área, como os presentes no livro *Growing Object-Oriented Software, Guided by Tests,* de Freeman e Pryce (2009). Um destes padrões é o padrão *Test Data Builder*.

O padrão *Test Data Builder* é uma adaptação do padrão *Builder*, utilizado para simplificar a criação de objetos complexos. Por exemplo, se um teste precisar criar vários objetos, pode-se usar esse padrão para criar os objetos necessários, conforme ilustra a Listagem 1. Nela são criados dois objetos do tipo `User` semelhantes, mas apenas com um atributo que os diferencia, que é o e-mail.

```
UserBuilder fooBar = new UserBuilder().anUser.withName("Foo Bar");
```

[6] "Mock object - Wikipedia." https://en.wikipedia.org/wiki/Mock_object. Acessado em 22 jun.. 2021.

[7] "Unit testing Spring MVC." 9 jun.. 2021, https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#spring-mvc-test-framework. Acessado em 23 jun.. 2021.

```
User fooBarWithInvalidEmail = fooBar.withEmail("invalid
email").build();

User fooBarWithValidEmail =
fooBar.withEmail("valid@email.com").build();
```

Listagem 1. Exemplo da criação de dois objetos semelhantes utilizando o padrão Test Data Builder (Código em Java)

Fonte: O Autor

Por meio deste padrão é possível experimentar nos testes muitas variações de objetos para explorar diferentes cenários.

2.3. Anotações

Nesta seção, apresentamos de forma breve o que é uma anotação no contexto do desenvolvimento de software e quais foram as anotações utilizadas ao longo do processo de implementação dos testes no EducAPI.

Anotações, do inglês *annotation*, é uma forma de metadado sintático utilizada para os mais diversos fins na programação. Esses metadados sintáticos foram fundamentais para a implementação dos testes, e nesta seção iremos descrever as principais anotações utilizadas na construção dos testes desenvolvidos para este trabalho.

A *annotation* `@ExtendWith`[8], fornecida pelo JUnit 5 tem como objetivo registrar extensões declarativamente, de modo que quando determinado evento ocorra durante a execução dos testes, o JUnit chame a classe extendida. Neste projeto foi utilizada para estender as funcionalidades do *Mockito*, possibilitando utilizar o *framework Mockito* para implementar os *Mocks* necessários para o desenvolvimento dos testes.

O `@Mock`[9] é uma anotação que foi utilizada para simular uma classe e o `@InjectMocks`[10] foi utilizado para injetar os *mocks* no objeto de testes automaticamente. O `@BeforeEach`[11] foi usado para definir um conjunto de métodos a ser executado antes de cada teste, enquanto o `@Test`[12] foi utilizado para declarar que um método em específico é um teste e que precisa ser executado pelo JUnit.

---

[8] "ExtendWith (JUnit 5.0.3 API)." https://junit.org/junit5/docs/5.0.3/api/org/junit/jupiter/api/extension/ExtendWith.html. Acessado em 30 jun.. 2021.

[9] "Mockito 3.11.2 API - javadoc.io." https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#mock_annotation. Acessado em 30 jun.. 2021.

[10] "Mockito 3.11.2 API - javadoc.io." https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html. Acessado em 30 jun.. 2021.

[11] "BeforeEach (JUnit 5.0.2 API)." https://junit.org/junit5/docs/5.0.2/api/org/junit/jupiter/api/BeforeEach.html. Acessado em 30 jun.. 2021.

[12] "Test (JUnit 5.0.2 API)." https://junit.org/junit5/docs/5.0.2/api/org/junit/jupiter/api/Test.html. Acessado em 30 jun.. 2021.

Também foi utilizada neste trabalho a anotação `@WebMvcTest`[13] fornecida pelo Spring, para configurar questões de segurança em alguns testes referentes a camada *controller*. Foi utilizada a anotação `@Autowired`[14] para injetar automaticamente as dependências necessárias e o `@MockBean`[15], que possui uma função semelhante ao `@Mock` e `@InjectMocks`. Contudo, o `@MockBean` é utilizado em testes de integração por simular objetos no contexto de uma aplicação Spring. A principal diferença entre o `@InjectMocks` em conjunto com o `@Mocks` para o `@MockBean` é que para os dois primeiros, não é necessário subir a aplicação Spring. Em contrapartida, o `@MockBean`, atua em uma aplicação Spring.

3. Processo de Implementação dos Testes

Nesta seção apresentamos o processo para entender o EducAPI, para definir as ferramentas de execução de testes e para implementar os testes unitários.

3.1 Entendendo o EducAPI

Para definir qual o módulo alvo para a implementação dos testes, foi necessário entender qual problema o EducAPI resolve e principalmente como ele resolve, bem como sua estrutura interna. Para isso, inicialmente foi preciso dialogar com alguns dos contribuidores do projeto Apps4Society e ler a descrição publicada no site do mesmo. Em seguida, buscamos compreender como é organizada sua estrutura interna, o que fica em cada pasta e quais suas responsabilidades, bem como a arquitetura utilizada. Por exemplo, foi necessário entender mais sobre o padrão arquitetural MVC (*Model, View, Controller*) e sobre os elementos do EducAPI. Os diálogos, pesquisas e a análise de sua estrutura interna resultaram na escrita da Seção 2.1 deste trabalho, em que explicamos um pouco sobre o EducAPI e seus módulos.

Após isso, decidimos focar no módulo de usuários por acreditarmos ser um módulo importante do sistema. Mesmo assim, apresentamos na Figura 3 as principais classes correspondentes ao *model* de cada um dos 3 módulos do EducAPI (Módulo de Usuários, Módulo de Contextos e Módulo de Desafios). Consideramos o módulo de Usuários importante pois caso ocorra algum problema nesse módulo, o sistema pode ficar comprometido, impedindo os usuários de cadastrarem novos contextos e/ou desafios ou até mesmo bloqueando o acesso dos usuários à API.

---

[13] "WebMvcTest (Spring Boot 2.5.2 API)." https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html. Acessado em 30 jun.. 2021.

[14] "Autowired (Spring Framework 5.3.8 API)." https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html. Acessado em 30 jun.. 2021.

[15] "MockBean (Spring Boot 2.5.2 API)." https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/MockBean.html. Acessado em 30 jun.. 2021.

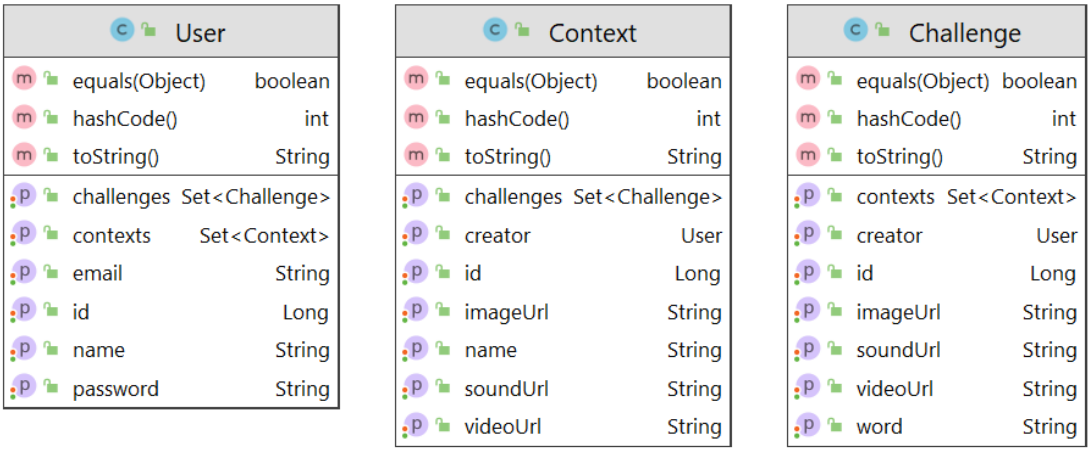

Figura 3. Entidades do EducAPI

Fonte: O autor

Com o módulo a ser testado definido, seguimos para a próxima etapa, a de definir as ferramentas de execução de testes, que é explicada na seção a seguir.

3.2 Definindo as Ferramentas de Execução de Testes

Nesta subseção apresentamos o processo de escolha das ferramentas utilizadas para implementar os testes unitários e os critérios para a escolha das ferramentas.

Para definirmos as ferramentas de execução de testes após entender o EducAPI, foi necessário definirmos os critérios para a escolha das ferramentas. Visando a fácil adoção dos testes e dos padrões nesse artigo de forma simples nos mais diversos contextos de aplicações que utilizem o *framework* Spring, levamos em consideração três critérios: ser indicada na documentação do *framework* Spring, ser popularmente utilizada e de simples configuração no contexto deste projeto.

Com os critérios em consideração, observamos que a documentação oficial do Spring Boot[16] no tópico 5.26.1, responsável por informar quais as bibliotecas do escopo de testes estão presentes na dependência "*spring-boot-starter-test*", destaca o JUnit como "*The de-facto standard for unit testing Java applications.*" que em tradução livre significa "O padrão de fato para testes de unidade em aplicações Java". Neste tópico também é citado o Mockito como um *framework* para *mocks* em Java e o Spring Test que contém o MockMvc como um conjunto de utilitários e suporte para testes em aplicativos Spring Boot.

Logo, decidimos usar o JUnit para realizar os testes referentes ao *Service*. Escolhemos também o Mockito para construir os *mocks* quando necessário, tanto nos testes referentes ao *service* quanto ao *controller*, e com isso isolar o componente a ser testado dos componentes externos. A última ferramenta definida foi o Mockito, que foi a ferramenta que decidimos utilizar nos testes referentes ao *controller*, visto que no *controller* é preciso simular a requisição HTTP na rota que a API disponibiliza.

[16] "Spring Boot Reference Documentation." https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/. Acessado em 23 jun.. 2021.

Após as ferramentas definidas, demos início à implementação dos testes, cujo processo descrevemos na próxima seção.

3.3 Implementando os Testes

Dado o conhecimento a respeito do sistema EducAPI e a definição das ferramentas de execução de testes, demos então início ao processo de implementação dos testes unitários. Para explicarmos este processo e os obstáculos encontrados, iremos descrever inicialmente o processo de forma geral, e em seguida descrevemos brevemente os principais desafios encontrados na implementação dos testes do *service*, e do *controller*, que neste sistema é chamado de *resource*.

O processo de implementação dos testes no *service* e no *resource* de usuário, que tem como seus respectivos nomes `UserService` e `UserResource`, consistiu em analisar a estrutura da classe a ser testada, planejar os testes a serem implementados, visando o máximo de cobertura das funcionalidades dessas classes, e implementar os testes planejados utilizando as ferramentas de execução de testes definidas. Para a implementação dos testes utilizando as ferramentas definidas, seguimos uma estrutura simples, chamada de AAA (*Arrange-Act-Assert*[17]) em que inicialmente é realizada toda a preparação necessária, por exemplo criando os objetos ou configurando os *mocks*, em seguida são realizadas as ações, em que é invocado o método a ser testado, e finaliza-se com a verificação, em que se verifica se o resultado é o esperado.

Durante esse processo de implementação dos testes no `UserService` e da classe que dá suporte a esta gerenciando o acesso de usuários, que é o `JWTService`, nos deparamos com diversos desafios. Dentre eles, quatro se destacaram, que foram: qual projeto utilizar para os testes, como não repetir as mensagens de erro (*exceptions*) no formato *string*, até que ponto é preciso isolar um método para testes no EducAPI e como utilizar o mock em variáveis com a anotação `@Value`.

No processo de implementação dos testes no `UserResource`, os principais desafios ocorreram devido à falta de familiaridade com a ferramenta de execução de testes, o *MockMvc*. Dentre os desafios encontrados, destacam-se erros referentes à codificação de caracteres e referentes à falta de dados nas requisições enviadas nos testes.

Ao fim do processo de implementação dos testes em todos os métodos utilizados nas classes propostas, obtivemos 85,71% de cobertura de métodos referentes às classes `UserService`, `UserResource` e `JWTService`. De tal cobertura, os 14,29% restantes, referem-se a dois métodos que estão no `UserService` mas não são utilizados no sistema EducAPI. Na Seção 4 descrevemos esses desafios em maiores detalhes e como foram superados.

4. Desafios e Lições Aprendidas ao Implementar Testes para o EducAPI

Nesta seção, apresentamos os principais desafios e lições aprendidas no processo de criação de testes para o sistema EducAPI. Estes desafios e lições foram obtidos por meio da experiência ao implementar os testes no módulo de Usuário

[17] "Arrange Act Assert - C2 wiki." http://wiki.c2.com/?ArrangeActAssert. Acessado em 28 jun.. 2021.

que corresponderam às classes de testes `JWTServiceTest`, `UserResourceTest` e `UserServiceTest`. Antes destes testes, a cobertura de código do EducAPI era de 0%, e após a implementação dos testes ela passou para 35% das classes de todo o sistema e 18% de todas as linhas de código do sistema. O código completo destas classes se encontram nos Apêndices A, B e C. Considerando as classes em que decidimos nos concentrar, vimos que a cobertura de suas linhas de código foi de 94,91%, sendo as 5,09% referentes a funções que não estão sendo utilizadas em todo o sistema.

4.1. Estrutura de Código

Um dos primeiros desafios ao se criar testes para sistemas que usam o *Framework* Spring é a forma de estruturar o código dos testes. Nesse sentido, os códigos foram estruturados de modo a facilitar o entendimento e manutenção dos testes, iniciando a classe de teste com a criação dos objetos necessários. Em cada caso de teste (método marcado com a anotação `@Test`) é invocado o método a ser testado, e então são executadas as verificações para checar se o resultado obtido está de acordo com o resultado esperado. Por exemplo, no caso do `UserServiceTest` é criada uma instância da classe `UserService` e objetos utilizados por esta classe, como podemos observar na Listagem 2.

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    private final UserRegisterDTO userRegisterDTO =
UserBuilder.anUser().buildUserRegisterDTO();
    private final Optional<User> userOptional =
UserBuilder.anUser().buildOptionalUser();

    @Mock
    private UserRepository userRepository;
    @Mock
    private JWTService jwtService;
    @InjectMocks
    private UserService service;


    /**
     * Test the find method
     *
     * Verifying if when find an User using your user token, the
response is User
     *
     * @throws InvalidUserException if have not an UserDTO mocked with
the email used in the test
     */
    @Test
    public void findAUserTest() throws InvalidUserException {
```

```
        Mockito.when(jwtService.recoverUserEmailByToken("token
valid")).thenReturn(Optional.of("teste@teste.com"));

Mockito.when(userRepository.findByEmail("teste@teste.com")).thenReturn
(userOptional);

        User response = service.find("token valid");

        assertEquals(userOptional.get().getEmail(),
response.getEmail());
        assertEquals(userOptional.get().getName(),
response.getName());
        assertEquals(userOptional.get().getPassword(),
response.getPassword());
    }
```

Listagem 2. Trecho de código da classe `UserServiceTest`

Fonte: O autor

A parte inicial do código de cada método de teste apresenta a configuração dos *mocks*, seguida da inicialização dos objetos e chamadas às funções, e finalizando então com todas as asserções necessárias para os testes, como mostrado no teste de nome `findAUserTest`, presente na Listagem 2, e que verifica se ao invocar a operação `find` é retornado um usuário com o nome, email e senha configurados no teste.

O código da classe mostrado na Listagem 2 inicia com a anotação `@ExtendWith(MockitoExtension.class)`, que informa ao JUnit que esta classe de teste necessita do Mockito e este será chamado ao executar os testes nesta classe. Em seguida, são declarados os objetos que serão utilizados nos mais diversos testes relacionados ao `UserService`, como por exemplo o objeto `userRegisterDTO` e o `userOptional`. Também são declarados os objetos das classes que se comunicam com a classe a ser testada, como o `UserRepository` e o `JWTService`. Para substituir o comportamento real de objetos dessas classes, é utilizada a anotação `@Mock`, informando que é utilizada uma simulação da classe real, permitindo a utilização dos *mocks* que serão configurados nos testes. É finalizada a parte de declaração de objetos com o objeto a ser testado, que neste caso é o `UserService` em conjunto com a anotação `@InjectMocks`, que indica que nesse objeto serão injetados os objetos marcados com a anotação `@Mock`.

Após essas configurações iniciais, são implementados os métodos de teste. No caso do método de teste `findAUserTest` da Listagem 2, podemos observar que nos passos iniciais são configurados os *mocks* que o método `find` necessita por meio do *Mockito*. Posteriormente, é chamado o método `find`, que é o método a ser testado e em seguida acontecem as verificações, utilizando várias chamadas ao método `assertEquals` do JUnit. Neste teste é chamado o método `find` e comparamos se o retorno (representado pela variável `response`) apresenta todas as informações esperadas (valores configurados para o nome, email e senha).

Um outro exemplo de classe de teste implementada está ilustrado pela Listagem 3. Esta classe contém uma estrutura semelhante à que foi mostrada na Listagem 2. Contudo, por se tratar de outra camada do sistema, a *Controller*, são utilizadas anotações diferentes, como o `@WebMvcTest(controllers = UserResource.class)`, para identificar a classe a ser testada e efetuar as configurações necessárias. Posteriormente esta classe apresenta a declaração dos objetos que serão utilizados nos diversos testes relacionados ao `UserResource`. Neste caso, é utilizada a anotação `@Autowired` para instanciar os objetos necessários para o teste (injetar dependências necessárias), que no caso são da classe `ObjectMapper` e `MockMVC`. Além disso, utilizamos a anotação `@MockBean` para criar um mock para a classe `UserService` e que será utilizado no teste.

```
@WebMvcTest(controllers = UserResource.class)
public class UserResourceTest {

   private final String uriBase = "/v1/api";
   private final UserRegisterDTO userRegisterDTO =
UserBuilder.anUser().buildUserRegisterDTO();
   private final User user = UserBuilder.anUser().buildUser();
   private final Optional<User> optionalUser =
UserBuilder.anUser().buildOptionalUser();
   private final UserDTO userDTO =
UserBuilder.anUser().withId(1L).buildUserDTO();

   @Autowired
   ObjectMapper objectMapper;
   @Autowired
   private MockMvc mockMvc;
   @MockBean
   private UserService service;

   /**
    * Test the find method
    *
    * Verifying if when find an User using your user token, the
response is an json with the User data
    * @throws Exception if have not an token mocked with the token
used in the test
    */
   @Test
   public void findAnUserTest() throws Exception {

       Mockito.when(service.find("token")).thenReturn(user);

       mockMvc.perform(get(uriBase +
"/auth/users").characterEncoding("UTF-8")
               .header("Authorization", "token"))
               .andExpect(status().isOk())
.andExpect(content().string(objectMapper.writeValueAsString(user)));
```

```
    }
```

Listagem 3. Trecho de código da classe `UserResourceTest`

Fonte: O autor

A declaração dos casos de testes utiliza a mesma anotação da Listagem 2, ou seja, o `@Test`. Em cada caso de teste, também temos como passo inicial as configurações dos *mocks* necessários para o teste. Porém, para o caso de teste mostrado na Listagem 3, onde se testa o *controller*, a forma de chamar o método a ser testado é diferente, pois é necessário efetuar a requisição HTTP por meio do objeto da classe MockMvc. Para isso, utilizamos a URI a ser testada (`uriBase + "/auth/users"`) e configuramos essa requisição especificando, por exemplo, o tipo de configuração dos caracteres (`characterEncoding("UTF-8")`) e o cabeçalho (`header("Authorization", "token")`). Em seguida, são realizadas as verificações. Porém, ao invés de usar asserções, utilizamos a chamada ao método `andExpect` onde especificamos o comportamento esperado. Para o caso deste exemplo do caso de teste da Listagem 3, é verificado se o *status code* (código que informa o *status* da requisição) está correto e se o conteúdo da resposta da requisição é o esperado.

4.2. Desafios Relativos à Implementação dos Testes do `UserService` e `JWTService` e Lições Aprendidas com estes Desafios

Nesta subseção descrevemos de forma mais detalhada os quatro desafios relacionados à implementação dos testes nas classes `UserService` e `JWTService` (classe responsável pela autenticação) e que foram: qual projeto utilizar, como não repetir as mensagens de erro (texto das *exceptions*), até que ponto é preciso isolar um método para testes no EducAPI e como utilizar o mock em variáveis com a anotação `@Value`.

O primeiro desafio, relacionado ao projeto a utilizar, surgiu após observarmos a classe a ser testada. Constatamos que o sistema EducAPI utiliza diversas classes semelhantes. Dentre essas, existe o `User`, `UserDTO` e `UserRegisterDTO`, que são utilizadas na classe `UserService`. As semelhanças podem ser vistas na Figura 4, em que são mostradas três entidades com pouca variação de atributos. Por exemplo, a classe `UserDTO` difere da classe `UserRegisterDTO` apenas pelo acréscimo do id. A classe `User` difere da `UserDTO` pelo acréscimo dos atributos *challenges* e *contexts*, que são os contextos e desafios atrelados a um usuário.

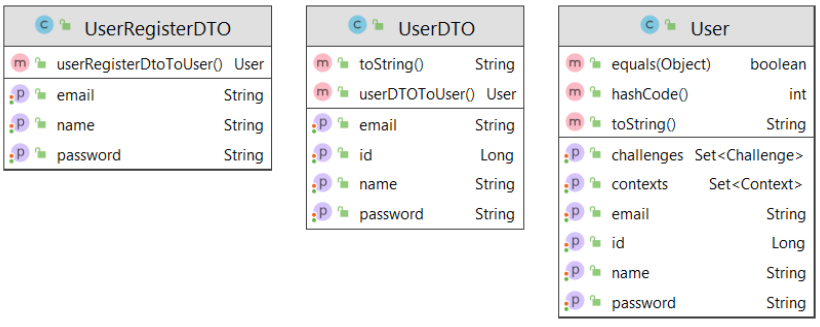

Figura 4. `User`, `UserDTO` e `UserRegisterDTO` do EducAPI

Fonte: O autor

A necessidade de um bom projeto se deu ao fato de que em testes de unidade, usualmente é necessária a criação de diversos objetos diferentes mas com dados semelhantes, como por exemplo dois usuários com os mesmos dados, porém com e-mails diferentes. Para superar este desafio, analisamos alguns dos padrões de projetos conhecidos que têm como objetivo a criação de objetos complexos de forma simplificada, visto que padrões de projeto são formas de documentar boas práticas de projeto de software. Nesta análise, dois padrões se destacaram, o *Builder* e o *Test Data Builder*. Observamos que o padrão *Builder* e o *Test Data Builder* têm como foco a criação de objetos complexos por meio de um passo a passo, sendo *Test Data Builder*, uma adaptação do *Builder* voltada mais especificamente para testes.

Para definir qual dos dois padrões utilizar neste projeto, foi realizada uma POC (prova de conceito), em que implementamos os dois padrões no sistema para identificar qual dos dois é o mais simples de utilizar, que gera menos classes e qual o mais legível. Ao compararmos ambas as POCs (Figuras 5 e 6), vimos que o *Test Data Builder* (Figura 5) apresentou uma melhor legibilidade e simplicidade, porém ainda seria necessária uma classe *Test Data Builder*, para cada classe semelhante do EducAPI. Portanto, fizemos uma adaptação do *Test Data Builder*[18] para reduzir a quantidade de classes necessárias, conforme pode ser visto na Figura 7. O código fonte da classe `UserBuilder`, que foi o resultado da adaptação, encontra-se no Apêndice D.

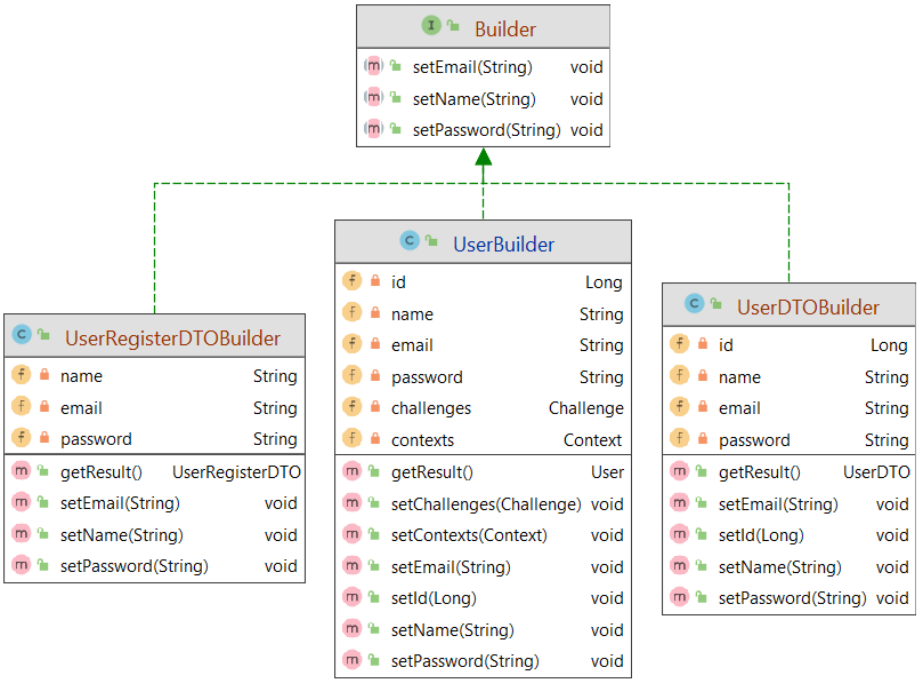

Figura 5. Diagrama da implementação do Builder no EducAPI

Fonte: O autor

[18] "Adaptação do Test Data Builder." https://github.com/a4s-ufpb/EducAPI/blob/master/src/test/java/br/ufpb/dcx/apps4society/educapi/unit/domain/builder/UserBuilder.java. Acessado em 29 jun.. 2021.

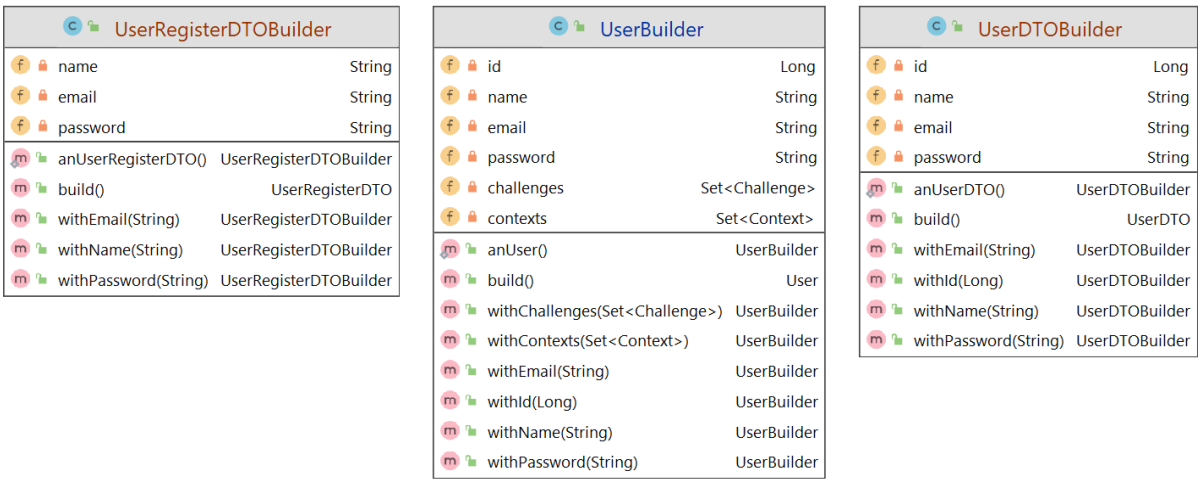

Figura 6. Diagrama da implementação do Test Data Builder no EducAPI

Fonte: O autor

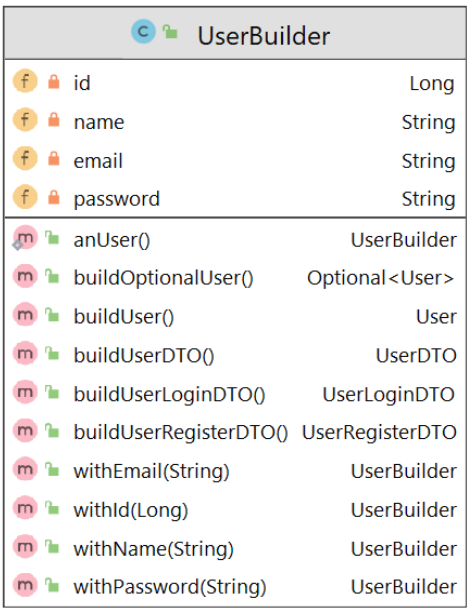

Figura 7. Diagrama da adaptação do Test Data Builder no EducAPI

Fonte: O autor

O segundo desafio encontrado no processo de construção de testes do `UserService` é referente à repetição de mensagens de erro (literais utilizadas nos códigos da aplicação) e que se repetiam nos testes. Ao invés de repetir os literais, consideramos importante criar constantes. Com isso, buscamos evitar problemas de manutenção do código da aplicação e dos testes, visto que caso seja preciso realizar uma alteração na mensagem de erro, seria necessário realizar uma busca em todo o sistema para encontrar os locais em que a mensagem de erro é utilizada, enquanto que se for utilizada uma constante ou alguma classe que fornece mensagens de erro, seria

necessário alterar o valor em apenas um local, reduzindo a possibilidade de que alguma mensagem de erro fique desatualizada.

Com o objetivo de superar este desafio, vimos que o livro *Refactoring: Improving the Design of Existing Code*, escrito por Fowler e Beck (2018, p. 160) descreve a técnica de refatoração intitulada encapsular variável (*Encapsulate Variable*), e constatamos que o conceito é semelhante ao que precisávamos. Após uma pesquisa mais aprofundada, observamos que a técnica mais adequada para essa situação no contexto desse sistema e proposto em IDEs como o IntelliJ é a funcionalidade *Extract Constant*, em que a principal diferença é que, em vez de utilizar uma variável, é utilizada uma constante. O resultado de sua utilização foi a criação da classe *Messages*, contendo as mensagens de erro encontradas no código do sistema ao longo da implementação dos testes, e definidas como constantes. Um outro resultado foi a notificação aos desenvolvedores do EducAPI a respeito deste ponto de melhoria no design de código.

O terceiro desafio encontrado no desenvolvimento dos testes do `UserService` foi identificar até que ponto é preciso isolar um método para realizar os testes unitários no EducAPI. Isso ocorreu devido a um método invocar outro método pertencente à mesma classe e que é privado. Porém, um método privado não deve ser acessado diretamente por qualquer classe externa, como por exemplo, uma classe de testes. Assim, para testar isoladamente o método privado, é necessária a utilização de técnicas como reflexão. Contudo, a utilização desta técnica não nos pareceu muito apropriada pois aumenta a complexidade dos testes. Devido a isto, decidimos que isolaríamos de interferências externas apenas a classe, ou seja, invocações a outras classes, mas as invocações de seus próprios métodos internos, não. Sendo assim, os métodos privados eram testados através das invocações feitas a eles pelos métodos públicos sendo testados.

O quarto desafio foi encontrado durante a implementação dos testes do `JWTService` e estava relacionado à utilização de *mocks* em variáveis com a anotação `@Value`. Este desafio foi um dos maiores enfrentados, visto que a princípio não tínhamos o conhecimento de como a anotação `@Value` funciona e os testes nas classes que continham esta anotação falhavam. Na mensagem da falha se indicava que uma variável estava com valor nulo. Sendo assim, foi necessário pesquisar e compreender melhor como essa anotação funciona. Vimos que ela injeta um valor em uma variável, método ou construtor. Devido a isso, ela é comumente utilizada para definir configurações que são específicas ao ambiente em que o software será executado.

Por se tratar de injeção de dados e pelo fato da execução dos testes ocorrer em um ambiente isolado, essa injeção não ocorre e gera o erro detectado (da variável estar com um valor nulo). Após entender o que esta anotação faz, seguimos para o próximo passo, que foi o de definir como configurar os mocks para esta anotação.

Após diversas tentativas, o único método que encontramos até o momento da escrita deste artigo, foi por meio da utilização da técnica de reflexão (que não é a única solução, mas foi a que resolveu o nosso problema em tempo hábil). Por meio de uma pesquisa nos métodos fornecidos pelo *framework Spring*, encontramos o método `ReflectionTestUtils`, em que definimos o objeto que possui a *annotation*, o nome da

variável e o valor a ser atribuído a ela, como mostrado no trecho de código da Listagem 4, referente à classe de teste JWTServiceTest.

```
@ExtendWith(MockitoExtension.class)
public class JWTServiceTest {

   private final UserLoginDTO userLoginDTO =
UserBuilder.anUser().buildUserLoginDTO();
   private final Optional<User> userOptional =
UserBuilder.anUser().buildOptionalUser();
   private final String invalidToken = "Bearer
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJtYWlhd2VlZUB0ZXN0LmNvbSIsImV4cCI6MTYxN
TM" +

"5OTkyN30.1qNJIgwjlnm6YcZuIDFLZrQLs58qOwLFkCtXOcaUD-fQZyTa4usOMVgGa19E
m_e8WdoXfnaJSv9O-c8IRp-C9Q";
   @Mock
   UserRepository userRepository;
   @InjectMocks
   JWTService service;

   /**
   * Before each test, sets the value of the "TOKEN_KEY" field in
JWTService
   *
   * If this value is not set, the TOKEN_KEY will be null and generate
an SignatureException in tests
   */
   @BeforeEach
   public void setUp() {
      ReflectionTestUtils.setField(service, "TOKEN_KEY", "it's a
token key");
   }
```

Listagem 4. Trecho de código da classe `JWTServiceTest`

Fonte: O autor

Com todos os percalços resolvidos, finalizamos os testes referentes ao `UserService` e `JWTService` e demos sequência aos testes referentes à classe `UserResource`.

4.3. Desafios Relativos à Implementação dos Testes do `UserResource` e Lições Aprendidas com estes Desafios

Nesta subseção descrevemos de forma mais detalhada os desafios relacionados à implementação dos testes para a classe `UserResource` que eram referentes ao envio de dados através de requisições usando o *MockMvc*.

Ao implementar e executar os testes relativos à classe `UserResource`, apareceram erros referentes à codificação de caracteres e referentes à falta de dados nas

requisições enviadas nos testes, Para resolver o primeiro problema, vimos que era necessário explicitar o *character encoding*[19] (codificação dos caracteres utilizados na requisição). Vimos que em uma requisição HTTP é enviado em conjunto com os dados o tipo de dado (*Content-Type*) e o tipo de compressão utilizada (o *character encoding*) que está sendo enviado. Em algumas ferramentas de execução de testes a configuração da codificação utilizada é definida por padrão na codificação UTF-8, e apenas em casos específicos é necessário alterar esta configuração. Contudo, no *MockMvc* esta configuração não é feita por padrão, e então foi preciso que explicitássemos essa codificação por meio da chamada `characterEncoding("UTF-8")`, conforme mostrado na Listagem 5.

Com relação aos erros relativos à falta de dados nas requisições utilizadas nos testes, vimos que para enviar um dado no corpo da requisição no *MockMvc* é preciso utilizar o método `content()`. Em uma requisição HTTP, no *header* (cabeçalho) são enviados dados como o *status* da requisição, o *content-type*, a quantidade de *bytes* que há no corpo da requisição, dentre outras informações. No corpo da requisição, chamado de *body*, são enviados os dados, no formato (*content-type*) definido. Um exemplo de trecho de código de testes onde tivemos de utilizar o método `content` é mostrado na Listagem 5. Conforme podemos observar neste trecho, para passar na requisição dados sobre o usuário no formato *json (Java Script Object Notation)*, foi utilizado o método content recebendo como parâmetro os dados do usuário a serem passados nesse formado no corpo da requisição.

```
 @Test
public void insertAnUserTest() throws Exception {

Mockito.when(service.insert(any(UserRegisterDTO.class))).thenReturn(us
erDTO);

   mockMvc.perform(post(uriBase + "/users")
.contentType("application/json")
.characterEncoding("UTF-8")
.content(objectMapper.writeValueAsString(userRegisterDTO)))
.andExpect(status().isCreated()).andExpect(content().string(objectMapp
er.writeValueAsString(userDTO)));
}
```

Listagem 5. Trecho de código da classe `UserResourceTest`

Fonte: O autor

Em outras ferramentas de execução de testes, para enviar um dado no corpo da requisição é utilizado um método de nome *body* (corpo). No Postman, por exemplo, há um local específico para colocar o corpo da requisição, no caso, o *json*, e ele tem o nome de *body*. Contudo, no MockMvc, não há nenhum método de nome *body* ou semelhante. No processo de tentar resolver esse problema, vimos que no MockMvc há o método `content()`, e ao utilizá-lo constatamos que era o método necessário para efetuar a requisição HTTP enviando um dado. Resolvidos todos os desafios, foi dado

[19] "Character encoding - Glossário | MDN." 22 jun.. 2021, https://developer.mozilla.org/pt-BR/docs/Glossary/character_encoding. Acessado em 30 jun.. 2021.

prosseguimento e posteriormente finalizamos a implementação dos testes e chegamos aos resultados apresentados como lições neste trabalho e relatados anteriormente.

5. Considerações Finais e Trabalhos Futuros

Este artigo teve como objetivo relatar a experiência sobre o processo de construção de testes unitários no sistema EducAPI, um sistema construído utilizando o framework Spring.

Considerando o que foi apresentado ao longo do artigo, acreditamos que o objetivo principal foi atingido, uma vez que descrevemos o processo para a implementação dos testes no sistema EducAPI, os principais desafios encontrados e como os superamos através de soluções que foram apresentadas no trabalho.

Este relato contribui também para um melhor direcionamento para a implementação dos testes visto que ao descrevermos o motivo para as tomadas de decisões, tornamos mais viáveis a adaptação do processo descrito neste artigo para a implementação de testes nos mais diversos sistemas baseados em Spring. Outro ponto é que todo o código criado ao longo da escrita deste artigo está disponível no repositório EducAPI (https://github.com/a4s-ufpb/EducAPI), mantido pelo projeto Apps4Society. Além disso, por meio do histórico das alterações realizadas, é possível acompanhar a evolução dos testes, bem como o resultado das POCs realizadas ao longo da implementação.

Como trabalhos futuros, pretendemos implementar testes de integração e relatar este processo, possibilitando um nível a mais de testes. Além disso, pretendemos também realizar análises com base nos testes implementados e no processo descrito neste artigo, para identificar o grau de adaptação necessário para implementar testes em outros módulos e até em outros sistemas. Um outro trabalho futuro planejado é a documentação de boas práticas para o design de testes de Sistemas Spring como padrões.

6. Referências

Buschmann, Frank; Henney, Kevlin; Schmidt, Douglas. Pattern-Oriented Software Archtecture: A Pattern Language for Distributed Computing. Volume 4 ed. Wiley, 2007.

Cohn, Mike. Succeeding with Agile: Software Development Using Scrum. 1ª. ed. Addison-Wesley Professional, 2009.

EducAPI e EducAPI Manager. Apps4Society, [s/d]. Disponível em: <https://apps4society.dcx.ufpb.br/?page_id=942>. Acesso em: 25 de fev. de 2021.

Fowler, Martin; Beck, Kent. Refactoring: Improving the Design of Existing Code. 2ª. ed. Addison-Wesley Professional, 2018.

Fowler, Martim; Rice, David; Foemmel, Matthew; Hieatt, Edward; Mee, Robert; Stafford, Randy. Patterns of Enterprise Application Architecture. 1ª. ed. Addison-Wesley Professional, 2002.

ISTQB. Certified tester: foundation level syllabus 2018br. 2018. Disponível em: <https://bstqb.org.br/b9/doc/syllabus_ctfl_2018br.pdf>. Acessado em: 29 de jun. de 2021.

Maldonado, J. Automatização de Teste de Software com Ferramentas de Software Livre. [Elsevier, 2016, 2a edição]: Grupo GEN, 2018. 9788595155305. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788595155305/>. Acesso em: 25 fev 2021.

Mockito Framework, [s/d]. Disponível em: <https://site.mockito.org/>. Acessado em: 29 de jun. de 2021;

Niemeyer, Patrick; Leuck, Daniel. Learning Java: A Bestselling Hands-On Java Tutorial. 1ª. ed. O'Reilly Media, 2013.

Silva, Matheus; Alves, Robson. Rebouças, Ayla. SisAlfa: Um Serviço Colaborativo para apoiar a criação de Sistemas para Alfabetização. Anais dos Workshops do Congresso Brasileiro de Informática na Educação, [S.l.], p. 418, out. 2017. ISSN 2316-8889. Disponível em: <http://www.br-ie.org/pub/index.php/wcbie/article/view/7417>. Acesso em: 23 jun. 2021.

Spring Boot Reference Documentation. Spring, [s/d]. Disponível em: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-testing/>. Acesso em: 13 de jan. de 2021.

Spring Framework, [s/d]. Disponível em: <https://spring.io/projects/spring-framework>. Acesso em: 23 de jun. de 2021.

Tudose, Catalin; JUnit in Action. 3ª. ed. Manning Publications, 2020.

Apêndice A - Código fonte da classe de teste `UserResourceTest.java`

Disponível em:
https://github.com/a4s-ufpb/EducAPI/blob/dev/src/test/java/br/ufpb/dcx/apps4society/educapi/unit/resource/UserResourceTest.java

Acessado em: 13 jul. de 2021

```
package br.ufpb.dcx.apps4society.educapi.unit.resource;

import br.ufpb.dcx.apps4society.educapi.domain.User;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserDTO;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserRegisterDTO;
import br.ufpb.dcx.apps4society.educapi.resources.UserResource;
import br.ufpb.dcx.apps4society.educapi.services.UserService;
import
br.ufpb.dcx.apps4society.educapi.services.exceptions.InvalidUserExcept
ion;
import
br.ufpb.dcx.apps4society.educapi.services.exceptions.UserAlreadyExists
Exception;
import
br.ufpb.dcx.apps4society.educapi.unit.domain.builder.UserBuilder;
import br.ufpb.dcx.apps4society.educapi.util.Messages;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Optional;

import static org.mockito.ArgumentMatchers.any;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.cont
ent;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.stat
us;

/**
* This class have only unit tests reference to UserResource
*
* @author Enos Tetéo
*/
@WebMvcTest(controllers = UserResource.class)
public class UserResourceTest {
```

```
   private final UserRegisterDTO userRegisterDTO =
UserBuilder.anUser().buildUserRegisterDTO();
   private final String uriBase = "/v1/api";
   private final User user = UserBuilder.anUser().buildUser();
   private final Optional<User> optionalUser =
UserBuilder.anUser().buildOptionalUser();
   private final UserDTO userDTO =
UserBuilder.anUser().withId(1L).buildUserDTO();
   @Autowired
   ObjectMapper objectMapper;
   @Autowired
   private MockMvc mockMvc;
   @MockBean
   private UserService service;

   /**
    * Test the find method
    *
    * Verifying if when find an User using your user token, the
response is an json with the User data
    * @throws Exception if have not an token mocked with the token
used in the test
    */
   @Test
   public void findAnUserTest() throws Exception {
       Mockito.when(service.find("token")).thenReturn(user);

       mockMvc.perform(get(uriBase +
"/auth/users").characterEncoding("UTF-8")
               .header("Authorization", "token"))
               .andExpect(status().isOk())

.andExpect(content().string(objectMapper.writeValueAsString(user)));
   }

   /**
    * Test the find method with a token that return an
InvalidUserException
    *
    * Verifying if when use a find method with an invalid token, the
response is forbidden
    * @throws Exception if have not an find method in UserService
mocked
    */
   @Test
   public void findAInvalidUserTest() throws Exception {
       Mockito.when(service.find("token")).thenThrow(new
InvalidUserException(Messages.INVALID_USER_CHECK_THE_TOKEN));
```

```
        mockMvc.perform(get(uriBase +
"/auth/users").characterEncoding("UTF-8")
                .header("Authorization", "token"))
                .andExpect(status().isForbidden());
    }

    /**
     * Test the insert method
     *
     * Verifying if when insert an json with an UserRegisterDTO,
     * the response has contain an json with the data the
UserRegisterDTO
     *
     * @throws Exception if have not an find method in UserService
mocked
     */
    @Test
    public void insertAnUserTest() throws Exception {

Mockito.when(service.insert(any(UserRegisterDTO.class))).thenReturn(us
erDTO);

        mockMvc.perform(post(uriBase +
"/users").contentType("application/json").characterEncoding("UTF-8")

.content(objectMapper.writeValueAsString(userRegisterDTO))).andExpect(
status().isCreated())

.andExpect(content().string(objectMapper.writeValueAsString(userDTO)))
;
    }

    /**
     * Test the insert method with UserAlreadyExistsException
     *
     * Verifying if when insert an json with an UserRegisterDTO that
already exist in the system,
     * the response is not content
     *
     * @throws Exception if have not an user mocked
     */
    @Test
    public void insertAnUserAlreadyExistTest() throws Exception {

Mockito.when(service.insert(any(UserRegisterDTO.class))).thenThrow(new
UserAlreadyExistsException(Messages.USER_ALREADY_EXISTS));

        mockMvc.perform(post(uriBase +
"/users").contentType("application/json").characterEncoding("UTF-8")
```

```
.content(objectMapper.writeValueAsString(userRegisterDTO))).andExpect(
status().isNoContent());
    }

    /**
     * Test the update method
     *
     * Verifying if when update an json with an UserRegisterDTO and
passing an token in the Autorization,
     * the response has contain an json with the UserDTO edited
     *
     * @throws Exception if have not mocked the update method in
UserService
     */
    @Test
    public void updateAnUserTest() throws Exception {
        Mockito.when(service.update(any(),
any(UserRegisterDTO.class))).thenReturn(userDTO);

        mockMvc.perform(put(uriBase +
"/auth/users").contentType("application/json").characterEncoding("UTF-
8")
                .header("Authorization", "token")

.content(objectMapper.writeValueAsString(userRegisterDTO)))
                .andExpect(status().isOk()).andExpect(content()
                .string(objectMapper.writeValueAsString(userDTO)));
    }

    /**
     * Test the update method with a token that return an
InvalidUserException
     *
     * Verifying if when update an json with an UserRegisterDTO and
passing an invalid token in the Autorization,
     * the response has contain the status code is forbidden
     *
     * @throws Exception if have not mocked the update method in
UserService
     */
    @Test
    public void updateAnUserWithInvalidTokenTest() throws Exception {
        Mockito.when(service.update(any(),
any(UserRegisterDTO.class))).thenThrow(new
InvalidUserException(Messages.INVALID_USER_CHECK_THE_TOKEN));

        mockMvc.perform(put(uriBase +
"/auth/users").contentType("application/json").characterEncoding("UTF-
8")
                .header("Authorization", "token")
```

```
.content(objectMapper.writeValueAsString(userRegisterDTO)))
                .andExpect(status().isForbidden());
    }

    /**
     * Test the delete method
     *
     * Verifying if when delete an User parsing an Authorization with
the token,
     * the response has contain the UserDTO deleted
     *
     * @throws Exception if have not mocked the delete method in
UserService
     */
    @Test
    public void deleteAnUserTest() throws Exception {
        Mockito.when(service.delete("token")).thenReturn(userDTO);

        mockMvc.perform(delete(uriBase +
"/auth/users").characterEncoding("UTF-8")
                .header("Authorization", "token"))
                .andExpect(status().isOk())

.andExpect(content().string(objectMapper.writeValueAsString(userDTO)))
;
    }

    /**
     * Test the delete method
     *
     * Verifying if when delete an User parsing an Authorization with
the token invalid,
     * the response has contain the InvalidUserException
     *
     * @throws Exception if have not mocked the delete method in
UserService
     */
    @Test
    public void deleteAnUserWithinvalidTokenTest() throws Exception {
        Mockito.when(service.delete("token")).thenThrow(new
InvalidUserException(Messages.INVALID_USER_CHECK_THE_TOKEN));

        mockMvc.perform(delete(uriBase +
"/auth/users").characterEncoding("UTF-8")
                .header("Authorization", "token"))
                .andExpect(status().isForbidden());
    }
}
```

Apêndice B - Código fonte da classe de teste JWTServiceTest.java

Disponível em:
https://github.com/a4s-ufpb/EducAPI/blob/dev/src/test/java/br/ufpb/dcx/apps4society/educapi/unit/service/JWTServiceTest.java

Acessado em: 13 jul. de 2021

```
package br.ufpb.dcx.apps4society.educapi.unit.service;

import br.ufpb.dcx.apps4society.educapi.domain.User;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserLoginDTO;
import br.ufpb.dcx.apps4society.educapi.repositories.UserRepository;
import br.ufpb.dcx.apps4society.educapi.response.LoginResponse;
import br.ufpb.dcx.apps4society.educapi.services.JWTService;
import
br.ufpb.dcx.apps4society.educapi.services.exceptions.InvalidUserExcept
ion;
import
br.ufpb.dcx.apps4society.educapi.unit.domain.builder.UserBuilder;
import br.ufpb.dcx.apps4society.educapi.util.Messages;
import io.jsonwebtoken.Jwts;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.test.util.ReflectionTestUtils;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;

/**
* This class have only unit tests related to JWTService.
*
* @author Enos Teteo
*/
@ExtendWith(MockitoExtension.class)
public class JWTServiceTest {

   private final UserLoginDTO userLoginDTO =
UserBuilder.anUser().buildUserLoginDTO();
   private final Optional<User> userOptional =
UserBuilder.anUser().buildOptionalUser();
   private final String invalidToken = "Bearer
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJtYWlhd2VlZUB0ZXN0LmNvbSIsImV4cCI6MTYxN
TM" +

"5OTkyN30.1qNJIgwjlnm6YcZuIDFLZrQLs58qOwLFkCtXOcaUD-fQZyTa4usOMVgGa19E
```

```
m_e8WdoXfnaJSv9O-c8IRp-C9Q";
    @Mock
    UserRepository userRepository;
    @InjectMocks
    JWTService service;

    /**
     * Format String to token format adding "Bearer " before the String
     * Example:
     *      FormatTo.token("any String");
     *      // return "Bearer any String"
     *
     * @param token any String
     * @return token formatted
     */
    private String formatToToken(String token) {
        return "Bearer " + token;
    }

    /**
     * Extract the content of the token
     * @param token any String encrypted using JWTs with value "it's a
token key"
     * @return subject in the token body
     */
    private String extractSubjectFromToken(String token) {
        return Jwts.parser().setSigningKey("it's a token
key").parseClaimsJws(token).getBody().getSubject();
    }

    /**
     * Before each test, sets the value of the "TOKEN_KEY" field in
JWTService
     *
     * If this value is not set, the TOKEN_KEY will be null and
generate an SignatureException in tests
     */
    @BeforeEach
    public void setUp() {
        ReflectionTestUtils.setField(service, "TOKEN_KEY", "it's a
token key");
    }

    /**
     * Test the authenticate method.
     *
     * Verifying if the received token has the correct email
     * @throws InvalidUserException if the UseLoginDTO is not correctly
mocked
     */
```

```
    @Test
    public void authenticateTest() throws InvalidUserException {

Mockito.when(this.userRepository.findByEmailAndPassword(this.userLogin
DTO.getEmail(), this.userLoginDTO.getPassword()))
                .thenReturn(this.userOptional);

        LoginResponse response =
this.service.authenticate(this.userLoginDTO);
        String subject = extractSubjectFromToken(response.getToken());

        assertNotNull(response.getToken());
        assertEquals(userLoginDTO.getEmail(), subject);
    }

    /**
     * Test exception in the authenticate method.
     *
     * Verifying if an exception is thrown when try to authenticate
without configuring an user in DB
     */
    @Test
    public void authenticateUnregisterUserTest() {
        assertThrows(InvalidUserException.class, () -> {
            this.service.authenticate(this.userLoginDTO);
        });
    }

    /**
     * Test the recoverUser method with a valid token
     *
     * Verifying if the email returned by recoverUser is equals to the
email contained in the token
     * @throws InvalidUserException if the UserLoginDTO not correctly
mocked
     */
    @Test
    public void recoverUserTest() throws InvalidUserException {

Mockito.when(this.userRepository.findByEmailAndPassword(this.userLogin
DTO.getEmail(), this.userLoginDTO.getPassword()))
                .thenReturn(this.userOptional);

        LoginResponse response =
this.service.authenticate(userLoginDTO);
        String token = formatToToken(response.getToken());

        String userRecovered =
this.service.recoverUserEmailByToken(token).get();
```

```
        assertEquals(this.userLoginDTO.getEmail(), userRecovered);
    }

    /**
     * Test the recoverUser method with a null token
     *
     * Verifying if an exception is thrown when try to recoverUser with
a null param
     */
    @Test
    public void revocerUserWithNullTokenTest() {
        assertThrows(SecurityException.class, () -> {
            this.service.recoverUserEmailByToken(null);
        });
    }

    /**
     * Test the recoverUser method with an expired token
     *
     * Verifying if an exception is thrown and have the correctly
message when try to recoverUser with an expired token
     */
    @Test
    public void revocerUserWithExpiredTokenTest() {
        Exception exception = assertThrows(SecurityException.class, ()
-> {
            this.service.recoverUserEmailByToken(invalidToken);
        });
        assertEquals(Messages.TOKEN_INVALID_OR_EXPIRED,
exception.getMessage());
    }
}
```

Apêndice C - Código fonte da classe de teste UserServiceTest.java

Disponível em:
https://github.com/a4s-ufpb/EducAPI/blob/dev/src/test/java/br/ufpb/dcx/apps4society/educapi/unit/service/UserServiceTest.java

Acessado em: 13 jul. de 2021

```
package br.ufpb.dcx.apps4society.educapi.unit.service;

import br.ufpb.dcx.apps4society.educapi.domain.User;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserDTO;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserRegisterDTO;
import br.ufpb.dcx.apps4society.educapi.repositories.UserRepository;
import br.ufpb.dcx.apps4society.educapi.services.JWTService;
import br.ufpb.dcx.apps4society.educapi.services.UserService;
import
br.ufpb.dcx.apps4society.educapi.services.exceptions.InvalidUserExcept
ion;
import
br.ufpb.dcx.apps4society.educapi.services.exceptions.UserAlreadyExists
Exception;
import
br.ufpb.dcx.apps4society.educapi.unit.domain.builder.UserBuilder;
import br.ufpb.dcx.apps4society.educapi.util.Messages;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;

/**
* This class have only unit tests reference to UserService
*
* @author Enos Tetéo
*/
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

   private final UserRegisterDTO userRegisterDTO =
UserBuilder.anUser().buildUserRegisterDTO();
   private final Optional<User> userOptional =
UserBuilder.anUser().buildOptionalUser();

   @Mock
   private UserRepository userRepository;
   @Mock
```

```
    private JWTService jwtService;
    @InjectMocks
    private UserService service;

    /**
     * Test the insert method
     *
     * Verifying if when insert an UserRegisterDTO, the response has
contain the data the UserRegisterDTO
     *
     * @throws UserAlreadyExistsException if have an UserRegisterDTO
mocked with the equals emails
     */
    @Test
    public void insertAUserTest() throws UserAlreadyExistsException {
        UserDTO response = service.insert(this.userRegisterDTO);

        assertEquals(response.getName(),
this.userRegisterDTO.getName());
        assertEquals(response.getEmail(),
this.userRegisterDTO.getEmail());
        assertEquals(response.getPassword(),
this.userRegisterDTO.getPassword());
    }

    /**
     * Test the insert method with already email registered
     *
     * Verifying if when insert an UserRegisterDTO with email
registered in the system, the response is UserAlreadyExistsException
     */
    @Test
    public void insertAUserAlreadyExistTest() {

Mockito.when(this.userRepository.findByEmail(this.userRegisterDTO.getE
mail())).thenReturn(this.userOptional);

        Exception exception =
assertThrows(UserAlreadyExistsException.class, () -> {
            service.insert(this.userRegisterDTO);
        });

        assertEquals(Messages.USER_ALREADY_EXISTS,
exception.getMessage());
    }


    /**
     * Test the find method
     *
```

```
    * Verifying if when find an User using your user token, the
response is User
    *
    * @throws InvalidUserException if have not an UserDTO mocked with
the email used in the test
    */
   @Test
   public void findAUserTest() throws InvalidUserException {
       Mockito.when(jwtService.recoverUserEmailByToken("token
valid")).thenReturn(Optional.of("teste@teste.com"));

Mockito.when(userRepository.findByEmail("teste@teste.com")).thenReturn
(userOptional);

       User response = service.find("token valid");

       assertEquals(userOptional.get().getEmail(),
response.getEmail());
       assertEquals(userOptional.get().getName(), response.getName());
       assertEquals(userOptional.get().getPassword(),
response.getPassword());
   }

   /**
    * Test the find method with a null token
    *
    * Verifying if when use a find method with a null token, the
response is InvalidUserException
    */
   @Test
   public void findAInvalidUserTest() {
       Exception exception = assertThrows(InvalidUserException.class,
() -> {
           service.find(null);
       });
       assertEquals(Messages.INVALID_USER_CHECK_THE_TOKEN,
exception.getMessage());
   }

   /**
    * Test the find method with a user that is not registered in
system
    *
    * Verifying if when use a find method with a token valid and email
not registered in the system, the response is Exception
    */
   @Test
   public void findANotRegisteredUserTest() throws
InvalidUserException {
       Mockito.when(jwtService.recoverUserEmailByToken("token
```

```
valid")).thenReturn(Optional.of("teste@teste.com"));

        Exception exception = assertThrows(InvalidUserException.class,
() -> {
            User response = service.find("token valid");
        });
        assertEquals(Messages.INVALID_USER_CHECK_THE_EMAIL,
exception.getMessage());
    }

    /**
     * Test the update method
     *
     * Verifying if when update an User, the response has contain the
UserDTO edited
     *
     * @throws InvalidUserException if have not mocked the find method
     */
    @Test
    public void updateAUserTest() throws InvalidUserException {
        Mockito.when(jwtService.recoverUserEmailByToken("token
valid")).thenReturn(Optional.of("teste@teste.com"));

Mockito.when(userRepository.findByEmail("teste@teste.com")).thenReturn
(userOptional);

        userRegisterDTO.setName("testUpdate");
        userRegisterDTO.setEmail("testUpdate@mail.com");
        userRegisterDTO.setPassword("testPass");

        assertNotEquals(userOptional.get().getName(),
userRegisterDTO.getName());
        assertNotEquals(userOptional.get().getEmail(),
userRegisterDTO.getEmail());
        assertNotEquals(userOptional.get().getPassword(),
userRegisterDTO.getPassword());

        UserDTO response = service.update("token valid",
userRegisterDTO);


        assertEquals("testUpdate", response.getName());
        assertEquals("testUpdate@mail.com", response.getEmail());
        assertEquals("testPass", response.getPassword());

        assertEquals(userOptional.get().getId(), response.getId());
    }

    /**
     * Test the delete method
```

```
     *
     * Verifying if when delete an User, the response has contain the
UserDTO deleted
     *
     * @throws InvalidUserException
     */
    @Test
    public void deleteAUserTest() throws InvalidUserException {
        Mockito.when(jwtService.recoverUserEmailByToken("token
valid")).thenReturn(Optional.of("teste@teste.com"));

Mockito.when(userRepository.findByEmail("teste@teste.com")).thenReturn
(userOptional);

        UserDTO response = service.delete("token valid");

        assertEquals(userOptional.get().getName(), response.getName());
        assertEquals(userOptional.get().getEmail(),
response.getEmail());
        assertEquals(userOptional.get().getPassword(),
response.getPassword());
        assertEquals(userOptional.get().getId(), response.getId());

    }

}
```

Apêndice D - Código fonte da classe de teste UserBuilder.java

Disponível em:
https://github.com/a4s-ufpb/EducAPI/blob/dev/src/test/java/br/ufpb/dcx/apps4society/educapi/unit/domain/builder/UserBuilder.java

Acessado em: 13 jul. de 2021

```
package br.ufpb.dcx.apps4society.educapi.unit.domain.builder;

import br.ufpb.dcx.apps4society.educapi.domain.User;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserDTO;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserLoginDTO;
import br.ufpb.dcx.apps4society.educapi.dto.user.UserRegisterDTO;

import java.util.Optional;

/**
* Create instances of UserBuilder, UserRegisterDTO, UserLoginDTO or
OptionalUser classes
* Can create an instances with custom or default data
* This class used an adaptation of Test Data Builder pattern and Builder
pattern
*
* Example 1:
*      UserLoginDTO userLoginDTO =
UserBuilder.anUser().buildUserLoginDTO();
*      This returns UserLoginDTO with email = "user@educapi.com" and
password = "testpassword";
*
* Example 2:
*      UserLoginDTO userLoginDTO =
UserBuilder.anUser().withEmail("foo@bar.com").buildUserLoginDTO();
*      This returns UserLoginDTO with email = "foo@bar.com" and password =
"testpassword";
*
* @author Enos Teteo
*/
public class UserBuilder {

   private Long id = null;
   private String name = "User";
   private String email = "user@educapi.com";
   private String password = "testpassword";

   /**
    * Init a new UserBuilder with default values that can personalized with
others methods
    * Is similar to getInstance in the Builder pattern
    *
```

```
     * Note: id by default is null but you can change it using the function
.withId("put your id").
     * @return a new User Builder
     */
    public static UserBuilder anUser() {
        return new UserBuilder();
    }

    /**
     * Changes the id of this UserBuilder
     * id is an identification, and in class User is only
     *
     * @param id type Long; example: 1L
     * @return UserBuilder with id changed
     */
    public UserBuilder withId(Long id) {
        this.id = id;
        return this;
    }

    /**
     * Changes the email of this UserBuilder
     * email is any email with type String
     * But if you use to test, can use any string
     *
     * @param email type String; example: "foo@bar.com"
     * @return UserBuilder with email changed
     */
    public UserBuilder withEmail(String email) {
        this.email = email;
        return this;
    }

    /**
     * Changes the name of this UserBuilder
     * name is your name or your object name
     *
     * @param name type String; Example: "test name"
     * @return UserBuilder with name changed
     */
    public UserBuilder withName(String name) {
        this.name = name;
        return this;
    }

    /**
     * Changes the password of this UserBuilder
     * password is an secret key usually used in authentication in systems
```

```
     *
     * @param password type String; Example: "testpass"
     * @return UserBuilder with password changed
     */
    public UserBuilder withPassword(String password) {
        this.password = password;
        return this;
    }

    /**
     * Generate an Optional object containing an User using custom or
default data
     * Example 1:
     *      UserBuilder.anUser().buildOptionalUser();
     *      This returns an Optional<User> with default data
     *
     * Example 2:
     *      UserBuilder.anUser().withName("Optional
User").buildOptionalUser();
     *      This returns an Optional<User> with all data default, but with
name "Optional User"
     *
     * @return Optional<User>
     */
    public Optional<User> buildOptionalUser() { return Optional.of(new
User(this.id, this.name, this.email, this.password)); }

    /**
     * Generate an UserRegisterDTO object containing custom or default data
     * Example 1:
     *      UserBuilder.anUser().buildUserRegisterDTO();
     *      This returns an UserRegisterDTO with default data
     *
     * Example 2:
     *      UserBuilder.anUser().withName("User Register
DTO").buildUserRegisterDTO();
     *      This returns an UserRegisterDTO with all data default, but with
name "User Register DTO"
     *
     * @return UserRegisterDTO
     */
    public UserRegisterDTO buildUserRegisterDTO() {
        return new UserRegisterDTO(this.name, this.email, this.password);
    }

    /**
     * Generate an UserLoginDTO object containing custom or default data
     * Example 1:
```

```
     *      UserBuilder.anUser().buildUserLoginDTO();
     *      This returns an UserLoginDTO with default data
     *
     * Example 2:
     *      UserBuilder.anUser().withName("User Login
DTO").buildUserLoginDTO();
     *      This returns an UserLoginDTO with all data default, but with
name "User Login DTO"
     *
     * @return UserLoginDTO
     */
    public UserLoginDTO buildUserLoginDTO() {
        return new UserLoginDTO(this.email, this.password);
    }

    public User buildUser() { return new User(this.id, this.name,
this.email, this.password);
    }

    /**
     * Generate an UserDTO object containint custom or default data
     *
     * Example 1:
     *      UserBuilder.anUser().buildUserDTO();
     *      This returns an UserDTO with default data
     *
     * Example 2:
     *      UserBuilder.anUser().withName("User DTO").buildUserDTO();
     *      This returns an UserDTO with all data default, but with name
"User DTO"
     *
     * @return UserDTO
     */
    public UserDTO buildUserDTO() { return new UserDTO(buildUser()); }
}
```