UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MATHIAS ABREU TRAJANO

MOCKTESTS: UMA FERRAMENTA PARA MODELAGEM EFICIENTE DE TESTES ENVOLVENDO END-POINTS DE APLICAÇÕES BACK-END.

CAMPINA GRANDE - PB

2022

MATHIAS ABREU TRAJANO

MOCKTESTS: UMA FERRAMENTA PARA MODELAGEM EFICIENTE DE TESTES ENVOLVENDO END-POINTS DE APLICAÇÕES BACK-END.

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Professor Dr. Adalberto Cajueiro de Farias

CAMPINA GRANDE - PB

2022

MATHIAS ABREU TRAJANO

MOCKTESTS: UMA FERRAMENTA PARA MODELAGEM EFICIENTE DE TESTES ENVOLVENDO END-POINTS DE APLICAÇÕES BACK-END.

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Adalberto Cajueiro de Farias

Orientador – UASC/CEEI/UFCG

Professora Dra. Melina Mongiovi Brito Lira

Examinador – UASC/CEEI/UFCG

Francisco Vilar Brasileiro

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 02 de Setembro de 2022.

CAMPINA GRANDE - PB

RESUMO

Objetos sintéticos são utilizados no desenvolvimento de software para simular o comportamento de objetos reais de forma controlada. Durante o desenvolvimento de servidores back-end, existe uma constante necessidade de modelar testes com objetos simulados para garantir o funcionamento adequado de funcionalidades dos mesmos. Todavia, essa tarefa acaba por se tornar repetitiva e complicada conforme a aplicação cresce. Ademais, testes que envolvam endpoints de aplicações se mostram mais complexos por ter necessidade de modelar requisições completas do tipo HTTP. A biblioteca MockTests tem por finalidades principais promover uma elaboração de testes sem a necessidade de inserção repetitiva de alguns componentes a cada caso de teste, simplificando assim diversos aspectos e solucionando empecilhos que podem surgir durante o desenvolvimento. Este presente trabalho tem por finalidade relatar as etapas do desenvolvimento da biblioteca MockTests, concebida para solucionar percalços durante a criação de testes envolvendo endpoints de aplicações. Os resultados obtidos demonstraram como a ferramenta se adequou ao ser usada em testes de uma aplicação backend real, e como a mesma solucionou diversos problemas encontrados na logística de construção dos respectivos testes.

MockTests: Uma ferramenta para modelagem eficiente de testes envolvendo end-points de aplicações Back-end.

Mathias Abreu Trajano
mathias.trajano@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

Adalberto Cajueiro de Farias
adalberto@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

RESUMO

Objetos sintéticos são utilizados no desenvolvimento de software para simular o comportamento de objetos reais de forma controlada. Durante o desenvolvimento de servidores back-end, existe uma constante necessidade de modelar testes com objetos simulados para garantir o funcionamento adequado de funcionalidades dos mesmos. Todavia, essa tarefa acaba por se tornar repetitiva e complicada conforme a aplicação cresce. Ademais, testes que envolvam endpoints de aplicações se mostram mais complexos por ter necessidade de modelar requisições completas do tipo HTTP. A biblioteca MockTests tem por finalidades principais promover uma elaboração de testes sem a necessidade de inserção repetitiva de alguns componentes a cada caso de teste, simplificando assim diversos aspectos e solucionando empecilhos que podem surgir durante o desenvolvimento. Este presente trabalho tem por finalidade relatar as etapas do desenvolvimento da biblioteca MockTests, concebida para solucionar percalços durante a criação de testes envolvendo endpoints de aplicações. Os resultados obtidos demonstraram como a ferramenta se adequou ao ser usada em testes de uma aplicação backend real, e como a mesma solucionou diversos problemas encontrados na logística de construção dos respectivos testes.

REPOSITÓRIOS

https://github.com/api-mocktests/mocktests
https://jitpack.io/#api-mocktests/mocktests/1.7.5

Palavras-chave

Testes; Aplicações back-end; Junit; MockMVC; Testes de Integração.

1. INTRODUÇÃO

Aplicações back-end são de suma importância para o funcionamento de grande parte das aplicações hospedadas na internet. Fluxo de acesso, controle de dados, requisitos de segurança e manutenção de funcionalidades, são alguns dos aspectos em que se encontram sob responsabilidade das mesmas.

No desenvolvimento de software, uma das etapas mais importantes é a validação do código por testes. Foge a norma disponibilizar softwares que não foram expostos a nenhum tipo de testagem. Conforme uma aplicação tem funcionalidades adicionadas ao seu código, modelar testes se torna mais complexo e custoso, dentre outras causas, devido especialmente à expansão do escopo. A abordagem mais utilizada por desenvolvedores é a construção de testes alinhada à produção de código, visando prevenção ao acúmulo de escopo a ser validado.

Em linguagens de programação amplamente utilizadas, existem bibliotecas e frameworks elaboradas com foco em suprir necessidades no tocante aos testes. Na linguagem Java, amplamente utilizada para desenvolver aplicações back-end, existe o JUnit, um framework open source especializado para geração de testes unitários e automatizados. Testes de unidade, ou testes unitários como são conhecidos, são testes criados com foco em validar código de determinadas funcionalidades separadas das demais. Durante o desenvolvimento de aplicações, tradicionalmente os testes unitários são utilizados como ferramenta predominante na verificação de código. Contudo, e em se tratando de aplicações back-end, esse tipo de testagem sem inclusão de formas para simular o fluxo de dados de entrada e saída advindo de requisições pode ser considerado uma forma incompleta, senão incoerente, de verificar como uma aplicação back-end deve reagir em contato com seu front-end[1].

[1] Aplicação relacionada a interface gráfica do projeto, onde o usuário irá interagir diretamente, seja em softwares, sites ou aplicativos.

2. MOTIVAÇÃO

No tocante a aplicações back-end, é comum que grande parte das operações dispostas necessitem ser manipuladas apenas por usuários que tenham os devidos privilégios e estejam devidamente identificados. Autenticar um usuário é confirmar que ele seja realmente quem se diz ser, enquanto que autorizá-lo é ter a certeza de que o mesmo possa acessar tal recurso requerido.

Para acessar uma determinada rota protegida de uma aplicação, comumente é necessário que o usuário envie junto a requisição, um conjunto de informações que representem seus privilégios para tal acesso. Em caso de um envio inexistente, ou com informações mal formatadas, é certo que o acesso seja negado. Bearer Token[1] e OAuth2.0[2] são exemplos dos vários tipos de sistemas de autenticação que podem ser implementados em aplicações, cada um com peculiaridades distintas, porém com objetivos em comum: garantir acesso a usuários devidamente registrados.

Ao adentrar no desenvolvimento de testes para aplicações back-end, especialmente aqueles que englobam a camada de controle da aplicação, o desenvolvedor se encontra na necessidade de formular lógicas para obter informações de login e/ou acesso válidas para cada teste que vá criar, com o intuito de verificar rotas específicas da aplicação. Grandes frameworks e bibliotecas geralmente não dão suporte para esses aspectos, deixando a obrigação dessas validações para os desenvolvedores. Como consequência direta, fica a critério dos mesmos encontrar alguma maneira de resolver, mesmo que de forma ineficiente, o problema descrito acima.

Outros fatores relacionados à criação de testes envolvem a legibilidade e necessidade de escrita excessiva de código, como também a necessidade do uso de tecnologias auxiliares para complementar funcionalidades. Os pontos citados acima constituem os principais fatores que impactam negativamente o desenvolvimento de código funcional e prático para testes.

A proposição da ferramenta MockTests incorpora um agrupamento de diversas tecnologias e metodologias para disponibilizar formas eficientes e práticas de contornar tais problemas. Sendo uma ferramenta restrita a ser utilizada dentro do escopo de desenvolvimento nas tecnologias descritas na seção 3, especificamente nas subseções 3.1 e 3.2 deste presente trabalho. Essa ferramenta foi gerada utilizando como base o conjunto de tecnologias descritas na seção 3. A maior parte destas tecnologias são utilizadas como uma solução preexistente para desenvolvedores que criam testes para aplicações back-end. Elas atuam em aspectos diversos na geração e funcionamento de testes, e cada uma delas necessita ser configurada de forma manual pelo desenvolvedor. O mockTests se torna uma ferramenta que utiliza esse conglomerado de tecnologias configurando-as automaticamente, sem que haja intervenção direta do desenvolvedor, cabendo ao mesmo só utilizar tais funcionalidades oferecidas.

3. TECNOLOGIAS UTILIZADAS

Nas próximas subseções serão aprofundados detalhes sobre as tecnologias utilizadas.

3.1 Java

Java é uma das linguagens de programação mais utilizadas para desenvolvimento de software pelo mundo, tendo foco no paradigma de programação estruturado e orientado a objetos, mas também com enlaces que permitem aos desenvolvedores implementarem código baseado em outros paradigmas como o funcional, o imperativo e o reflexivo. Como sendo um software de uso livre, o Java se tornou um padrão a ser utilizado para o desenvolvimento de aplicações back-ends. No presente estudo, foi utilizado como base a versão 11 do Java, por ser uma versão mais estável e com diversos recursos aproveitados.

3.2 Spring Boot

O Spring boot é um framework open source desenvolvido para a plataforma Java, que tem por funções principais simplificar configurações de projetos, como também o controle de APIs RESTful[3] que geralmente são empregadas em projetos de aplicações back-end. Além disso, ele se torna um framework útil por fornecer uma série de requisitos não funcionais que são pré configurados para o projeto, como por exemplo, acesso a bases de dados, segurança, servidor de aplicações embarcadas, etc. O Spring boot também permite injeção de dependências para a utilização de outras tecnologias que não sejam nativas do mesmo, se tornando assim, uma ferramenta extremamente versátil e de fácil configuração para desenvolvedores que optem por montar projetos e aplicações de forma fácil e prática.

---

[1] https://jwt.io/introduction

[2] https://oauth.net/2/

[3] https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api

3.3 JUnit

O JUnit é um framework open source que fornece suporte a criação de testes automatizados para a linguagem Java. Ele facilita a criação e manutenção do código que constituem a automação de testes, e possibilita também a apresentação dos resultados ao serem executados. Ao utilizar o JUnit, o desenvolvedor obtém a possibilidade de criar um modelo padrão para os testes, podendo fazer isso de forma automatizada.

3.4 MockMVC

MockMVC é uma biblioteca inclusa no framework Spring, que torna possível testar o processamento de solicitações na camada de controle de aplicações alvo. O processo envolve a montagem de uma requisição completa através dos métodos disponibilizados pelo mesmo. Ao submeter a requisição em um teste, o MockMVC se encarrega de instanciar versões temporárias das classes de controles da aplicação, e assim direciona os dados da requisição criada a respectiva rota indicada na requisição. Após a rota responder a requisição, seja com informações de sucesso ou erro, o MockMVC se encarrega de gerar uma resposta e em seguida apaga quaisquer instâncias temporárias que não sejam mais úteis.

3.5 ObjectMapper

Pertencente a biblioteca Jackson, o ObjectMapper é uma classe utilitária capaz de realizar conversões de classes POJO[1] em JSON[2], e vice-versa. Sendo o uso de JSONs o padrão mais utilizado na comunicação entre back-ends e front-ends, é necessário que sejam realizadas conversões ao precisar enviar, por exemplo, objetos qualquer no corpo de uma requisição em teste. Ao mesmo tempo, também se torna útil para converter objetos JSON, que advenham de requisições, em seus respectivos objetos representados. A utilização do ObjectMapper é uma das melhores opções no tocante a conversão entre POJO e JSON.

3.6 Reflections[3]

Reflexão é um recurso nativo da linguagem Java, ele permite que um programa que esteja sendo executado tenha acesso para examinar e obter informações sobre seu próprio código ou o código de outros programas, e assim, possa manipular propriedades internas dos mesmos.

---

[1] https://pt.wikipedia.org/wiki/Plain_Old_Java_Objects

[2] https://pt.wikipedia.org/wiki/JSON

[3] https://www.oracle.com/technical-resources/articles/java/javareflection

4. ARQUITETURA E PROJETO DA SOLUÇÃO

Esta seção irá detalhar a solução desenvolvida neste presente trabalho. Para isso, foram criando subseções, onde 4.1 explica o conceito da solução, 4.2 descreve detalhadamente cada componente da solução e 4.3 detalha todas as otimizações resultantes.

4.1 Visão Geral

O MockTests se constitui como um aglomerado de funcionalidades que tem por funções primordiais, permitir ao desenvolvedor criar testes de uma forma clara, legível, de fácil manutenção e sem a necessidade de inserção de componentes repetitivos entre cada caso de teste. Também visa disponibilizar diversos métodos para configurar e inserir informações de login e acesso para os testes que venham a necessitar de tais informações.

Entre tais métodos descritos acima, pode se citar a possibilidade de criar uma requisição específica que pode ser executada antes de cada teste e assim obter informações de login que serão automaticamente inseridas nos campos respectivos no teste antes que o mesmo seja executado. Também permite uma inserção geral de informações de login na declaração da classe de testes, e com isso, essas informações seriam inseridas de forma automática nos respectivos campos em cada caso de teste da classe. Por último, também permite uma inserção mais limpa e objetiva de informações em cada caso de teste de forma manual pelo desenvolvedor, caso seja do agrado dele. Fica a critério do desenvolvedor escolher quais funcionalidades utilizar em cada caso, tendo possibilidade também de utilizar várias delas em um mesmo caso.

Nas próximas seções serão descritas cada uma das funcionalidades implementadas pelo MockTests.

4.2 Componentes

Os componentes funcionais desenvolvidos no Mocktests atuam em uma lógica de conjunto com o intuito de serem uma ferramenta extremamente funcional e de simples usabilidade. Detalhes sobre os mesmos serão ressaltadas nas subseções a seguir.

4.2.1 Request

A classe Request foi desenvolvida com o intuito de servir como um modelo de requisição a ser utilizado por cada teste. Possui uma dinâmica simples para ser instanciada e ter seus atributos definidos. Para criar um novo objeto do tipo Request, só é necessário utilizar o comando *new* nativo da linguagem Java, conforme demonstrado abaixo:

Figura 01: Instanciando um objeto Request.

Uma requisição necessita de diversos tipos de campos com informações para servir ao seu propósito, diante disso, a classe Request recebeu atributos relacionados a cada uma dessas informações importantes.

4.2.1.1 Method

Se refere ao tipo de operação HTTP ao qual a requisição representa. Pode ser dos cinco tipos de operações: GET, POST, PUT, PATCH e DELETE.

4.2.1.2 Url

Representa o endereço da rota do servidor ao qual a requisição deve ser enviada.

4.2.1.3 PathParams

Parâmetros de caminho são partes variáveis pertencentes a um caminho Url. Eles podem ser utilizados como forma de apontar para um recurso específico dentro de uma coleção. Uma Url pode conter vários parâmetros de caminho.

4.2.1.4 Header

Representa os cabeçalhos HTTP, que permitem troca de informações adicionais nas requisições ou respostas HTTP. São compostos por uma tupla de chave e valores, e comumente utilizados para enviar informações de acesso.

4.2.1.5 Params

Os parâmetros de consulta são normalmente os mais utilizados em requisições. Eles aparecem no final da Url, e são compostos no formato, nome=valores.

4.2.1.6 ContentType

É um cabeçalho de representação usado para indicar o tipo de mídia a ser enviado nos recursos da requisição.

4.2.1.7 Body

Corresponde ao corpo da requisição, contém todos os dados associados à mesma.

Para configurar um objeto Request, não existem campos obrigatórios, o mesmo pode ser configurado apenas com os dados necessários para tal. Existem duas formas para adicionar tais campos: junto a instanciação do objeto ou de forma complementar após o mesmo ser instanciado.

Figura 02: Demonstração das formas possíveis para adicionar campos em um objeto Request.

4.2.2 Anotação @AutoConfigureRequest

A anotação @AutoConfigureRequest foi projetada para ser inserida na declaração da classe que contém os casos de testes. Ela possui por objetivos permitir que o desenvolvedor insira campos que sejam repetitivos para alguns ou todos os casos de teste da classe. Dessa forma, esses campos seriam automaticamente inseridos em cada requisição simulada com a classe Request que esteja sendo utilizada em algum teste.

Diante do escopo reduzido deste trabalho, somente dois campos foram incluídos como aptos a serem adicionados nesta anotação: o campo de ContentType e o campo Header.

O campo de ContentType é representado na anotação pelo campo mediaType, que recebe como parâmetro uma String, por exemplo “application/json”, que informa qual tipo de mídia está sendo enviada pela requisição. Caso uma requisição tenha sido estruturada sem a definição de um campo ContentType válido, e um mediaType tenha sido inserido na anotação, o campo ContentType da requisição será automaticamente definido com o valor do mediaType. Para o caso de uma requisição que tenha o campo

ContentType configurado, pela lógica de automatização implementada pela ferramenta, não haverá substituição pelo mediaType presente na anotação.

O campo de Header representado na anotação não depende diretamente que a anotação @AuthenticatedTest, descrita com detalhes na subseção 4.2.4, esteja vinculada nos casos de testes que necessitem de informações de acesso para a rota verificada. O campo pode ser preenchido opcionalmente, e de duas formas distintas a critério do desenvolvedor.

Na primeira forma de preenchimento, o desenvolvedor pode inserir todas as informações de acesso no campo de header, e com isso, a lógica implementada pela ferramenta irá verificar cada teste que foi anotado com a anotação @AuthenticatedTest, e que esteja sem definição de campo header, e irá realizar o processo de adicionar o campo de forma automática.

Figura 03: Demonstração de preenchimento da anotação @AutoConfigureRequest com um header totalmente preenchido.

Na segunda forma de preenchimento, o desenvolvedor não possui todas as informações de login necessárias, como por exemplo, a falta de um token de acesso. Como forma de diminuir ainda a necessidade de escrita excessiva de dados nos casos de testes, o campo header da anotação permite que somente os cabeçalhos dos header sejam inseridos na anotação, e que o token de acesso possa ser adicionado manualmente pelo desenvolvedor em cada caso de testes. Nessa forma descrita, há necessidade de que o teste que dependa de informações da anotação @AutoConfigureRequest esteja anotado com a anotação @AuthenticatedTest.

Figura 04: Demonstração de preenchimento da anotação @AutoConfigureRequest com um header parcialmente preenchido.

4.2.3 Anotação @Authenticate

A anotação @Authenticate foi elaborada com o intuito de servir como uma requisição que ao ser executada, forneça informações de login e acesso completas para casos de testes que necessitem de tais informações. Ela somente se torna funcional quando é inserida em uma declaração de objeto do tipo Request, devendo ser configurada dentro do escopo de código da classe de testes. Nesse caso, o objeto que é anotado por ela é processado, executado como uma requisição independente, e caso seu resultado tenha credenciais de acesso válidas, essas são inseridas em cada requisição presente nos casos de teste que estejam anotados com a anotação @AuthenticatedTest.

Figura 05: Demonstração de uma requisição de login anotada com @Authenticate.

4.2.4 Anotação @AuthenticatedTest

A anotação @AuthenticatedTest foi desenvolvida com o intuito de ser colocada especificamente nos casos de testes que contenham requisições que precisem obter informações

de acesso para obter um bom funcionamento dos mesmos. O fluxo de execução quando um teste recebe essa anotação é buscar pelas outras duas anotações perante toda a classe de testes, e caso encontre, obtenha as informações tentando primeiramente pela requisição de login anotada por @Auhenticate, e caso não consiga informações por ela, verifica se a anotação @AutoConfigureRequest possui informações completas de acesso para serem incrementadas nas requisições teste.

Um segundo fluxo de execução alternativo ocorre quando um teste anotado com @AuthenticatedTest possui uma requisição que teve seu campo header preenchido apenas com um token de acesso, mas sem os devidos cabeçalhos de identificação. Nesse caso, o fluxo de execução busca diretamente pela anotação de classe @AutoConfigureRequest, e caso esteja descrito no campo header da mesma, informações de cabeçalho, essas informações são concatenadas automaticamente no campo header da requisição junto com o token de acesso que já continha no referido campo.

```
@AutoConfigureRequest(
        mediaType = "application/json",
        header = {"Authorization", "Bearer"}
)
public class ExampleTests {
    @Test
    @AuthenticatedTest
    public void test01() {
      new Request().method(Method.POST)
                   .url("/api/add")
                   .header(token);
    }
```

Figura 06: Demonstração de interação entre as anotações @AutoConfigureRequest e @AuthenticatedTest ao concatenar dados no campo header da requisição.

4.2.5 Classe MockTest

A classe de nome MockTest é a responsável por reunir todas as operações utilitárias responsáveis pelo bom funcionamento da ferramenta, como também é responsável pela execução de todos os fluxos de automatizações que foram descritos nas subseções 4.3 acima relacionadas às anotações implementadas. Além disso, a classe MockTest implementa a função *"performTest"*, que recebe como parâmetro de entrada uma requisição do tipo Request, e é responsável por realizar todas as conversões e automatizações necessárias, enviar as requisições e entregar os devidos resultados das mesmas.

```
@AutoConfigureRequest(
        mediaType = "application/json"
)
public class ExampleTests {

    @Autowired
    MockTest mockTest;

    @Test
    @AuthenticatedTest
    public void test01() {
      mockTest.performTest(new Request()
                           .method(Method.POST)
                           .url("/api/add"))
              .andExpect(status().is2xxSuccessful());;
}
```

Figura 07: Demonstração do uso da classe MockTests e de sua referida função principal "performTest".

4.3 Otimizações resultantes

Esta subseção irá detalhar sobre todas as otimizações resultantes da solução implementada.

4.3.1 Melhoria de Legibilidade

Ao realizar um comparativo entre os casos de testes criados utilizando o MockTests, e casos que foram criados sem a ferramenta, ficou nítido que os casos que utilizaram a ferramenta ficaram mais legíveis e de fácil entendimento e manutenção em relação aos demais. No modelo anterior de criação de testes, alguns campos, como por exemplo, os campos que especificam os métodos http, a url da rota de destino e os parâmetros que complementam a url necessitavam se manter aninhados, o que poderia dificultar entendimento e manutenção casos os dados forem extensos. Pelo MockTests, os respectivos campos podem ser adicionados de forma separada, facilitando assim todo o processo de criação, manutenção e melhoria da legibilidade.

4.3.2 Diminuição da repetição de código

Outro ponto importante a ser detalhado é a diminuição de código necessário para formular testes ao utilizar a ferramenta MockTest. Com as automatizações possibilitadas pela ferramenta, vários campos importantes para as requisições, como o campo contentType e os headers podem ser declarados apenas uma única vez, e assim serem utilizados em todos os casos de teste de uma classe. Essa automatização é possível ao utilizar as anotações @AutoConfigureRequest e @AuthenticatedTest, oriundas da ferramenta MockTest.

4.3.3 Tecnologias auxiliares englobadas

Para criar os testes direcionados a aplicação, o desenvolvedor necessita utilizar tecnologias auxiliares como o MockMVC e a classe ObjectMapper. O MockMVC pode ser difícil de ser devidamente configurado para acessar corretamente as camadas de controle da aplicação alvo. O uso do ObjectMapper pode se fazer necessário, já que se mostra uma das tecnologias de conversão de objetos em texto JSON, procedimento necessário para enviar objetos no corpo de requisições de teste.

A ferramenta MockTests engloba essas e outras ferramentas às configurando de forma automatizada, sem necessidade de qualquer intervenção do desenvolvedor. Ademais, a classe Request implementa o método de *"body"* para o corpo de requisições, onde o mesmo aceita qualquer tipo de objeto convertendo-o implicitamente para ser enviado na requisição teste.

4.3.4 Formas alternativas para obtenção de informações de acesso

Sendo um dos principais objetivos implementados pelo MockTests, essa funcionalidade permite que o desenvolvedor monte uma requisição de login para obter informações de acesso independentes de quaisquer testes, podendo assim ser útil para toda uma classe de casos de testes diferentes. A integração dessa funcionalidade depende exclusivamente do uso das anotações @Authenticate para identificar a requisição de login a ser executada, e @AuthenticatedTest que identifica que o teste necessita de informações de acesso válidas. Assim, o MockTests se encarrega de realizar ambas as operações de forma automática, não cabendo ao desenvolvedor implementar lógicas adicionais para conseguir credenciais de acesso para os testes.

4.3.5 Exemplos

Nesta subseção, serão apresentados exemplos de testes implementados de formas convencionais e testes sendo implementados com as funcionalidades do MockTests que foram descritas nas subseções 4.3.1, 4.3.2, 4.3.3 e 4.3.4.

```
@ParameterizedTest
public void test01(Object object) throws Exception {

  ObjectMapper objectMapper = new ObjectMapper();

  mockMvc.perform(post("api/objects")
                      .header("Authorization", "Bearer " + userLogin.token)
                      .contentType("application/json")
                      .content(objectMapper.writeValueAsString(object)))
              .andExpect(status().is2xxSuccessful())
}
```

Figura 08: Caso de teste implementado sem nenhuma funcionalidade do MockTests.

```
@ParameterizedTest
@AuthenticatedTest
public void test01(Object object) throws Exception {

  mockTest.performRequest(new Request().method(Method.POST)
                  .url("api/objects").body(object))
                .andExpect(status().is2xxSuccessful())

}
```

Figura 09: Mesmo caso de teste da figura 08 sendo implementado com as funcionalidades do MockTests.

```
@Authenticate
Request request = new
Request().method(Method.POST).url("api/login")
         .body(new User("email","password"));
```

Figura 10: Requisição de login que complementa automaticamente o campo header do caso de teste representado na figura 09.

Como o teste representado na figura 09 foi anotado com a anotação @AuthenticatedTest, o fluxo de execução do MockTest que foi utilizado no caso de teste, vai verificar que não existe um campo header preenchido no próprio caso de teste, e assim vai buscar pela anotação @Authenticate. Caso encontre, o objeto Request que foi anotado será executado e o seu resultado será verificado e terá as informações retornadas convertidas e inseridas na requisição do caso de teste representado na figura 09.

5. AVALIAÇÃO E RESULTADOS

Esta seção irá detalhar sobre o processo de avaliação que foi submetido a biblioteca MockTests e com isso quais resultados foram obtidos.

5.1 Estudo de Caso

Para demonstrar que as funcionalidades do MockTests teriam as devidas propriedades propostas durante sua etapa de desenvolvimento, a biblioteca foi introduzida em uma aplicação backend real que tinha uma fatia considerável de testes que suportassem tal ferramenta. Para tal, foi escolhido o backend da aplicação Questões.PRO, desenvolvida por alunos do Curso de Bacharelado em Ciência da Computação pela Universidade Federal de Campina Grande. Na aplicação Questões.PRO, foram constatadas 9 classes contendo testes de integração voltados para suas funcionalidades. Nestas classes, havia um total de 193 casos de testes que utilizavam as tecnologias descritas na seção 3, subseções 3.3, 3.4 e 3.5. Ademais, foram encontradas lógicas adicionais que eram utilizadas para obter informações de login e acesso para serem utilizadas durante a execução de tais testes.

5.2 Conversão de tecnologias nos testes

Após análise dos testes e entendimento de todos os aspectos da lógica de funcionamento dos mesmos, foi iniciado o processo de inclusão e substituição das tecnologias anteriores pelo MockTests. Para tal, foi criada uma branch específica no repositório de código da aplicação, de nome *"sprint-6-tests"*. O processo de conversão dos testes não encontrou casos em que a tecnologia não pudesse ser utilizada.

5.3 Exemplos

Essa subseção foi dedicada para demonstrar exemplos de testes da aplicação Questões.PRO antes e depois do uso do MockTests.

```
@ParameterizedTest
@MethodSource("classroomCases")
@DisplayName("Tests creation of a valid classroom")
public void endpointWhenSavingClassroom(Classroom classroom) {

    mockMvc.perform(post(CLASSROOMS_ENDPOINT)
                    .header("Authorization", "Bearer " + teacherLogin.token)
                    .contentType("application/json")
                    .content(objectMapper.writeValueAsString(classroom)))
            .andExpect(status().is2xxSuccessful())
}
```

Figura 11: Primeiro exemplo de teste da aplicação Questões.PRO sem utilizar a ferramenta MockTests.

```
@ParameterizedTest
@MethodSource("classroomCases")
@DisplayName("Tests creation of a valid classroom")
@AuthenticatedTest
public void endpointWhenSavingClassroom(Classroom classroom) {

    mockTest.performRequest(new Request().method(Method.POST)
                    .url(CLASSROOMS_ENDPOINT).body(classroom))
            .andExpect(status().is2xxSuccessful()
}
```

Figura 12: Mesmo caso de teste da figura 10, após automatizações oferecidas pela ferramenta MockTests.

```
@Test
@DisplayName("Tests getting an exam answer by id that is inexistent")
public void endPointWhenGettingAnswerById() throws Exception {
    mockMvc.perform(get(EXAM_ANSWERS_BY_CODE_ENDPOINT, 1L)
                    .header("Authorization", "Bearer " + studentLogin.token))
            .andExpect(status().isNotFound());
}
```

Figura 13: Segundo exemplo de teste da aplicação Questões.PRO sem utilizar a ferramenta MockTests.

```
@Test
@DisplayName("Tests getting an exam answer by id that is inexistent")
@AuthenticatedTest
public void endPointWhenGettingAnswerById() throws Exception {
    mockTest.performRequest(new Request().method(Method.GET)
                    .url(EXAM_ANSWERS_BY_CODE_ENDPOINT).pathParams(1L))
            .andExpect(status().isNotFound());
}
```

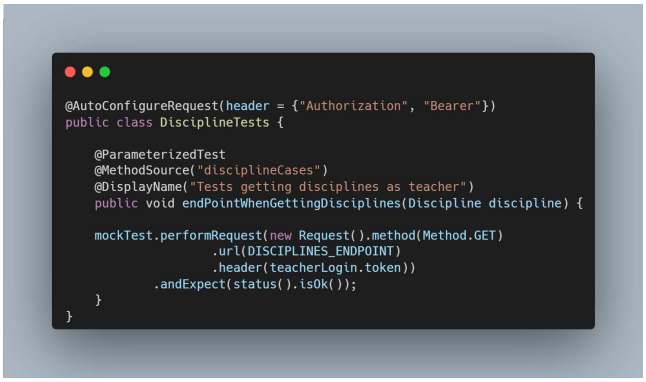
Figura 14: Mesmo caso de teste da figura 12, após automatizações oferecidas pela ferramenta MockTests.

5.4 Resultados

Ao utilizar as funcionalidades da ferramenta MockTests, houve 100% de aproveitamento ao converter todos os 193 testes da aplicação Questões.PRO. As classes Request e MockTest puderam ser utilizadas em todos os 193 testes, totalizando 100% de aproveitamento. A anotação @AutoConfigureRequest pode ser utilizada em todas as 9 classes de teste encontradas, sendo úteis para todos os 193 testes, totalizando assim também 100% de aproveitamento desta anotação.

Dos 193 testes da aplicação, 177 necessitam de informações de acesso válido, a anotação @AuthenticatedTest pôde ser utilizada em 102 desses testes, totalizando 57,62% dos testes, substituindo assim a

lógica que era utilizada para que os mesmos obtivessem tais credenciais de acesso. Alguns dos testes não puderam utilizar a anotação, pois a classe de testes não operava apenas com testes que envolvessem um único tipo de usuários da aplicação. Mesmo diante disso, a anotação @AutoConfigureRequest pôde ser utilizada para padronizar os cabeçalhos de header dos testes, servindo também como uma otimização parcial dos campos.



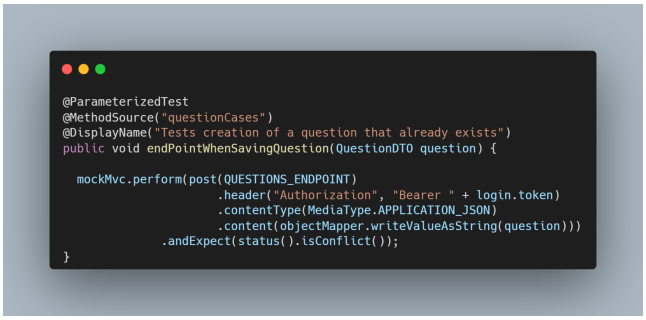
```
@AutoConfigureRequest(header = {"Authorization", "Bearer"})
public class DisciplineTests {

    @ParameterizedTest
    @MethodSource("disciplineCases")
    @DisplayName("Tests getting disciplines as teacher")
    public void endPointWhenGettingDisciplines(Discipline discipline) {

    mockTest.performRequest(new Request().method(Method.GET)
                    .url(DISCIPLINES_ENDPOINT)
                    .header(teacherLogin.token))
            .andExpect(status().isOk());
    }
}
```

Figura 15: Demonstração da otimização parcial do campo header da requisição com a anotação @AutoConfigureRequest.

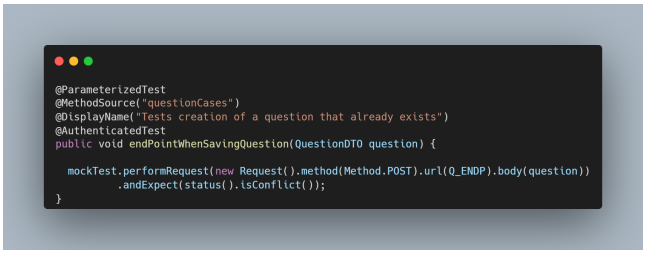
Por fim, a anotação @Authenticate foi utilizada para substituir as lógicas de obtenção de informações de login em 7 das 9 classes de teste, totalizando 77,7% de aproveitamento. Em um panorama geral, 79 testes, aproximadamente 40,93%, foram otimizados de forma a serem reescritos em apenas uma linha de código, conforme exemplos abaixo:



```
@ParameterizedTest
@MethodSource("questionCases")
@DisplayName("Tests creation of a question that already exists")
public void endPointWhenSavingQuestion(QuestionDTO question) {

  mockMvc.perform(post(QUESTIONS_ENDPOINT)
                    .header("Authorization", "Bearer " + login.token)
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(objectMapper.writeValueAsString(question)))
            .andExpect(status().isConflict());
}
```

Figura 16: Exemplo de caso de teste não utilizando a ferramenta MockTests para otimizações.



```
@ParameterizedTest
@MethodSource("questionCases")
@DisplayName("Tests creation of a question that already exists")
@AuthenticatedTest
public void endPointWhenSavingQuestion(QuestionDTO question) {

  mockTest.performRequest(new Request().method(Method.POST).url(Q_ENDP).body(question))
          .andExpect(status().isConflict());
}
```

Figura 17: Mesmo caso de teste da figura 16, reescrito utilizando automatizações da ferramenta MockTests.

Diante dos dados, a ferramenta MockTests se mostrou eficiente e flexível, podendo ser utilizada em diversos tipos de testes envolvendo requisições e se adaptando a diversas lógicas implementadas nas classes de testes, reduzindo e automatizando também partes do código.

6. EXPERIÊNCIAS

A ideia de desenvolvimento da solução surgiu mediante as necessidades encontradas no desenvolvimento de testes para uma aplicação back-end na qual participo enquanto integrante de um projeto de P&D (Pesquisa e Desenvolvimento) no SPLab (Laboratório de Práticas de Software) pertencente à UFCG. O processo de desenvolvimento da ferramenta MockTests possibilitou a mim interação com novas tecnologias de desenvolvimento, especialmente voltadas à reflexão de código fonte na linguagem Java. Desenvolver a ferramenta MockTests e disponibilizá-la tanto para o projeto P&D ao qual participo, como para a comunidade open-source mostrou o quão gratificante pode ser compartilhar conhecimento com outros desenvolvedores do ramo. Além disso, a contribuição dessa ferramenta para tal comunidade poderá ser de grande utilidade para diversos desenvolvedores pelo mundo.

7. TRABALHOS FUTUROS

A ferramenta MockTests encontra-se apenas em sua primeira versão. O objetivo é incrementá-la com mais funcionalidades que visam diminuir parte do trabalho repetitivo que o desenvolvedor venha desempenhar ao criar testes para aplicações backend. Num futuro próximo, a ferramenta MockTests poderá ser capaz de identificar e realizar leituras nas camadas de controle de aplicações back-end, e com isso, gerar testes de forma automática com base nos dados coletados em cada rota de uma aplicação back-end.

Agradecimentos

Ao meu orientador Adalberto Cajueiro, por ser guia e inspiração para o desenvolvimento deste presente trabalho. Aos meus familiares, especialmente da parte materna, que apesar de todos os obstáculos, nunca deixaram de acreditar no meu potencial e sempre estiveram ao meu lado. Aos meus pais, Edcarla e Adailton, que nunca desacreditaram dos meus sonhos, apesar do quão difíceis aparentavam ser. Aos meus avós, que por vezes distantes fisicamente, mas nunca sentimentalmente, estiveram comigo desde meu nascimento. Aos meus colegas de curso, que levarei para a vida, por todas as experiências, momentos de descontração e por todo o conhecimento que compartilhamos uns com os outros. Aos meus colegas do SPLab, que estiveram presentes nessa reta final, e prestaram grandes auxílios a mim. E por fim, a todos os meus amigos, que me acompanharam desde o começo, arrancando de mim risadas mesmo diante dos momentos mais difíceis. A todos, minha humilde gratidão.