Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Testify

Uma plataforma para aprender testes de software

Vinícius Guimarães Pereira

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman
Cossupervisor: MsC. João F. Daniel

São Paulo
2022

Resumo

Vinícius Guimarães Pereira. **Testify: *Uma plataforma para aprender testes de software***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Muitos programadores acabam desistindo de estudar sobre testes de software por conta da falta de conteúdo organizado na internet, fazendo com que hajam poucos engenheiros de software experientes na área. O objetivo deste trabalho é ajudar desenvolvedores que desejam aprender sobre testes de software. Faremos isto através de uma solução que os incetive a utilizarem livros, que seja acessível e fácil de se utilizar, que contenha um formato de estudo prático e, além de tudo, que seja aberta a contribuições de outros desenvolvedores. A solução encontrada foi desenvolver uma plataforma, chamada *Testify*, que busca alcançar os objetivos do trabalho. Ela possui todo seu conteúdo baseado no livro "Effective Software Testing: A developer's guide", provendo resumos dos capítulos do livro e com exercícios práticos para fixação de conhecimento. O *website* foi implementado utilizando tecnologias atuais. A arquitetura e a implementação foram feitas visando a extensibilidade. Outros desenvolvedores podem adicionar conteúdo na plataforma sem a necessidade de implementar nada novo. Por fim, a estrutura do código implementado torna mais fácil a modificação e extensão, por seguir padrões atuais de design. A plataforma busca servir de solução para os problemas que os desenvolvedores enfrentam quando tentam estudar sobre testes de software. O MVP está funcional e atende aos requisitos mencionados, como o incentivo a leitura de livros, exercícios práticos e a abertura à extensibilidade. Próximos passos já estão mapeados, contando com um trabalho em conjunto com Maurício Aniche.

Palavras-chave: Testes de software. Aprendizado. Plataforma.

Abstract

Vinícius Guimarães Pereira. **Testify: *A platform to learn software testing***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Most programmers give up on styding software testing as it's difficult to find organized sources on the internet, causing few experienced software engineers in the area. The goal of this project is to help developers to learn software testing. We will do that through a solution that incentives them to use books, which will be accessible and easy to use, which will have a practical study format, and beyond everything that will be open to contributions. The found solution was to develop a platform called Testify, which seeks to achieve the goals of this project. It has all content based on the book "Effective Software Testing: A developer's guide.". Besides, it provides summaries of the book's chapters and contains practical exercises for knowledge fixation. The website has been built using the latest technologies. The architecture and implementation were done aiming for extensibility. Other developers can add content to the platform without having to implement anything new. Finally, the structure of the implemented code makes modification and extension easier, as it follows known design patterns. The platform seeks to serve as a solution for all the problems that developers have when trying to study software testing. The MVP is functional and meets the mentioned requirements, such as the incentive to book reading, practical exercises, and openness to extensibility. The next steps are mapped, including working together with Maurício Aniche.

Keywords: Software Testing. Learning. Platform.

Sumário

Capítulo 1

Introdução

1.1 Bob descobre o Testify

A figura 1.1 contém uma história em quadrinhos que conta o trajeto do personagem Bob em sua descoberta da plataforma *Testify*, a solução final deste trabalho.

De forma resumida a história começa com Bob passando dificuldades ao tentar aprender sobre testes de software pela internet, por não achar nenhum conteúdo que lhe agrade. Ao pedir ajuda a amigos, recebe respostas desmotivadoras, porém, quando busca auxílio de seu professor, este lhe indica o livro "Effective Software Testing", de Maurício Aniche, que ensina sobre os princípios básicos de teste de software e uma série de metodologias para o desenvolvimento de testes. Mesmo com o livro, Bob sente que apenas ler não é o suficiente para que tenha um aprendizado completo, portanto resolve buscar ajuda na internet novamente, dessa vez pesquisando por exercícios práticos, o que o leva a encontrar o *Testify*, uma plataforma em que o conteúdo é totalmente baseado no livro, contendo exercícios práticos e resumos para auxiliar o programador. Dessa forma, Bob consegue aprender sobre testes de software e concluir seus objetivos.

Como o *Testify* é enxergado como um produto, esta história em quadrinhos foi criada para funcionar como uma propaganda. Seguindo o padrão de jornada do herói ela mostra todas as utilidades do *Testify* e os problemas que a plataforma busca resolver.

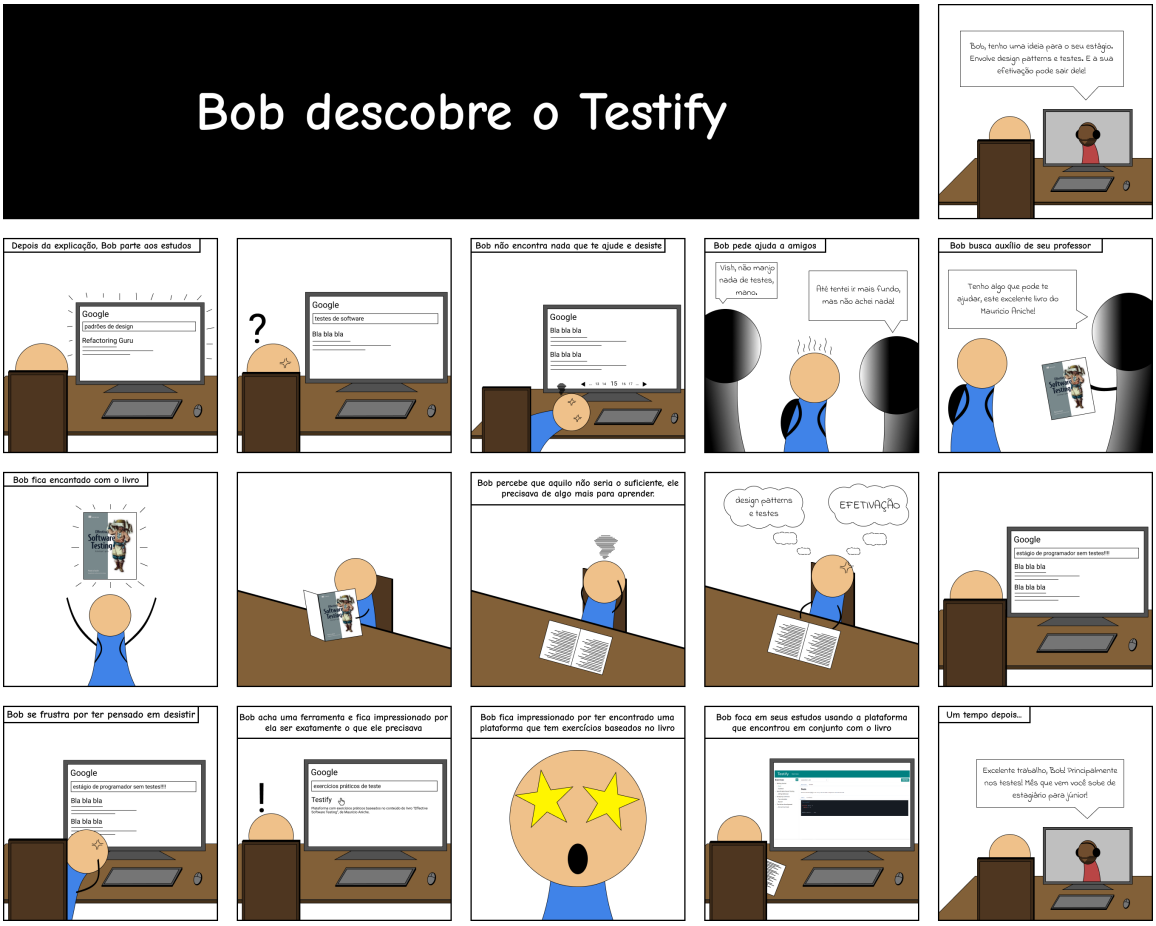

Figura 1.1: *História em quadrinhos do Testify*

1.2 Objetivo do trabalho

Um dos maiores desafios enfrentados por desenvolvedores quando buscam aprender algo novo é saber por onde devem começar. O primeiro passo mais comum é pesquisar por conteúdos na internet, com a busca por leitura tradicional deixada em segundo plano, seja por falta de recomendações de bons livros ou por dificuldade de se aprender sem prática. Quando se trata de testes de software, a situação é a mesma, o que há de diferente de outras áreas da programação é que há uma carência de material organizado na internet que guie os programadores sobre o assunto. Estes dois fatores fazem com que muitos desistam de aprender sobre testes e haja poucos desenvolvedores com entendimento sobre o assunto, mesmo hoje em dia sendo algo necessário para o bom funcionamento e manutenção de sistemas complexos.

O objetivo deste trabalho é ajudar desenvolvedores que desejam aprender sobre testes de software, tanto iniciantes quanto experientes, e aumentar o número de engenheiros experientes no assunto. Queremos incentivar os programadores a utilizarem livros, por conta da didática e do conhecimento aprofundado que possuem. Necessitamos de uma solução que seja acessível, que funcione mesmo para os que não podem adquirir livros, e fácil de se utilizar, para que os estudantes possam aprender sem esforço. Deve possuir

exercícios práticos, por ser algo comum e funcional para fixação de conhecimento. E por fim, na questão de implementação, deve ser extensível para contribuições de outros desenvolvedores.

1.3 Resultados alcançados

Os objetivos deste projeto foram alcançados através da implementação da plataforma *Testify*, que teve seu conteúdo construído com base no livro "Effective Software Testing: A developer's guide", de Maurício Aniche. O *Testify* possui seções baseadas nos capítulos da obra, com pequenos resumos dos conceitos explicados em cada um, e grupos de exercícios que são a parte prática da plataforma.

1.4 Estrutura do documento

No Capítulo 2 desta monografia entendemos sobre os assuntos que moldam a base de conhecimento necessária para o planejamento e implementação da plataforma *Testify*, tendo dois pontos principais, os Testes de Software e o Desenvolvimento Web. No Capítulo 3, entendemos sobre a solução implementada neste trabalho. São explicados seus objetivos, bases, requisitos e a plataforma, que é a solução proposta pelo projeto para alcançar os objetivos propostos. O Capítulo 4 contém explicações sobre a implementação e detalhes técnicos, portanto nos aprofundamos na arquitetura do projeto. Por fim, o capítulo 5 deste documento apresenta a conclusão do projeto, retomando as contribuições e indicando trabalhos futuros.

Capítulo 2

Base Teórica

2.1 Conceitos gerais

2.1.1 Injeção de Dependências

A injeção de dependências é uma estratégia utilizada em programação orientada a objetos. Seu objetivo é o desacoplamento de código, removendo a dependência de uma classe sobre a outra.

A Figura 2.1 contém um exemplo de dependência entre classes. Note que *ClassA* cria uma instância de *ClassB* diretamente em sua implementação. Caso haja necessidade de substituir *ClassB* por *ClassC* dentro do *method1*, é necessário modificação direta no código da *ClassA* pelo programador, o que sugere uma dependência entre as classes *ClassA* e *ClassB*.

A Figura 2.2 por sua vez mostra a estratégia de injeção de dependências sendo utilizada. Ambas as classes *ClassB* e *ClassC* implementam a mesma interface *InterfaceB*. A classe *ClassA* recebe uma instância desta interface diretamente em seu construtor, possibilitando a substituição da classe *ClassB* por qualquer outra que implemente a interface *InterfaceB*, sem que haja necessidade de alteração no código de *ClassA*.

A vantagem é muito utilizada por ajudar programadores a evitarem possíveis problemas gerados por alterações indevidas no código. Caso a classe *ClassA* seja de extrema importância para o sistema, é mais vantajoso evitar mudanças que não são necessárias. Além de promover maior facilidade em mudanças nesse tipo, já que utilizar injeção de dependências também promove a mudança dinâmica entre dependências (por exemplo, poderiam haver diversas implementações da interface *InterfaceB* que mudam de acordo com o *input* do usuário).

```
interface InterfaceB {
    // methods...
}

class ClassB implements InterfaceB {
    // methods...
}

class ClassC implements InterfaceB {
    // methods...
}

class ClassA {
    constructor() {}

    public method1() {
        const instanceB = new ClassB();
        // uses instanceB methods
    }
}
```

Figura 2.1: *Dependência entre classes*

```
interface InterfaceB {
    // methods...
}

class ClassB implements InterfaceB {
    // methods...
}

class ClassC implements InterfaceB {
    // methods...
}

class ClassA {
    private instanceInterfaceB: InterfaceB;

    constructor(instanceInterfaceB: InterfaceB) {
        this.instanceInterfaceB = instanceInterfaceB;
    }

    public method1() {
        // uses this.instanceInterfaceB methods
    }
}
```

Figura 2.2: *Injeção de dependência*

2.2 Testes de Software

Testes são ferramentas utilizadas para verificar a aderência do código aos requisitos funcionais e não-funcionais do sistema. Hoje em dia os desenvolvedores utilizam diversas ferramentas para executarem estes testes de forma automática, criando o que alguns chamam de Códigos Auto-testáveis (Fowler, 2014).

Já não se discute mais a importância de testes automatizados para sistemas de software. Não implementá-los significa que seu programa estará sob risco de falhas com a mínima alteração. Erros que podem causar problemas sérios a negócios ou até mesmo a sociedade dependendo do quão importante é este sistema.

Uma das principais vantagens de se utilizar testes automatizados é a prevenção de bugs. Qualquer mudança indesejada no comportamento do código é identificada durante a execução dos testes, e qualquer novo bug que seja encontrado no código faz com que um novo caso de teste seja implementado cobrindo este caso em específico. Além disso, também ajuda a manter a qualidade do código e a trazer mais confiança aos desenvolvedores para implementarem mudanças.

Hoje em dia existem diversos tipos de testes automatizados de software, cada um com seus próprios objetivos, vantagens e desvantagens. Aqui estão alguns exemplos de tipos de teste:

- Testes de unidade: Este geralmente é o primeiro tipo de teste que os desenvolvedores aprendem a implementar. O objetivo dele é testar uma unidade individual do sistema. A ideia deste tipo de teste automatizado é que apenas uma unidade deve ser testada por vez. Para que isto funcione, geralmente é aplicado o conceito de injeção de dependências. Uma outra forma é sobrepor a implementação de funções ou métodos que são chamados dentro daqueles que estão sendo testados - uma funcionalidade fornecida dentre as diversas ferramentas de testes unitários (JUnit, Jest, Mocha etc).
- Testes de integração: Os testes de integração, ao contrário dos testes de unidade, buscam testar a conexão de duas ou mais partes do sistema, para certificar-se de que a integração delas está funcionando corretamente.
- Testes de Ponta-a-Ponta (End-to-End): Diferente dos anteriores, testam o sistema por completo, com todos os componentes conectados. Utilizado para certificar-se de que o fluxo geral está funcionando conforme os requisitos, geralmente simulando um caso de uso para verificar se é possível concluí-lo sem erros. Verificando também se as chamadas aos servidores e a interação com outros sistemas estão sendo realizadas corretamente.
- Testes de aceitação: São testes realizados para verificar se o produto atende a todos os requisitos especificados. Neste caso, são muito utilizados ambientes que simulam o ambiente de produção.

Dada a quantidade de tipos de teste de software que existem, é muito questionado qual deve receber a maior atenção dos desenvolvedores. Em 2009 foi apresentado o conceito de Pirâmide de Testes (Cohn, 2009), uma pirâmide formada por 3 camadas: na base, testes de unidade; no meio, testes de serviço; e na ponta, testes de interface – a Figura 2.3 ilustra esse modelo. Em direção a base da pirâmide estão os tipos de teste com maior isolamento, isto é, em que o menor número de componentes do sistema é testado por vez, enquanto para a ponta estão aqueles com maior integração, em que mais de um componente do sistema é testado por vez. Outro conceito aplicado à pirâmide é que em direção à base estão os testes que devem ser executados de forma mais rápida, enquanto em direção à ponta estão os mais lentos.

Hoje em dia, porém, muitos acreditam que os nomes citados na pirâmide não representam todos os cenários. Em alguns casos o formato da pirâmide chega a mudar dependendo do contexto em que ela é aplicada. Um exemplo é a Pirâmide criada por Kent Dodds, apelidada de "O Troféu de Testes" (Vocke, 2018), que representa o retorno do investimento nos tipos de testes em se tratando da linguagem de programação JavaScript (Figura 2.4). Um tema interessante a ser retratado, pois mostra a importância dos testes automatizados para empresas, trazendo grande retorno financeiro quando aplicado, já que depois de desenvolvidos salvam muito tempo de desenvolvimento a longo prazo, pois evitam bugs e mantém a qualidade do código alta e constante.

Hoje, existem diversas ferramentas excelentes para desenvolvimento de testes automatizados, como o JUnit ou AssertJ. Apesar de existirem diversos programadores que têm excelência no uso destas ferramentas, a maioria não sabe escrever testes de forma efetiva e sistemática, nem mesmo quais casos devem ser testados em seu programa. Por este motivo existem técnicas e metodologias que auxiliam o desenvolvedor neste processo (Aniche,

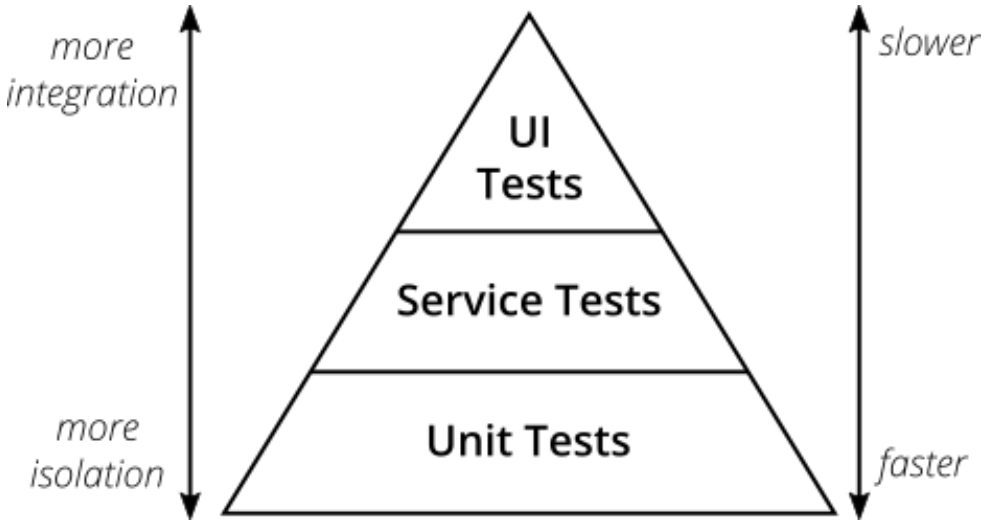

Figura 2.3: *Pirâmide de testes*

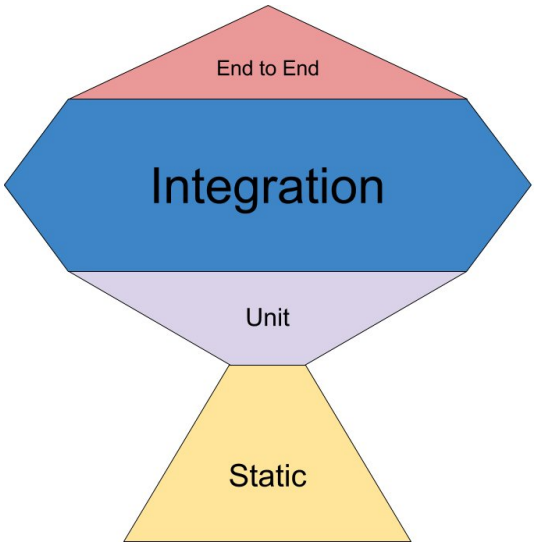

Figura 2.4: *Troféu de testes por Kent Dodds*

2021).

Utilizar técnicas de testes traz uma grande vantagem ao programador no momento de definir casos de teste. Esta tarefa pode ser difícil caso nenhuma estratégia seja utilizada, e definir apenas casos simples não ajuda o software a evitar bugs mais específicos.

Um exemplo de técnica de desenvolvimento de testes é o Specification-based Testing. É uma metodologia que utiliza os requisitos do software como base para os testes, tendo mais foco no desenvolvimento dos casos de teste do que na implementação em si, portanto neste caso os requisitos são mais importantes. Quando um software é implementado, esta técnica deveria ser a primeira utilizada. O processo desta metodologia pode ser dividido nos seguintes 7 passos, que não são necessariamente sequenciais, porém são iterativos (Aniche, 2021):

1. Entender os requisitos do programa.
2. Explorar o programa. Caso mais utilizado quando o código não foi escrito por aquele que o está testando. O programador executa diversos testes para entender o comportamento do programa.
3. Explorar as entradas e saídas possíveis do programa, e identificar partições.
4. Identificar os limites das partições.
5. Derivar testes baseados nos limites e partições encontrados.
6. Automatizar os testes.
7. Melhorar os testes utilizando criatividade e experiência.

Outro exemplo de metodologia de teste é o Test-Driven Development (TDD). É uma técnica de desenvolvimento de software em que os testes de unidade são escritos antes da implementação do código. O processo do TDD pode ser descrito de forma abstrata pelos seguintes passos (Aniche, 2021):

1. É criado um teste de unidade para uma nova funcionalidade;
2. O teste é executado falhando inicialmente, como esperado;
3. Implementa-se o mínimo necessário da funcionalidade para que o teste seja aceito;
4. Depois que o teste passa a ser aceito, deve refatorar-se o código implementado para livrar-se de duplicidade, caso necessário;
5. O processo recomeça para uma nova funcionalidade.

2.3 Desenvolvimento Web

O Desenvolvimento Web é uma área de ciência da computação que volta-se ao desenvolvimento de sistemas complexos de software para a Web, focados no serviço ao usuário.

Geralmente, o desenvolvimento Web divide-se entre o *front-end* e o *back-end*. O primeiro é voltado para a implementação do software que está em contato direto com o usuário,

ou seja, podemos dizer que renderiza a interface do sistema. O segundo é responsável por implementar os serviços que executam o funcionamento interno do sistema, por exemplo o armazenamento de informações, processamento de dados etc.

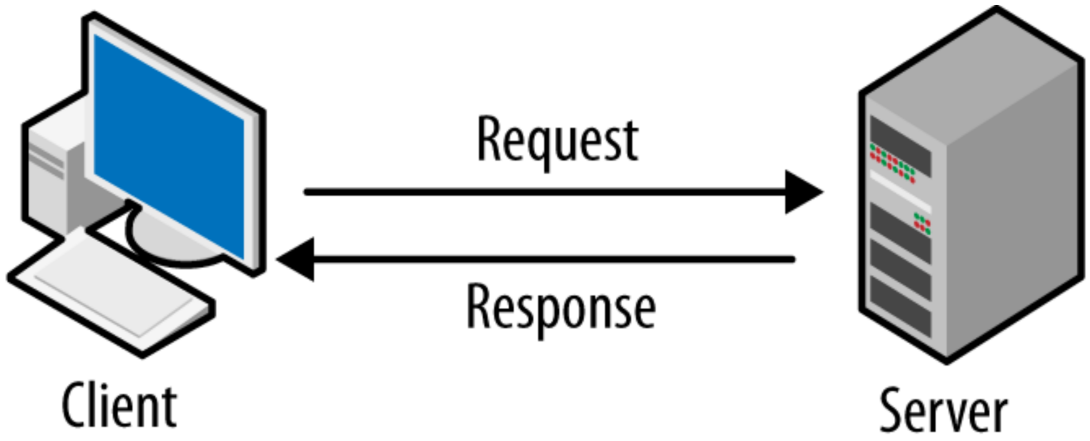

Figura 2.5: *Arquitetura cliente-servidor*

Na arquitetura cliente-servidor (Figura 2.5) o servidor provê alguma funcionalidade para o cliente, por exemplo o armazenamento de informações, processamento de dados etc, como citado anteriormente. O cliente pode representar tanto o front-end do sistema, que envia requisições ao servidor utilizando entradas vindas do usuário, ou mesmo outro servidor que utiliza os serviços do primeiro.

Ao desenvolver sistemas para a Web é muito comum e, hoje em dia, praticamente mandatório a utilização de ferramentas de suporte de desenvolvimento. Utilizar *frameworks* e bibliotecas que sejam bem documentadas e atualizadas facilita e agiliza muito o processo de desenvolvimento, pois o desenvolvedor não necessita desenvolver funcionalidades já muito comuns na web já providas por estas ferramentas. Além disso, muitas ferramentas determinam novos paradigmas de desenvolvimento que funcionam bem e são muito utilizados hoje em dia.

Um exemplo de biblioteca é o *React*. Seu intuito é o de fornecer ferramentas que ajudem o programador a desenvolver o *front-end* de sua aplicação através de componentes. Neste arcabouço, o programador define componentes de interface que podem ser reutilizados por todo o programa. Além disso, o *React* possui diversas funcionalidade de otimização de interface, permitindo renderizações rápidas mesmo de interfaces mais complexas, tornando a experiência do usuário final melhor.

Outra ferramenta utilizada neste meio é o *FastAPI*, um framework que facilita a construção de APIs REST. Seu foco é tornar o processo de desenvolvimento rápido, fácil e intuitivo, fazendo com que com poucas linhas de código seja possível criar um servidor. Além disso, este framework fornece diversas funcionalidades que facilitam a identificação de erros e a geração automática de documentações.

Para que a ferramenta se torne acessível para usuários ou outras ferramentas, é necessário que ela seja disponibilizada em algum serviço de implantação. Hoje, o mais comum é utilizar serviços em nuvem criados para este propósito. Geralmente os provedores de nuvem

oferecem ferramentas que funcionam como mecanismo de implantação de software. Um exemplo que pode ser citado são as instâncias *EC2*, providas pelo *AWS*, que são basicamente máquinas que podem ser acessadas virtualmente. Além disso, existem provedores que buscam facilitar a implantação de páginas estáticas, mais ligadas ao *front-end* de aplicações, como por exemplo o *Surge*, que permite isto através de um único comando.

Utilizar ferramentas deste tipo nos permite fornecer os serviços da nossa aplicação quase 100% do tempo, utilizando apenas serviços via Web. Diferente de tempos antigos onde apenas empresas conseguiam fornecer aplicações assim pois possuíam suas próprias máquinas que permaneciam ligadas o tempo todo.

Quando queremos criar uma parte isolada do nosso sistema para executar um conjunto de códigos, geralmente utilizamos a virtualização. Ela nos permite empacotar códigos juntos com todos os arquivos, ferramentas e componentes necessários, para que eles sejam executáveis de forma isolada dentro do sistema. Através dela o programador não necessita reescrever o código para implantá-lo em ambientes diferentes. O fato de tornar o ambiente de execução isolado do resto do sistema traz mais segurança, pois erros que venham a acontecer não prejudicam a máquina.

Os dois tipos de virtualização mais utilizados são as máquinas virtuais (VM) e os contêineres. Uma VM possui seu próprio sistema operacional, simulando uma máquina por completo via software, o que possibilita a execução de diversos serviços em uma única VM. O contêiner, por sua vez, possui uma única função, contendo somente os recursos necessários para executar ela. Por conta disso, o contêiner, comparado com uma máquina virtual, é extremamente leve, permitindo ao programador executar diversos programas de uma só vez, fornecendo escalabilidade ao sistema.

Atualmente existem ferramentas que facilitam a criação e administração de contêineres, sendo um grande exemplo o *Docker*. O *Docker* possui uma abstração que chamamos de imagem, que é basicamente a representação estática de um contêiner, com todas as suas dependências. Para criá-la é necessário implementar suas configurações, definindo parâmetros como as dependências, arquivos, variáveis de ambiente, portas etc. A partir destas configurações a ferramenta de conteinerização constrói a imagem. O contêiner, por sua vez, é executado a partir da imagem construída.

Capítulo 3

Plataforma Testify

O objetivo da plataforma *Testify* é ajudar desenvolvedores em seus aprendizados sobre testes de software e aumentar o número de engenheiros de software com capacidade de trabalhar nesta área. Com isso em mente, criamos uma solução que incentiva a leitura de livros, seja acessível e de fácil utilização, que possua uma forma prática de estudos e, além de tudo, seja aberta a contribuições de outros desenvolvedores, tanto na questão técnica quanto no conteúdo da plataforma.

3.1 Livro "Effective Software Testing: a developer's guide"

Um dos objetivos é incentivar programadores a lerem livros técnicos, pelos detalhes e aprofundamentos que eles tem, portanto é interessante que a solução possua como base um livro.Após uma breve leitura do livro "Effective Software Testing: A developer's guide", de Maurício Aniche, nota-se uma progressão de complexidade a cada capítulo. Isso é algo de extrema importância principalmente para aqueles que estão iniciando seu aprendizado em testes de software, pois estes sabem o que deve ser aprofundado antes de dar continuidade na leitura de capítulos mais complexos.

Esta progressão é um recurso que pode ser utilizado na solução, já que outro objetivo é que ela funcione como um guia de aprendizado para os programadores, e esta ordem de complexidade é importante para que isto funcione.

Além disso, o livro ensina metodologias de testes de software, que funcionam como ferramentas que fornecem passos de como implementar testes durante o desenvolvimento de sistemas. Algumas metodologias ensinadas no livro fornecem bases mais teóricas, como os *Specification-based Testing* e *Structural Testing*, que focam no planejamento de casos de teste. Outras focam mais nos passos utilizados, como o *Test-driven Development* (TDD).

3.2 Requisitos

Um problema que muitos sofrem ao aprender através de livros é a leitura não ser o suficiente para um aprendizado completo, sendo necessário algo prático. Portanto, a solução deverá fornecer exercícios com base nos capítulos do livro, por ser um formato muito utilizado para fixação de conhecimento e que funciona para muitas pessoas. Dessa forma os desenvolvedores podem utilizá-la em conjunto com o livro, servindo como uma forma de extensão.A ideia é que ao finalizarem um capítulo do livro, realizem os exercícios para verificarem o quanto entenderam do que foi lido.

Além disso, os exercícios devem possuir uma estrutura fácil de ser entendida e devem ser testáveis, para que os programadores possam verificar se suas respostas estão corretas.

Como um dos objetivos é que a solução funcione como um guia para os desenvolvedores, a organização do conteúdo é um fator importante. Pensando nisso, os exercícios devem ser separados em grupos ou seções, com cada um representando um capítulo do livro, para que os programadores saibam localizar os exercícios que devem realizar ao fim de um capítulo e sua experiência seja facilitada.

Mesmo que a solução seja baseada em um livro e que a ideia seja utilizá-la em conjunto com ele, muitos programadores não têm oportunidade de adquirir um. Como um dos objetivos é que a solução seja acessível a todos, ela deve funcionar sem um livro, fornecendo as informações necessárias para que os programadores possam aprender através dela ou ao menos entender por quais recursos devem buscar.

A solução deve ser extensível para que outros programadores que desejam contribuir para o aprendizado dos demais possam fazer isso com facilidade, tanto na questão da adição de exercícios quanto para a modificação do código fonte.

De forma a simplificar, os requisitos da plataforma são os seguintes:

- Conter exercícios práticos baseados no livro, com uma estrutura fácil de ser entendida, além de testáveis.
- Possuir um conteúdo organizado, com uma seção para cada capítulo do livro.
- Ser acessível mesmo para aqueles que não possuirem o livro.
- Ser extensível, para que outros desenvolvedores possam contribuir com a plataforma.

3.3 Solução

Para atender a todos os requisitos da solução proposta, uma plataforma, apelidada de *Testify*, foi implementada.

Uma plataforma hospedada em um website é um formato fácil de acesso aos desenvolvedores. Pode-se implementá-la utilizando ferramentas atuais e utilizando boas práticas de desenvolvimento, de forma a permitir futuras modificações e manutenções.

Além disso, implementando em um ambiente web nos permite encontrar e utilizar diversas bibliotecas para melhorar a experiência do usuário, como por exemplo o *React Textarea Code Editor*, que fornece um componente *React* que renderiza um editor de código com funcionalidades que facilitam a escrita de código, usada nos exercícios.

Ademais, podemos nos utilizar de ferramentas de mecanismos de pesquisa para tornar a plataforma mais visível aos usuários que pesquisam por recursos relacionados a testes de software.

O *Testify* é composto por seções, em que cada uma representa um capítulo do livro de Mauricio Aniche. A única exceção é a primeira seção, que tem como objetivo ser um exemplo de usabilidade da plataforma, e fornece uma breve explicação sobre a metodologia ensinada naquele capítulo.

Dessa forma, o usuário pode aprender o básico sobre ela e aprofundar o conceito no livro. Caso o usuário não tenha oportunidade de adquirir o livro, ainda assim a seção o ajuda a entender o que deve estudar para que possa resolver os exercícios.

Uma seção possui um ou mais exercícios para que o usuário possa praticar seus conhecimentos. Ao acessar uma seção, o usuário depara-se com um texto que explica de forma resumida a metodologia explicada no capítulo do livro representado por esta seção (Figura 3.1). Ao final, há uma lista de exercícios.

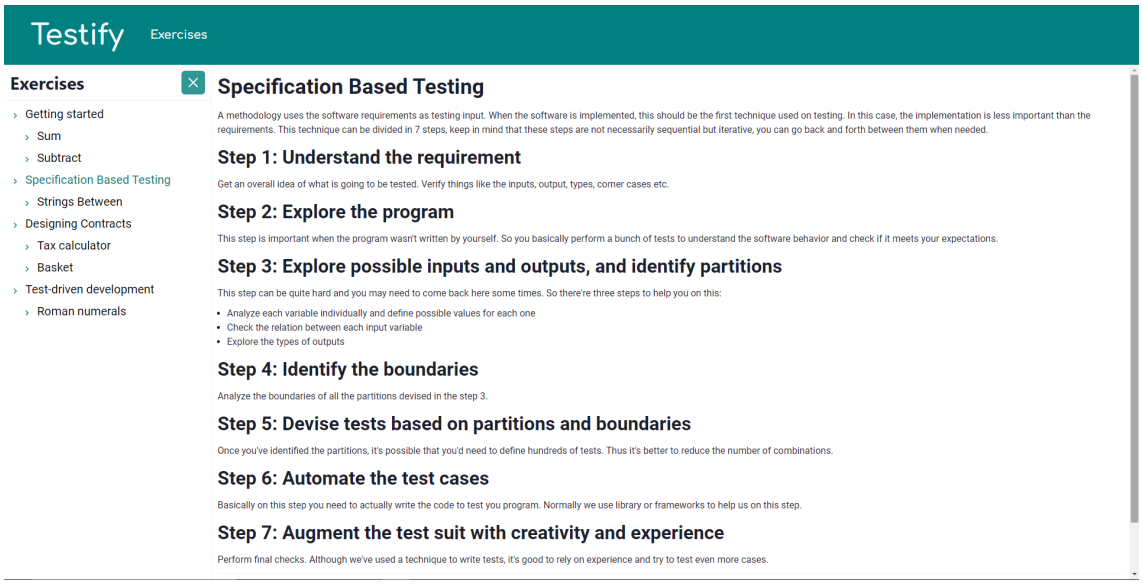

Figura 3.1: *Exemplo de seção do Testify*

Exercícios são onde os usuários podem pôr seus conhecimentos em prática. Cada um contém um problema que deve ser resolvido utilizando a metodologia da seção a que o exercício pertence.

Um exercício também pode ser auto-contido, isto é, não fazer parte de uma seção. Por este motivo, neste caso o próprio exercício representa um capítulo do livro e provê um pequeno resumo da metodologia explicada nele, além da descrição do próprio exercício.

Cada metodologia explicada no livro de Aniche tem sua forma de ser seguida. Apesar disso, todos os exercícios da plataforma seguem uma única estrutura, para que a experiência do usuário não seja afetada com a necessidade de aprender a lidar com múltiplas estruturas.

Os exercícios foram construídos utilizando testes de unidade e códigos que são testados por eles. Em alguns casos, o usuário deve modificar os arquivos de teste, em outros o arquivo de código. Além disso, há comentários nos arquivos que delimitam o espaço que é passível de modificação pelo usuário, para evitar que haja problemas durante a execução.

Por fim, para fazer com que apenas uma única estrutura funcionasse para todos os casos, foi adicionado um texto de solução para cada exercício. As soluções buscam detalhar como o exercício deve ser resolvido utilizando a metodologia da seção. Dessa forma, evitamos múltiplas estruturas e fornecemos uma resposta caso o usuário não consiga elaborar uma, melhorando sua experiência.

Os exercícios possuem uma série de funcionalidades e recursos que estão listados abaixo, numerados e identificados pelas Figuras 3.2, 3.3 e 3.4:

1. Um seletor de ambientes de teste. Dessa forma, o usuário pode resolver o exercício utilizando a linguagem e ferramenta de teste que deseja.
2. Funcionalidade para testar a solução implementada pelo usuário, que é executada ao clicar no botão “”Run Test”.
3. Abas que acessam tanto a descrição quanto a solução do exercício.
4. Descrição do que é esperado no exercício.
5. Solução que demonstra detalhadamente como resolver o problema utilizando a metodologia da seção.
6. Abas que representam os arquivos do exercício, que devem ser modificados para a resolução do problema.
7. Editor de código para modificação dos arquivos do exercício.
8. Comentários do código que demonstram quais regiões podem e/ou devem ser modificadas.
9. Uma aba específica dos arquivos que mostra os resultados dos testes do exercício, que abre após a execução estar finalizada.

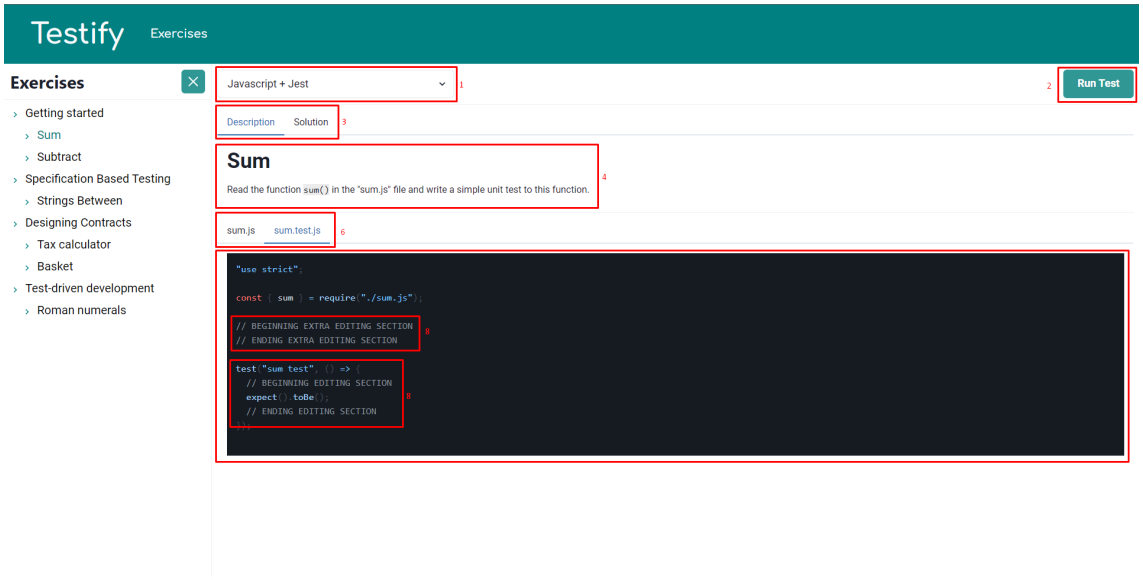

Figura 3.2: *Elementos de um exercício do Testify 1/3*

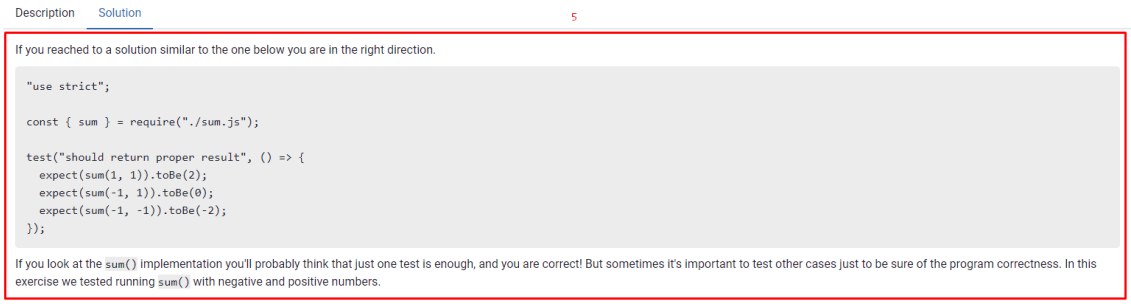

Figura 3.3: *Elementos de um exercício do Testify 2/3*

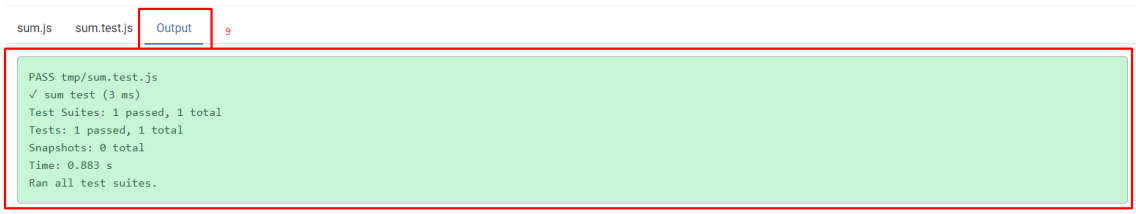

Figura 3.4: *Elementos de um exercício do Testify 3/3*

Capítulo 4

Implementação e detalhes técnicos

4.1 Visão Geral

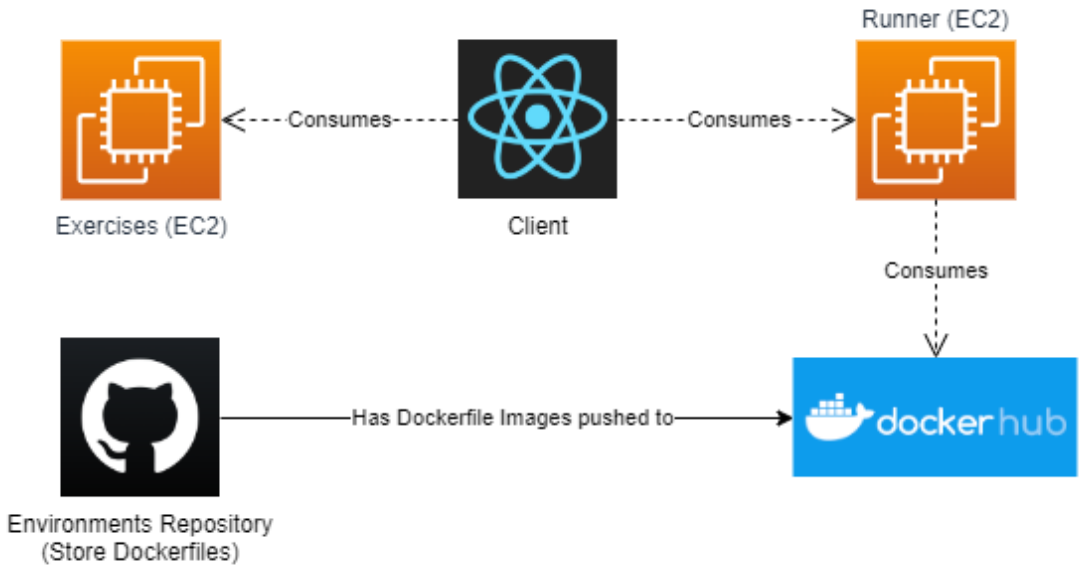

Figura 4.1: *Arquitetura do MVP da plataforma*

O MVP (*Minimum Viable Product*) do projeto contém três serviços principais responsáveis pela funcionalidade total da plataforma. A arquitetura inicial contém o *Client*, que consome outros dois serviços, o *Exercises* e o *Runner*.

Além disso, há também o repositório *Environments* que contém os ambientes de teste da plataforma. Os ambientes são determinados por *Dockerfiles*, portanto as imagens são salvas no *DockerHub*.

Vale mencionar que todas as explicações abaixo estão presentes na documentação da

plataforma e serão salvas no repositório de cada serviço.

4.2 *Client*

4.2.1 Propósito

Seu principal propósito é renderizar a interface da plataforma, portanto é responsável pela experiência final do usuário.

4.2.2 Ferramentas utilizadas

- *React*: responsável pela divisão da interface em componentes.
- *React Router*: permite a criação de múltiplas rotas mesmo em uma plataforma feita em uma única página.
- *Chackra UI*: arcabouço que fornece componentes com design pré-montados, mas que permite fácil customização.

4.3 *Runner*

4.3.1 Propósito

Este servidor é responsável por executar exercícios, contendo uma série de passos que se resumem em receber os arquivos de código, executá-los dentro de um contêiner *Docker* e enviar o output como resposta da requisição.

4.3.2 Implementação

Implementado em *Python* utilizando o framework *FastAPI*, que facilita a criação de servidores *REST*.

Há uma única rota, que é executada através do método *HTTP POST*, recebendo os seguintes parâmetros:

- O ID do ambiente de teste selecionado pelo usuário.
- O comando que deve ser utilizado para executar o exercício dentro do contêiner Docker.
- Uma lista de arquivos com nome e conteúdo, que no caso são os arquivos do exercício modificados pelo usuário.

E devolve como resposta:

- O output da execução dos testes do exercício.
- Uma variável booleana que determina se o teste passou ou não.

4.3.3 Fluxo de execução de exercícios

1. O nome da imagem Docker é obtido através do ID do ambiente de teste recebido.
2. Um verificador de teste também é obtido da mesma forma.
3. Os arquivos de teste recebidos na requisição são criados localmente no servidor em um diretório temporário.
4. É feito o download da imagem Docker pelo DockerHub,utilizando o nome obtido no primeiro passo.
5. O contêiner Docker é construído utilizando uma série de parâmetros que:
   (a) Criam um volume dentro do contêiner conectado com o diretório temporário criado no passo 3. Dessa forma os arquivos são acessíveis dentro do contêiner.
   (b) Executam o exercício utilizando o comando vindo da requisição.
   (c) Fornecem segurança ao servidor, impedindo códigos maliciosos de serem executados dentro do contêiner.
6. O output do contêiner é salvo.
7. O contêiner é interrompido e removido.
8. O verificador de testes verifica se os testes foram satisfeitos. Fazendo esta verificação através do *output* recebido da execução feita pela ferramenta de testes.
9. Os arquivos temporários são removidos.
10. A resposta é retornada.

4.3.4 Segurança

Como o servidor executa um código vindo diretamente do lado do cliente, é necessário uma camada de proteção para que não haja aberturas para códigos maliciosos no sistema.

A decisão de utilizar contêineres *Docker* para executar os exercícios teve como um dos motivos o de segurança. Um contêiner possui diversas configurações que permitem proteger o sistema limitando recursos usados pelo código e impedindo determinadas ações.

No caso do servidor *Runner*, não há possibilidade de realizar chamadas de *API* através do código, pois o contêiner está desconectado da rede. Além disso, temos os seguintes limitadores configurados:

- O máximo de memória que pode ser utilizada é 768MB. O mesmo serve para a memória do kernel.
- O número máximo de PIDS (*Process Identifiers*, que são identificadores dados pelo sistema operacional a cada processo que está sendo executado) é 128.

- Uma lista de *ulimits*, que são parâmetros que restringem o número de recursos que um processo pode utilizar. Neste caso, o processo que possui recursos limitados é a própria execução do contêiner.

4.3.5 Verificadores de teste

As ferramentas de teste atuais mostram os resultados de testes na saída padrão determinada pelo usuário, sendo esta normalmente o terminal do sistema operacional. Todas possuem elementos que indicam se todos os testes passaram ou não, para facilitar a usabilidade do programador.

Pensando nisso, foi criado uma classe que identifica os resultados do teste através do output retornado pela execução dos testes no contêiner *Docker*, retornando a resposta através de uma variável booleana.

Isto é feito para que o *Client* crie uma experiência melhor para os usuários com o uso dessa informação.

4.3.6 Adicionando novos ambientes

Ao criar um ambiente de teste novo, são necessárias algumas modificações no servidor *Runner* para que se torne funcional. Para isto, deve-se fazer o seguinte:

- Adicionar o nome da imagem docker adicionada ao *DockerHub* no enum *DockerImageName*.
- Adicionar o identificador do ambiente de teste no enum *TestingEnvironment*.
- No contêiner de dependências do servidor, adicionar um mapeamento do ambiente de teste adicionado para o nome da imagem *Docker*.
- Criar um verificador de teste para este ambiente de testes e adicioná-lo ao construtor *TestVerifierBuilder*.

O motivo de não ter sido utilizada uma lógica mais dinâmica para isso foi para que o servidor pudesse captar erros nos identificadores dos ambientes de teste. Caso a requisição contenha um ambiente que não exista, o código não será executado.

4.4 *Exercises*

4.4.1 Propósito

Este servidor foi criado com o objetivo de armazenar todos os exercícios e seções da plataforma. Além disso, como é esperado que outros programadores possam contribuir com a plataforma, adicionando novos exercícios, esta ferramenta foi criada para tornar este processo mais fácil.

4.4.2 Implementação

Assim como o servidor *Runner*, também foi implementado em *Python* utilizando o *framework FastAPI*.

Há uma única rota, que é executada através do método *HTTP GET*, que devolve os seguintes parâmetros:

- Uma lista de todos os exercícios e seções da página.
- Um mapa de IDs para exercícios ou seções.

4.4.3 Diretório de definições de exercícios e seções

O diretório *definitions* do repositório contém todas as definições de exercícios (no diretório *exercises*) e seções (no diretório *sections*).

4.4.4 Estrutura de um exercício no repositório

As descrições dos exercícios são seus enunciados, onde é especificado o que é esperado como resposta do usuário. Estes textos são escritos no formato *Markdown* e podem existir mais de um por exercício. Isto acontece pelo fato de existirem múltiplos ambientes de desenvolvimento. Como uma descrição pode conter trechos de código, é interessante que ele esteja escrito na linguagem escolhida pelo usuário. As descrições devem ser criadas em um diretório nomeado de descriptions, dentro do diretório do exercício.

Soluções demonstram um exemplo de como o exercício pode ser resolvido utilizando a metodologia da seção a que ele pertence, fazendo isso de forma detalhada para o melhor entendimento do usuário. Da mesma forma que as descrições, as soluções são escritas em *Markdown* e podem possuir múltiplas versões, variando de acordo com o ambiente de desenvolvimento escolhido. As soluções também possuem seu próprio diretório, chamado de solutions, dentro do diretório do exercício.

No diretório do exercício deve existir uma pasta chamada *files*, que contém os arquivos que serão modificados ou apenas consultados. Nela estarão arquivos de código na linguagem escolhida pelo usuário.

Os ambientes de teste possuem identificadores únicos entre eles, que serão explicados posteriormente nesta monografia. Cada exercício possui uma lista de ambientes de teste disponíveis, bem como os respectivos arquivos, descrições e soluções de cada um. Definimos um tipo de variável chamado *TestEnvironmentID* para este identificador, que será utilizado para futuras explicações.

A estrutura seguinte representa um arquivo que será utilizado no exercício, vindo da pasta *files* (em JSON):

- `fileName` (`String`): Nome do arquivo contido dentro da pasta files do exercício.
- `type` (`CODE` ou `TEST`): O tipo `CODE` é usado quando o arquivo representa um código, já `TEST` quando representa um arquivo de teste.

A estrutura demonstrada será chamada de *ExerciseFileDefinition*, para menções em futuras explicações.

O arquivo *definition.json* é utilizado para representar a estrutura geral do exercício, contendo suas informações principais. Sua estrutura é especificada pelos seguintes atributos:

- `id` (`ID`): Identificador do exercício. Deve ser único entre todas as seções e exercícios.
- `panelLabel` (`String`): Define o rótulo que será utilizado para representar o exercício no painel de exercícios. É importante que seja único também para não confundir o usuário.
- `panelPosition` (`Optional<Integer>`): A posição em que o exercício será renderizado no painel de exercícios. Caso não seja determinado, será colocado nas últimas posições. Se o exercício for parte de uma seção, sua posição será relativa aos exercícios desta mesma seção, caso contrário será relativa aos outros elementos do painel.
- `testingEnvironments` (`Array<TestingEnvironmentID>`): Lista de ambientes de teste habilitados neste exercício.
- `fileSchemasMap` (`Map<TestingEnvironmentID, Array<ExerciseFileDefinition>>`): Um mapa de identificadores de ambientes de teste para listas de definições de arquivos de exercícios. Utilizado para selecionar os arquivos de acordo com o ambiente de teste selecionado pelo usuário.
- `testCommandsMap` (`Map<TestEnvironmentID, String>`): Um mapa de ambientes de teste para comandos bash, que serão utilizados para executar o exercício no servidor *Runner*. Os comandos podem mudar de acordo com o ambiente de teste.
- `description` (`String` ou `Map<TestEnvironmentID, String>`): Este valor refere-se aos arquivos *Markdown* dentro da pasta descriptions do exercício. Se o valor for uma única *String*, então o mesmo *Markdown* será utilizado como descrição sem importar o ambiente de teste selecionado. Do contrário, o exercício terá uma descrição para cada ambiente de teste habilitado no exercício.
- `solution` (`String` ou `Map<TestEnvironmentID, String>`): Tem a mesma finalidade do campo description, porém com as soluções do exercício. Utiliza os arquivos *Markdown* dentro da pasta solutions. Possui a mesma lógica do campo description para diferença entre *String* e mapa.

Para criar um novo exercício dentro do servidor Exercises, basta seguir os seguintes passos abaixo:

1. Criar uma pasta dentro do diretório *definitions/exercises*.
2. Dentro do diretório criar um arquivo *definition.json* e as pastas descriptions, files e solutions.
3. Criar os arquivos *Markdown* das descrições dentro da pasta descriptions.
4. Criar os arquivos *Markdown* das soluções dentro da pasta solutions.
5. Criar os arquivos de código e de teste dentro da pasta files.

6. Definir a estrutura do exercício no *definition.json*, seguindo as descrições da tabela de campos.

7. Escrever um enunciado e/ou explicar o conceito utilizado no exercício nos arquivos *Markdown* de descrição.

8. Escrever uma solução detalhada que resolve o exercício utilizando a metodologia ensinada na seção ou no próprio exercício, para o melhor entendimento do usuário.

9. Preencher o conteúdo dos arquivos de código e teste da pasta *files*.

10. Executar o script *generate_exercises.sh*, para gerar um único arquivo JSON contendo todos os exercícios e seções da plataforma, que será utilizado na resposta da requisição do servidor.

4.4.5 Estrutura de uma seção no repositório

Toda seção possui uma descrição, que provê um resumo da metodologia ou conceito ensinado no capítulo a qual ela se refere. É escrita no formato *Markdown* em um arquivo chamado *description.md*, dentro do diretório da seção. Diferente dos exercícios, seções possuem uma única descrição, já que não há como escolher ambiente de desenvolvimento em seções. Assim como exercícios, seções também possuem este arquivo para determinar sua estrutura. Possui a seguinte estrutura JSON:

- `id` (`ID`): Identificador da seção. Deve ser único entre seções e exercícios.

- `panelLabel` (`String`): Define o rótulo da seção no painel de exercícios da plataforma. É importante que seja único para não afetar a experiência do usuário com múltiplos rótulos iguais.

- `panelPosition` (`Optional<Integer>`): Posição do rótulo da seção no painel de exercícios. Caso não seja determinado, será colocado nas últimas posições.

- `exerciseIds` (`Array<ID>`): Lista de *IDs* de exercícios que farão parte desta seção.

Vale mencionar que os exercícios desta seção devem ser construídos antes dela, pois seus identificadores serão utilizados para determinar a lista de exercícios da seção.

Os passos para criar uma nova seção no servidor *Exercises* são os seguintes:

1. Criar uma pasta dentro do diretório *definitions/sections* para a nova seção.

2. Criar os arquivos *definition.json* e *description.md* dentro da pasta criada.

3. Definir a estrutura da seção no arquivo *definition.json*, seguindo as descrições da tabela.

4. Escrever um resumo do conceito ou metodologia que é utilizado nos exercícios desta seção.

4.5 *Environments*

4.5.1 Propósito

É um repositório que armazena *Dockerfiles*, usados para construir os contêineres que montam os ambientes de teste da plataforma. Cada ambiente possui as linguagens e bibliotecas necessárias para executar os exercícios.

Por enquanto há apenas dois ambientes disponíveis, um deles com *Javascript* como linguagem e outro com *Typescript*, ambos com Jest como ferramenta de teste.

4.5.2 Armazenamento

No repositório, cada pasta representa um ambiente. Cada uma deve conter um *Dockerfile*, além dos arquivos necessários para construir o ambiente dentro dela. O ambiente de *Typescript* e *Jest*, por exemplo, além de conter o *Dockerfile*, que instala as ferramentas necessárias e constrói a imagem, possui alguns arquivos de configuração que são usados para determinar o comportamento da linguagem e da ferramenta de teste.

Depois que a imagem é construída, ela deve ser adicionada no perfil vinigpereira do *DockerHub*, pois é de onde será acessada pelo serviço *Runner*. O nome da imagem deve ser um identificador único (e.g. *javascript-jest*), pois este *ID* também será usado em outros serviços.

Todo o processo de criação no repositório e armazenamento no *DockerHub* é feito manualmente.

4.5.3 Definindo novos ambientes

O processo de criação de novos ambientes de teste é manual. É feito pelos seguintes passos:

1. Criar um novo diretório dentro do repositório. A prática recomendada é utilizar o *ID* do ambiente como nome do diretório.
2. Dentro do diretório, criar um *Dockerfile* que define o ambiente com tudo necessário para executar testes criados a partir de uma linguagem e uma ferramenta de teste. Arquivos que serão usados dentro do contêiner *Docker* devem ser criados neste diretório também.
3. Construir a imagem a partir do *Dockerfile* e adicioná-la ao perfil vinigpereira do *DockerHub*.
4. Adicionar o novo ambiente no *README* do repositório.
5. Para tornar o ambiente criado funcional, é necessário realizar algumas modificações no serviço Runner. Os passos estão na Seção 4.3.6.

Capítulo 5

Conclusão

5.1 Contribuições

Hoje, há uma carência de engenheiros de software com conhecimento na área de testes, seja por falta de interesse ou por desistência ao perceberem a falta de conteúdo organizado online.

A leitura tradicional é um método de estudo que vem sendo deixado em segundo plano nos dias de hoje, com a pesquisa por conteúdo na internet sendo colocada como prioridade. A razão disso talvez seja pelo fato dos estudantes necessitarem de uma forma prática de estudo.

Neste trabalho apresentamos o *Testify* como solução para estes problemas. A plataforma incentiva desenvolvedores a estudarem sobre testes através de uma forma visual e de fácil usabilidade; promove a leitura de livros técnicos, através de resumos e exercícios baseados no livro de Aniche; provê um conteúdo organizado com base no conceito e na complexidade e contém exercícios práticos para fixação de conhecimento.

5.2 Próximos passos em conjunto com Maurício Aniche

O produto final conclui os objetivos propostos por este trabalho, porém haverão mudanças futuras com base em sugestões e pedidos feitos por Maurício Aniche, o autor do livro que foi utilizado como base para este projeto. Ele possui uma ferramenta de feedback e uma lista de 50 exercícios que seus alunos utilizam para estudar. O objetivo é transferir estes exercícios para o *Testify*, de forma que possam utilizá-lo diretamente na plataforma para terem acesso a todas funcionalidades que ela possui, principalmente a de execução de exercícios. Todos os exercícios possuem a mesma estrutura de arquivos, portanto uma possível implementação deverá envolver o padrão de design *Adapter* para incorporá-los usando a estrutura que o servidor de *Exercises* reconhece e utilizá-los na construção do JSON final.

Referências

[Aniche 2021] Maurício Aniche. *Effective Software Testing - A developer's guide*. 2021 (citado nas pgs. 8, 10).

[Cohn 2009] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. 2009 (citado na pg. 8).

[Fowler 2014] Martin Fowler. *Self Testing Code*. url: https://martinfowler.com/bliki/SelfTestingCode.html (acesso em 01/05/2014) (citado na pg. 7).

[Vocke 2018] Ham Vocke. *The Practical Test Pyramid*. url: https://martinfowler.com/articles/practical-test-pyramid.html (acesso em 26/02/2018) (citado na pg. 8).