GUILHERME DIAS TAVARES

Arquitetura de um ambiente de teste simulado front e back-end

São Paulo
2025

GUILHERME DIAS TAVARES

Arquitetura de um ambiente de teste simulado front e back-end

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

São Paulo
2025

GUILHERME DIAS TAVARES

Arquitetura de um ambiente de teste simulado front e back-end

Versão Original

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Área de Concentração:
Engenharia de Software

Orientador:

Prof. Dr. Jorge Luis Risco Becerra

Co-orientador:

Prof. Alípio Ferro

São Paulo
2025

RESUMO

Este trabalho apresenta o desenvolvimento de um ambiente de teste simulado para sistemas que adotam arquitetura *headless*, com base no padrão arquitetural Modelo-Visão-Controlador (MVC). A proposta central é oferecer uma infraestrutura de teste modular e escalável que permita simular cenários reais de uso, garantindo resultados confiáveis e replicáveis. A abordagem prática incluiu a aplicação de um módulo CRUD (*Create*, *Read*, *Update* e *Delete*) genérico como exemplo ilustrativo, destacando sua interação com o ambiente de teste. Embora o trabalho não tenha incluído um levantamento formal de requisitos ou uma avaliação iterativa, ele demonstra os benefícios de ambientes de teste próximos às condições de produção, abordando questões como validação de APIs. O estudo conclui destacando a importância de manter o *mock data* alinhado ao *back-end* real, sugerindo que futuras pesquisas explorem ferramentas para sincronização automatizada de dados de teste.

Palavras-Chave: teste de *software*, arquitetura *headless*, MVC, ambiente de teste simulado, validação de API.

ABSTRACT

This study presents the development of a simulated testing environment for systems adopting a headless architecture, based on the Model-View-Controller (MVC) architectural pattern. The primary goal is to provide a modular and scalable testing infrastructure capable of simulating real-world usage scenarios, ensuring reliable and replicable results. The practical approach included the application of a generic CRUD module as an illustrative example, highlighting its interaction with the testing environment. Although the work did not involve a formal requirements gathering process or iterative evaluation, it demonstrates the benefits of testing environments closely resembling production conditions, addressing issues such as API validation. The study concludes by emphasizing the importance of keeping mock data aligned with the actual backend, suggesting future research to explore tools for automated synchronization of testing data.

Keywords – software testing, headless architecture, MVC, simulated testing environment, API validation.

LISTA DE FIGURAS

LISTA DE SIGLAS

API – *Application Programming Interface*

BPMN – *Business Process Model and Notation*

CMO – *Chief Marketing Officer*

CRM – *Customer Relationship Management*

CTO – *Chief Technology Officer*

ERP – *Enterprise Resource Planning*

MBA – *Master in Business Administration*

MVC – *Model-View-Controller* (Modelo-Visão-Controlador)

SAML – *Security Assertion Markup Language*

TI – Tecnologia da Informação

SUMÁRIO

1 INTRODUÇÃO

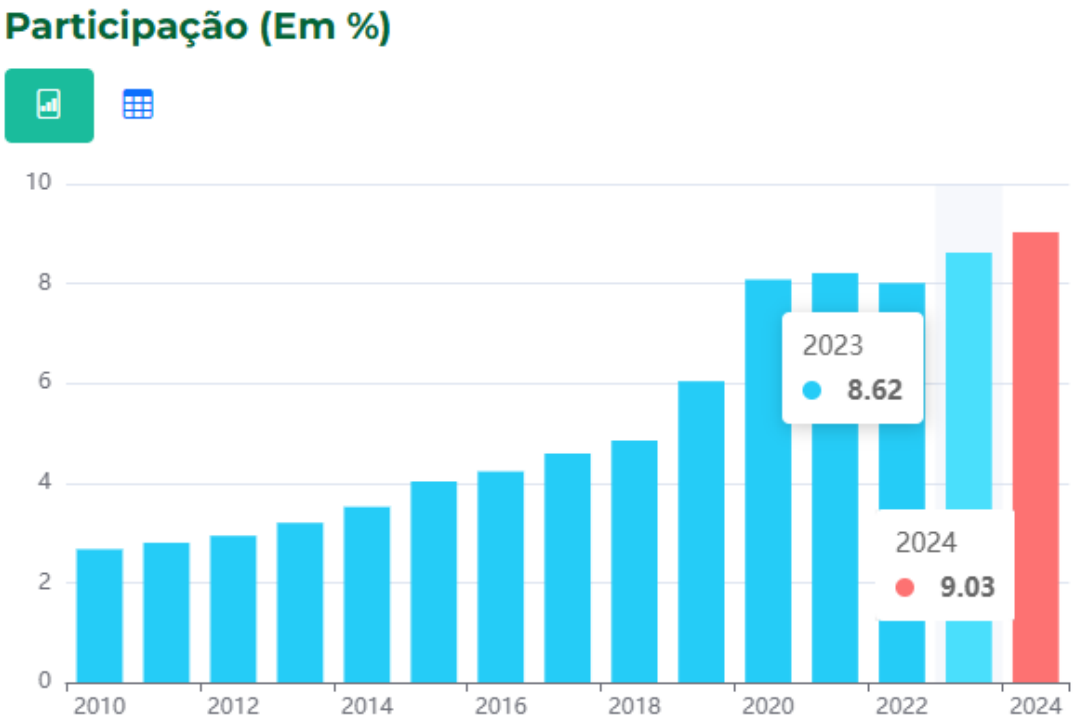
Este trabalho é desenvolvido como parte do *Master in Business Administration* - MBA de Engenharia de *Software*. Neste capítulo é apresentado o contexto o qual o trabalho está inserido, o objetivo, a justificativa, a metodologia e a organização do mesmo.

1.1 Contexto Inicial

A adoção de sistemas com arquitetura *headless* vem ganhando relevância, muito por conta da flexibilidade e escalabilidade que tal arquitetura fornece, uma vez que o desacoplamento total entre *back-end* e *front-end* possibilita uma experiência de usuário personalizada de acordo com o dispositivo a ser utilizado, como por exemplo, aplicativos móveis, TVs inteligentes e interfaces próprias.

Abordando um pouco mais a questão do mercado, aplicações de comércio eletrônico (*e-commerce*) foram um dos principais impulsionadores da transformação digital nas últimas décadas, tornando-se uma fonte importante de faturamento das empresas, sendo responsável no ano de 2023 por 8,62% do faturamento total do seguimento de varejo, segundo a Associação Brasileira de Comércio Eletrônico (ABComm) (2024). Ainda nesta linha, a previsão para o ano de 2024 é de que o faturamento ultrapasse a barreira dos 9%, conforme ilustra a Figura 1 a seguir:

Figura 1: Participação do E-commerce no Varejo



Fonte: Associação Brasileira de Comércio Eletrônico (ABComm) (2024)

Contudo, essa flexibilidade promovida pela arquitetura *headless* acabou gerando desafio no que tange aos testes de aplicação. Supondo que podemos ter duas aplicações visuais, com experiências totalmente distintas, porém com propósitos (vendas) e conexão iguais (neste caso, ambas apontam para o mesmo sistema de *back-end*), qual seria a melhor forma para testar as duas aplicações de forma isolada, ou seja, sem poder contar com o sistema que sustenta todas operações que são realizadas na plataforma? É justamente neste cenário que o presente trabalho está inserido.

1.2 Objetivo

Este trabalho tem como principal objetivo propor uma arquitetura de ambiente de teste simulado para *front-end* e *back-end*, baseada no modelo MVC (*Model-View-Controller*).

A arquitetura foi gerada considerando um contexto de aplicação em uma plataforma de *e-commerce* e restringindo um pouco mais o escopo para dentro do módulo de gerenciamento de carrinho. No módulo de gerenciamento de carrinho considerou-se quatro ações principais para fins ilustrativos: Criação do Carrinho, Obter dados do Carrinho, Atualizar dados do carrinho e Deletar carrinho. A escolha pelo módulo de carrinho foi estratégica, uma vez que é uma prática comum e desejada no mercado o compartilhamento do mesmo entre as plataformas, isto é, quando um cliente final montar um carrinho utilizando um aplicativo móvel em seu celular,

caso queira o mesmo poderá entrar via navegador de computador e ter os dados já persistidos junto ao seu usuário.

1.3 Justificativa

De acordo com uma pesquisa conduzida pela Censuswide, uma consultoria especializada em pesquisas de mercado quantitativas e qualitativas, a pedido da WP Engine, em julho de 2024, 73% das empresas entrevistadas já utilizam a arquitetura *headless*, representando um aumento de 14% em relação à 2021 (ENGINE, 2024). Dentre as empresas que ainda não adotaram essa tecnologia, 98% planejam avaliar sua implementação nos próximos 12 meses. A pesquisa incluiu respostas de executivos como diretores de tecnologia (CTOs), diretores de marketing (CMOs) e tomadores de decisão da área de TI, provenientes de empresas com receita média anual de aproximadamente 800 milhões de dólares. É importante ressaltar que o estudo foi conduzido em organizações localizadas nos Estados Unidos, Reino Unido e Austrália, refletindo tendências regionais específicas na adoção dessa arquitetura (ENGINE, 2024).

Seguindo essa sequência, foram examinados diversos artigos que têm ligação, seja direta ou indireta, completa ou parcial, com os temas abordados nesta monografia. Dentre esses, foram selecionados três artigos os quais serão sintetizados nos parágrafos seguintes.

O desenvolvimento de ambientes de testes que simulam o comportamento real de sistemas de produção é um desafio importante na engenharia de software. O artigo *Mimicking Production Behavior with Generated Mocks* de Tiwari et al. (2024) aborda a utilização de *mocks*, isto é, objetos ou componentes simulados que simulam o comportamento de dependências reais de um sistema, gerados automaticamente para replicar de forma precisa o comportamento do sistema em produção durante os testes. Essa abordagem permite a execução de testes mais confiáveis, reduzindo a dependência de ambientes de produção e aumentando a cobertura de cenários de falhas e carga. Essa técnica é diretamente aplicável à proposta desta monografia.

O artigo *A Method of Automated Mock Data Generation for RESTful API Testing* de Thu et al. (2022) explora a geração automatizada de dados *mock* para testes de APIs, abordando a criação de *mocks* com base nas especificações da API, o que pode ser diretamente relacionado à construção de um ambiente de testes simulados descrito na monografia. Embora a monografia foque na criação de um ambiente de teste para sistemas que adotam a arquitetura *headless*, a abordagem de geração automatizada de dados *mock* no artigo complementa a proposta de testar interações e respostas do sistema, sem a necessidade de um ambiente de produção.

1.4 Metodologia

A metodologia deste trabalho baseou-se em uma abordagem exploratória e aplicada, com foco na criação e apresentação de um ambiente de teste simulado para sistemas que utilizam arquitetura *headless*. A primeira etapa envolveu uma revisão bibliográfica direcionada, abordando conceitos teóricos sobre padrões arquiteturais, como o modelo MVC, e metodologias de teste de software. Essa revisão fundamentou as escolhas técnicas e conceituais empregadas ao longo do desenvolvimento do trabalho.

Em seguida, adotou-se uma abordagem prática com base em um exemplo aplicado de um módulo CRUD do módulo de gerenciamento de Carrinho. Este exemplo serviu para ilustrar como um ambiente de teste simulado pode interagir com um sistema a ser avaliado, mesmo sem um levantamento formal de requisitos ou iterações documentadas de avaliação. O foco principal esteve na representação arquitetural do ambiente de testes.

Apesar de suas limitações de escopo, a metodologia aplicada possibilitou a geração de uma arquitetura de um ambiente de testes de acordo com o que era previsto e desejado incialmente.

1.5 Organização

A organização do trabalho é composta da seguinte forma:

O Capítulo 1 INTRODUÇÃO apresenta o contexto inicial, o objetivo, as justificativas, a metodologia e a estrutura do trabalho;

O Capítulo 2 FUNDAMENTAÇÃO TEÓRICA apresenta os principais conceitos para o desenvolvimento deste trabalho, como Arquitetura, o padrão MVC e Testes de *Software*;

O Capítulo 3 DESENVOLVIMENTO apresenta o desenvolvimento da monografia, abordando o contexto da aplicação, o processo de desenvolvimento, o produto principal e a implementação.

O Capítulo 4 CONSIDERAÇÕES FINAIS apresenta a conclusão e propostas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O presente capítulo tem como objetivo fornecer o embasamento teórico necessário para o desenvolvimento desta monografia, alinhando os conceitos fundamentais ao contexto e aos objetivos estabelecidos no capítulo anterior.

Para isso, será realizada uma revisão bibliográfica de um pattern arquitetural muito difundido e utilizado, o Modelo-Visão-Controlador (*Model-View-Controller*) ou no acrônimo MVC. Além do mais, também é abordado os conceitos de Teste Web e teste distribuído.

2.1 Padrão de Arquitetura - Model-View-Controller

Arquitetura de *software* é constantemente definida como uma forma abstrata, de mais alto nível de representação de um projeto, podendo ainda, definir e exibir um projeto com vários pontos de vistas distintos, oportunizando assim uma ampla gama de aplicação e objetivos de se ter uma arquitetura, como por exemplo usar para documentar um projeto, representar requisitos, etc. É também, segundo Sommerville (2020), como o primeiro estágio no projeto de um *software*.

De acordo com Sommerville (2020), um projeto de arquitetura tem preocupação na demonstração em como um sistema deve ser organizado e sua estrutura geral, uma vez que é o elo crítico entre projeto e engenharia de requisitos, uma vez que identifica os principais componentes do projeto e a forma com que se relacionam.

Com base nisto, é importante e primordial existir modelos e padrões de arquitetura. Sommerville (2020) descreve um padrão de arquitetura como um conjunto de boas práticas testadas em diferentes contextos, que incluem orientações sobre sua aplicabilidade, pontos fortes e limitações.

O padrão arquitetural MVC (*Model-View-Controller*, ou em português, Modelo-Visão-Controlador) é um robusto padrão para desenvolvimento de sistemas de *software*, que tem origem no final da década de 70 e início da década de 80, durante o desenvolvimento do ambiente Smalltak-80, uma linguagem de programação orientada a objetos. Foi concebido para organizar e segregar a lógica do sistema, especialmente em sistemas que há a existência de interfaces gráficas e/ou sistemas Web. Este conceito divide e organiza a arquitetura em três componentes principais: Modelo, Visão e Controlador, como seu próprio nome sugere.

O componente Modelo é a camada responsável por encapsular os dados e regras de negócios que estão associadas aos mesmos. Em outras palavras, é o encarregado de gerenciar o estado do sistema.

Já o componente Visão tem como escopo a forma com que estes dados são exibidos ao usuário.

Por sua vez o componente Controlador é o responsável por gerenciar a integração entre Modelo e Visão, sendo uma espécie de "meio de campo" entre os componentes citados anteriormente.

Uma grande vantagem da abordagem de uma arquitetura MVC é a segregação entre as camadas de forma com que as alterações possam ser realizadas de forma pontuais e isoladas.

A fim de exemplificar de forma sucinta o conceito, Pressman (2014) apresenta o seguinte:

> Na arquitetura MVC, o comando do usuário é enviado da janela do navegador para um processador de comandos (controlador), o qual gerencia o acesso ao conteúdo (modelo) e instrui o modelo de renderização [NT] de informações (visão) a transformá-lo para exibição pelo *software* do navegador.

A Figura 2 demonstra uma ilustração visual da arquitetura MVC onde a comunicação e dependências entre os componentes são exibidas e ressaltadas:

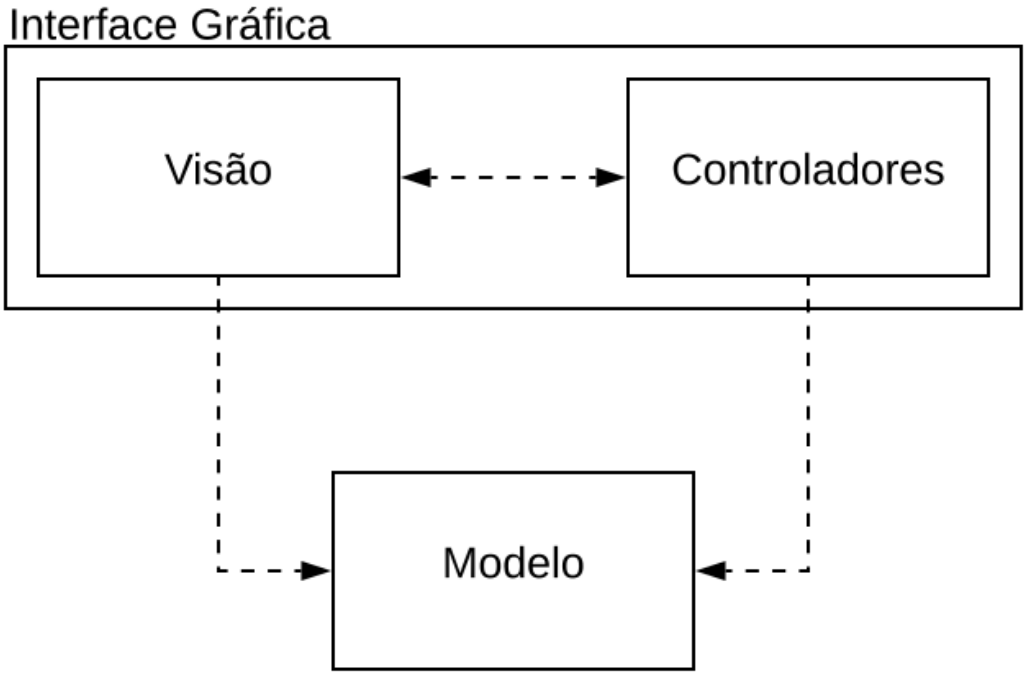


Figura 2: Arquitetura MVC

Fonte: Valente (2020)

2.2 Teste de *software*

Um dos grandes desafios de projetos complexos é garantir a estabilidade do mesmo, isto é, assegurar de forma consistente que toda alteração, por menor que ela seja, não gere impactos e/ou efeitos adversos, indesejados e inesperados no restante do projeto.

De acordo com Sommerville (2020):

> O teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso. Quando se testa o *software*, o programa é executado usando dados fictícios. Os resultados do teste são verificados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa.

Os testes de *software* desempenham um papel fundamental na garantia da qualidade dos sistemas, sendo definidos como um processo sistemático para identificar defeitos e verificar se o *software* atende aos requisitos especificados. Segundo Pressman (2014, p. 466), "Teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente". Os testes de *software* tem como preocupação garantir que o sistema esteja adequado ao seu propósito, atendendo as necessidades dos usuários, sendo executado de forma eficiente e confiável e também atendendo aos prazos estabelecidos bem como o orçamento e que o uso destas técnicas aumentou o nível de qualidade de *software* nos últimos anos (SOMMERVILLE, 2020).

O teste de *software* é composto por diversas técnicas e abordagens que variam de acordo com o ciclo de desenvolvimento e a complexidade do sistema, como testes unitários, de integração, de sistema e de aceitação. Além disso, os testes não apenas identificam defeitos técnicos, mas também servem como uma ferramenta para melhorar a confiabilidade e a segurança do *software*.

Os testes podem ser divididos em quatro etapas: teste de unidade, teste de integração, teste de validação e testes de ordem superior (PRESSMAN, 2014). O teste de unidade é basicamente o tipo de teste que se concentra em avaliar uma única unidade do projeto, seja ela um componente, uma classe, etc.). Por sua vez, o teste de integração tem o foco no projeto e sua arquitetura. O teste de validação verifica se a modelagem de requisitos foi cumprida. Por fim, teste de ordem superior realiza uma avaliação de como o *software* se comporta e interage com elementos externos.

2.2.1 Teste de Integração

O teste de integração é crucial para garantir que diferentes módulos de um sistema, funcionem corretamente em conjunto. Em um ambiente de arquitetura *headless*, onde existe o desacoplado, é essencial validar a comunicação entre as APIs e o sistema de interface do usuário.

Os testes de integração simulam interações reais para verificar se os dados são processados e apresentados corretamente, identificando falhas nas interfaces e integrando de maneira eficaz os componentes do sistema. Esse tipo de teste ajuda a garantir que as funcionalidades *end-to-end* atendam aos requisitos esperados.

2.2.2 Teste Simulado

Mocking é uma técnica utilizada para substituir dependências externas em testes de software, permitindo simular o comportamento de sistemas externos como APIs ou serviços. Conforme avaliado por Valente (2020) o *mock* permite a implementação de teste que não precisa acessar serviço remoto, potencialmente lento. Em ambientes de testes de integração, o uso de *mocks* é essencial para isolar os componentes que estão sendo testados, garantindo que falhas de integração com sistemas externos não interferirem nos resultados dos testes.

3 DESENVOLVIMENTO

Este capítulo tem como objetivo apresentar o processo de construção de uma arquitetura de ambiente de teste simulado, projetada para se aproximar das condições de produção em sistemas que adotam a arquitetura *headless*. A solução proposta, fundamentada no modelo MVC *(Model-View-Controller)*, foi desenvolvida para garantir que os testes sejam realizados de maneira modular, escalável e replicáveis. Serão discutidos neste capítulo as metodologias, técnicas e ferramentas utilizadas para a criação da arquitetura de um ambiente de teste.

No decorrer do capítulo, serão abordados os detalhes do contexto da aplicação, como o domínio da aplicação, as fases do processo de desenvolvimento, e os entregáveis intermediários. Além disso, será apresentada a estrutura do produto final, como este se organiza e opera dentro do ambiente de testes.

3.1 Contexto da Aplicação

O contexto da aplicação é fundamental para compreender o ambiente e os desafios nos quais a solução proposta será implementada. Este subitem busca abordar os principais aspectos relacionados à aplicação do produto desenvolvido nesta monografia, que se insere no domínio do comércio eletrônico, especificamente para plataformas que adotam arquiteturas *headless*. A partir dessa análise, serão detalhados o domínio da aplicação, os *stakeholders* envolvidos, os processos de negócios impactados, os requisitos e limitações do sistema, e as interações com agentes e sistemas externos.

3.1.1 Domínio da aplicação

Levando em consideração o que foi discorrido anteriomente, a solução proposta será aplicada a um hipotético sistema de *e-commerce headless*, visto que permite que múltiplas interfaces interajam com o mesmo núcleo de serviços.

Essa escolha de domínio reforça a relevância prática do produto da monografia, considerando as demandas atuais do mercado e a necessidade crescente de ambientes de teste que simulem com precisão cenários de produção.

3.1.2 Partes Interessadas (*Stakeholders*) e processos de negócios

Os *stakeholders* diretamente relacionados ao produto desta monografia incluem indivíduos e equipes que desempenham papéis essenciais no ciclo de vida de desenvolvimento e operação do sistema *e-commerce*, conforme detalhado abaixo:

1. *Stakeholders* técnicos

   (a) Desenvolvedores *front-end*: Responsáveis pela criação e manutenção das interfaces visuais que interagem com os usuários finais. Dependem de um ambiente de teste para validar o desenvolvimento levando em conta as respostas da integração com APIs e serviços do *back-end* desacoplado.

   (b) Desenvolvedores *back-end*: Projetam e mantêm os serviços e APIs utilizados pelo *front-end*. Seu foco está na garantia de que os *endpoints* de API funcionem conforme o esperado e atendam às especificações definidas.

   (c) Engenheiros de Qualidade: Testam o sistema como um todo, garantindo que tanto o *front-end* quanto o *back-end* operem corretamente em diferentes cenários de acordo com a especificação so sistema.

   (d) *DevOps*: Garantem que os ambientes de teste e produção sejam configurados de forma eficaz.

2. *Stakeholders* de negócios

   (a) Gerentes de Produto: Interessados em validar que os requisitos de negócio sejam atendidos e que as funcionalidades entreguem valor ao usuário final.

   (b) Executivos e Tomadores de Decisão: Avaliam os resultados dos testes como parte do processo de tomada de decisões estratégicas para otimização de custos e aumento de confiabilidade e expansão do sistema.

   (c) Usuários Finais (Clientes): Embora indiretamente, são afetados pela confiabilidade do sistema, que impacta diretamente a experiência de compra e a fidelização.

Um sistema de *e-commerce* possuí variados processos de negócios. A seguir são apresentados os mais impactantes no ecossistema:

- Gerenciamento de Catálogo de Produtos: As APIs precisam ser testadas para garantir a atualização e exibição correta de produtos nas interfaces do usuário, ou seja, uma vez que o *back-end* receba uma requisição de atualização para o banco, que também seja capaz de entregar o dado já atualizado para as consultas que sejam realizadas na sequência.

- Processamento de Pedidos e Transações: O *back-end* deve ser validado para garantir a integridade das transações, como pagamentos e atualizações de status de pedidos.
- Gerenciamento de Clientes: O fluxo de gerenciamento de cliente deve ser validado para que forneça dados com qualidade para serviços externos (como *Analytics*) mas que principalmente também respeite quaisquer regras de negócio atribuídas a este módulo (como por exemplo liberar compras de bebidas alcoólicas apenas para consumidores maiores de idade.

3.1.3 Requisitos e limitações

Para o desenvolvimento da arquitetura de ambiente de teste simulado proposto nesta monografia, foi necessário identificar e documentar os requisitos essenciais que guiam sua construção, bem como as limitações inerentes à solução adotada.

3.1.3.1 Requisitos

Para falar um pouco dos requisitos, decidiu-se dividir entre requisitos funcionais e requisitos não funcionais. Os requisitos funcionais determinam o que um sistema deve fazer, enquanto os requisitos não funcionais especificam propriedades e restrições do sistema. (SOMMERVILLE, 2020)

Requisitos funcionais:

- O ambiente de testes deve permitir a simulação de interações entre *front-end* e *back-end*.
- Deve suportar a execução de testes sob condições semelhantes ao ambiente de produção.
- Deve ser capaz de utilizar mocks para APIs e dados.
- Implementação de CRUD para validar operações básicas de sistemas.

Requisitos não funcionais:

- Escalabilidade para permitir expansão de testes com diferentes cenários.
- Manutenção de consistência entre dados simulados e dados reais.

3.1.3.2 Limitações

Apesar das vantagens esperadas, algumas limitações foram identificadas:

- Operacional: Exige familiaridade dos desenvolvedores com conceitos de teste automatizado e *mocks*. Custos adicionais para infraestrutura de testes.

- Complexidade técnica: Dependência de ferramentas específicas para geração de *mocks*. Dificuldade em sincronizar dados de *mock* com atualizações no *back-end* real.
- Escopo do Trabalho: O foco está na arquitetura para ambientes de teste, não no desenvolvimento completo de uma solução.

3.1.4 Agentes, sistemas e processos de negócios externos

O ambiente de teste simulado proposto neste trabalho opera em um ecossistema que interage com diversos agentes, sistemas e processos de negócios externos. A identificação desses elementos está listada a seguir.

a) Agentes externos

- Usuários finais: Interagem com o *front-end*, sendo responsáveis por realizar ações e fornecer dados que acionam processos no *back-end*.
- Desenvolvedores e equipes de QA: Atuam como operadores e validadores do ambiente de teste, utilizando o sistema para identificar defeitos, realizar análises e propor melhorias.
- Gestores de produtos: Influenciam as prioridades e escopos dos testes, além de tomar decisões com base nos resultados obtidos.

b) Sistemas externos

- APIs de terceiros: Dependência de APIs externas para integrar funcionalidades adicionais, como gateways de pagamento, serviços de geolocalização, ou provedores de dados analíticos.
- Banco de dados e repositórios de dados: Fontes de dados externas que armazenam informações críticas, necessárias para validações ou para simular condições reais no ambiente de teste.

c) Processos de negócios externos

- Fluxos de integração com parceiros: Processos que envolvem o intercâmbio de dados entre a aplicação e sistemas de parceiros, como logística, ERP ou CRM, e que precisam ser simulados no ambiente de teste.
- Cadeia de suporte ao cliente: Inclui processos que não são diretamente controlados pelo ambiente de teste, mas impactam a experiência geral, como suporte técnico e atendimento

pós-venda.

- Monitoramento e análise de desempenho: Sistemas externos de monitoramento, como ferramentas de observabilidade ou métricas de uso, que auxiliam na avaliação de desempenho da aplicação em produção.

3.2 Processo de desenvolvimento

A presente seção aborda o processo de desenvolvimento do produto proposto por esta monografia, abordando o ciclo de engenharia dividido em quatro fases: planejamento e projeto, implementação, validação e entrega, bem como as fases do processo e quais foram os produtos intermediários gerados durante o desenvolvimento.

3.2.1 Ciclo da engenharia

O desenvolvimento da arquitetura do ambiente de teste simulado seguiu um ciclo de engenharia estruturado, com foco na modularidade e escalabilidade.

3.2.1.1 Planejamento e Projeto

A fase inicial envolveu a análise dos requisitos levantados no contexto do ambiente de teste simulado. Nesse estágio, como já descrito no item 3.1.3, foram identificados requisitos principais para a elaboração desta arquitetura.

3.2.1.2 Implementação

A implementação foi realizada em etapas iterativas. Inicialmente, as funcionalidades básicas do ambiente de teste foram desenvolvidas, como a configuração de *endpoints* simulados e a automação de processos de teste. Cada módulo foi implementado separadamente, validado em pequenos ciclos de teste e integrado ao ambiente global de forma incremental.

3.2.1.3 Validação

Após cada ciclo de implementação, foram realizadas rodadas de testes para garantir a consistência e o alinhamento com os requisitos definidos. Esses testes foram conduzidos em cenários que simulavam condições reais de operação, incluindo a interação com APIs externas e simulação de cargas variadas.

3.2.1.4 Entrega final

Com a conclusão das etapas de validação, o ambiente de teste simulado foi consolidado e documentado. A documentação inclui o detalhamento da arquitetura, o fluxo de operação do

ambiente e as orientações para a execução dos testes, de modo a facilitar a adoção por equipes de desenvolvimento e QA.

3.2.2 Fases do processo

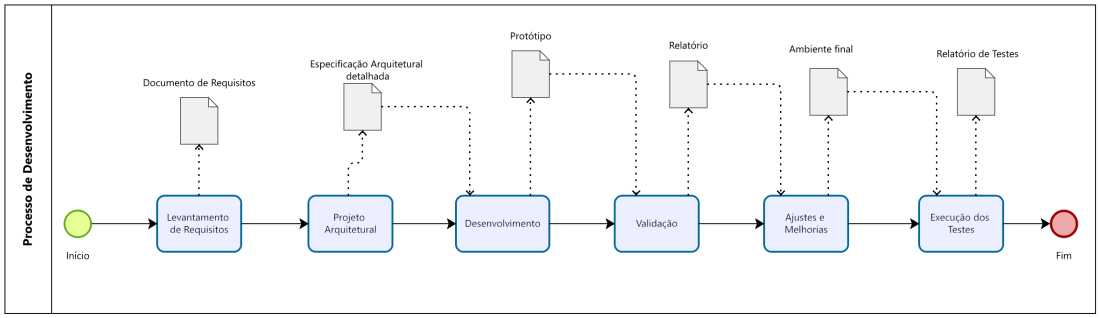
O ciclo de desenvolvimento foi organizado em fases distintas, com cada etapa produzindo subprodutos que contribuíram para a construção do ambiente de teste simulado. Essas fases foram representadas por meio de um modelo BPMN (*Business Process Model and Notation*), que ilustra os principais passos do processo de desenvolvimento.

Abaixo segue um descritivo com as fases, que posteriormente estarão representadas num modelo gráfico BPMN conforme ilustra a 3:

1. Levantamento de Requisitos
    - Atividade: Identificar necessidades técnicas e funcionais da arquitetura.
    - Subproduto: Documento de requisitos.
2. Projeto Arquitetural
    - Atividade: Projetar o modelo MVC e as interações entre as camadas.
    - Subproduto: Especificação arquitetural detalhada.
3. Desenvolvimento
    - Atividade: Implementar módulos básicos do ambiente (ex.: simulação de *endpoints* de criação de usuário).
    - Subproduto: Protótipo.
4. Validação
    - Atividade: Testar módulos isoladamente para verificar funcionalidade.
    - Subproduto: Relatório de validação de módulos.
5. Ajustes e Melhorias
    - Atividade: Ajustar de acordo com o que foi identificado no passo anterior e aplicar as melhorias identificadas.
    - Subproduto: Ambiente final.
6. Teste do Ambiente Completo
    - Atividade: Realizar testes abrangentes simulando condições reais de produção.

- Subproduto: Relatório de testes finais.

Figura 3: Modelo BPMN das Fases do Processo



Fonte: Autoria própria

3.2.3 Produtos intermediários

Os produtos intermediários foram gerados para assegurar o alinhamento entre os requisitos e o produto final, garantindo a validação em diferentes estágios do desenvolvimento.

1. Documento de Requisitos
    - Papel: Guiou todas as decisões do projeto, assegurando que as necessidades do ambiente fossem atendidas.
2. Especificação Arquitetural
    - Papel: Serviu como base para a implementação do ambiente, detalhando a divisão em camadas e a integração entre módulos.
3. Protótipo
    - Papel: Validar os conceitos aplicados e a forma de construção.
4. Relatório
    - Papel: Fornecer inputs para melhorias e correções do protótipo.
5. Ambiente Final
    - Papel: Permitiu a execução dos testes.
6. Relatório de Testes
    - Papel: Relatório dos testes realizados.

3.3 Produto principal

O produto principal desta monografia é a arquitetura de um ambiente de teste simulado projetado para se aproximar das condições reais de produção em sistemas baseados na arquitetura *headless*. A seguir, são detalhados a estrutura e o funcionamento do ambiente.

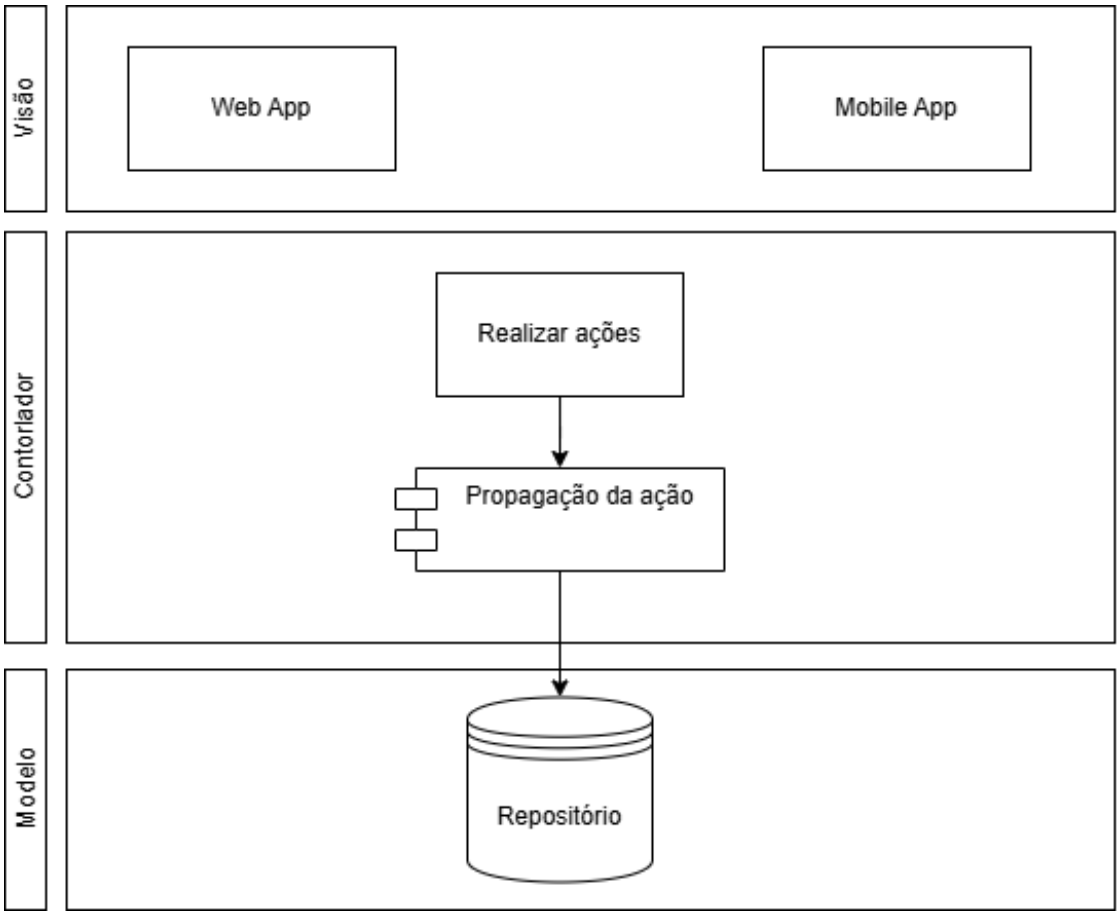
3.3.1 Estrutura do produto

O ambiente de teste foi concebido utilizando a arquitetura em camadas baseada no modelo MVC *(Model-View-Controller)*. Esta abordagem foi escolhida por sua capacidade de modularização e separação de responsabilidades, o que facilita a manutenção, escalabilidade e testes independentes. A seguir estão listadas as camadas MVC da arquitetura:

1. Model (Modelo):
    - Responsável pela simulação de dados e respostas das APIs.
    - Inclui *endpoints* representando diferentes cenários (ex.: respostas esperadas, erros simulados e variações de carga).
    - Implementado com ferramentas que permitem criar APIs REST simuladas e reprodutíveis.
2. View (Visão):
    - Representa a camada de interface do usuário.
    - Simula a interação do *front-end* com os *endpoints*, incluindo cenários como variações de dispositivos (desktop, mobile).
    - Utiliza frameworks de teste automatizado para verificar a exibição correta das respostas da API.
3. Controller (Controlador):
    - Realiza a coordenação entre as interações do *front-end* e *back-end*.
    - Processa as solicitações enviadas pela visão, repassando-as ao modelo e retornando as respostas apropriadas.
    - Inclui lógica de tratamento de erros e simulação de latência.

A figura 4 ilustra a montagem da arquitetura de um ecossistema *e-commerce headless*, considerando ações genéricas a serem performadas pelos usuários diretamente em contato com a camada de visão, seja ela um *Web App* ou um aplicativo móvel (*Mobile App*):

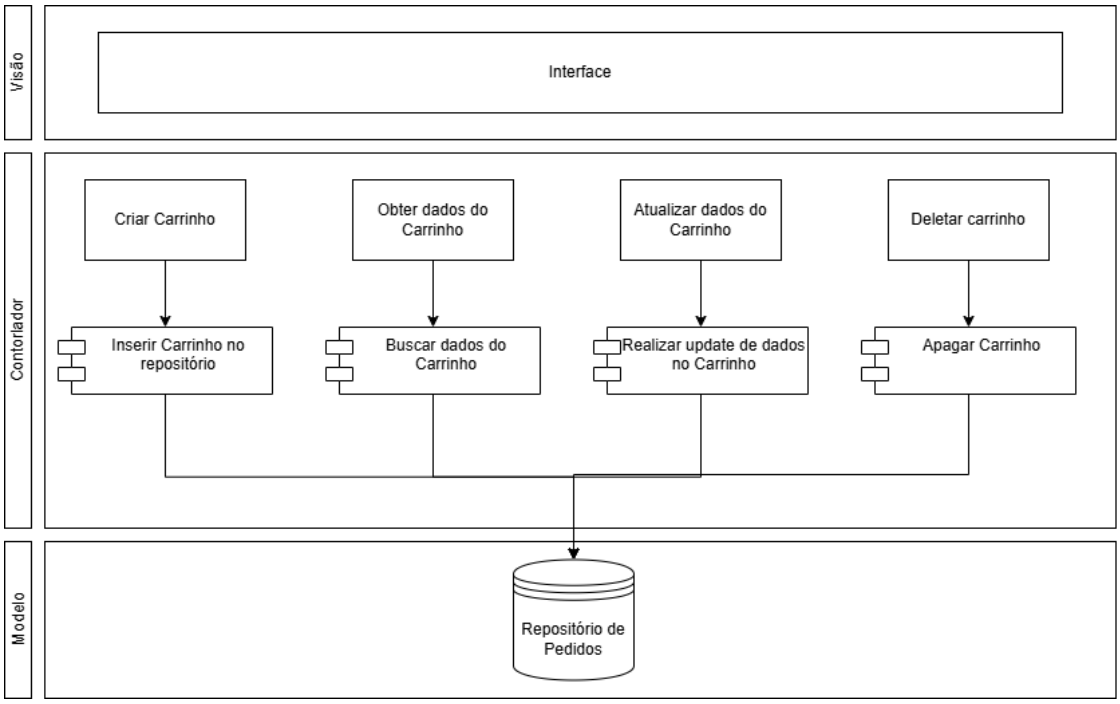
Figura 4: Representação MVC de um sistema de *e-commerce* genérico



Fonte: Próprio autor

Por sua vez a figura 5 já ilustra e exemplifica a arquitetura do ambiente de teste propriamente dito, focando na forma com que há interação nos módulos. Para esta representação foi eleito um *CRUD* para criação, leitura, atualização e deletar um carrinho:

Figura 5: Representação MVC do um sistema de testes sob a ótica de um módulo de gerenciamento de carrinho



Fonte: Próprio autor

3.3.2 Ciclo de operação

O ciclo de operação do ambiente de teste é projetado para replicar fluxos reais de interação entre o *front-end* e o *back-end* de sistemas *headless*. Abaixo, detalhamos as etapas e seu funcionamento no contexto de aplicação levando em conta o módulo de gerenciamento de carrinho:

1. Configuração inicial:
    - O usuário define os cenários de teste no ambiente, incluindo configurações de *endpoints*, simulação de cargas e casos de erro.
    - Ferramentas de orquestração automatizam a criação do ambiente, como por exemplo um *docker*.
2. Execução dos testes:
    - A camada de visão inicia solicitações para os *endpoints* do modelo, simulando interações reais de usuários.
    - O controlador coordena as respostas, processando casos como dados incompletos, latência e timeouts.

3. Coleta de resultados:

    - Os registros de eventos (*logs*) são gerados automaticamente, contendo informações detalhadas sobre cada solicitação e resposta, tempos de execução e erros encontrados.

4. Análise e iteração:

    - Os resultados são analisados para identificar falhas ou inconsistências entre o *front-end* e *back-end*.
    - As configurações do ambiente podem ser ajustadas para simular novos cenários ou refinar os testes existentes.

3.4 Implementação

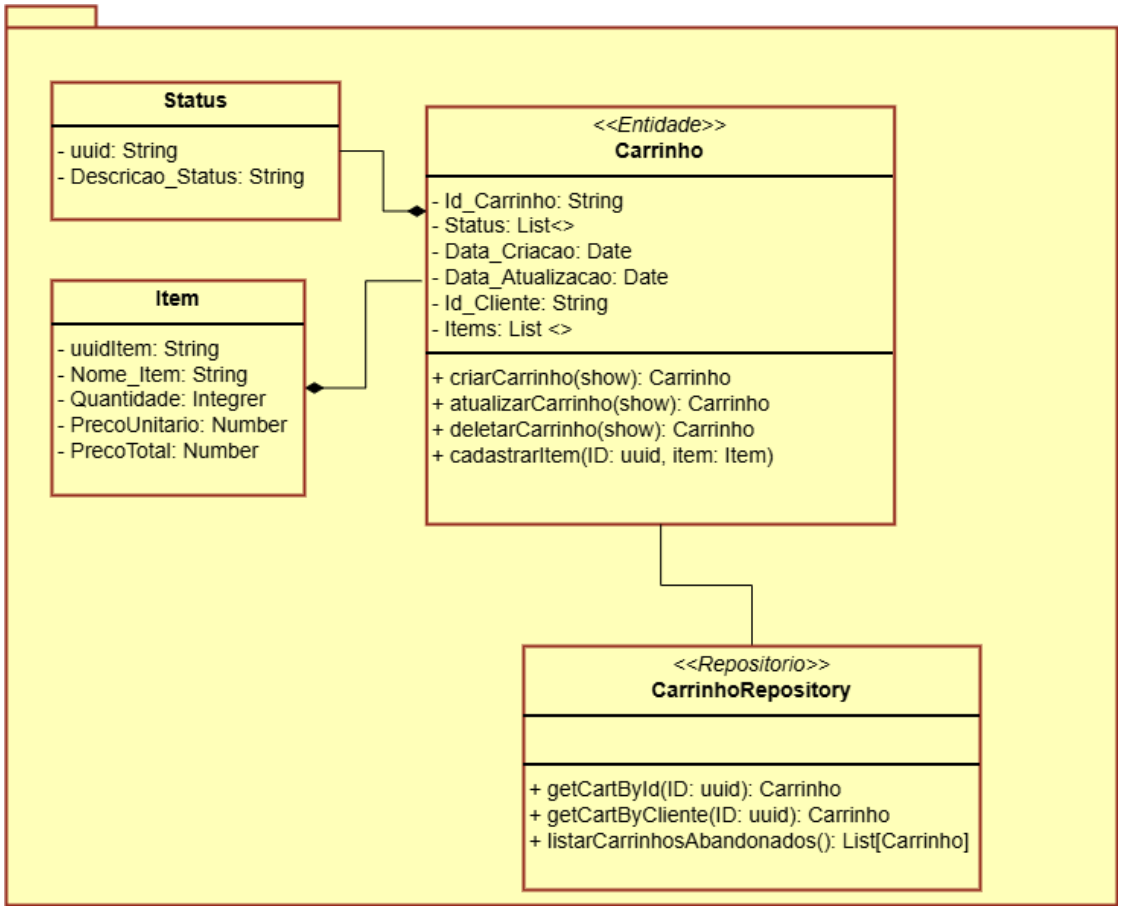
A presente seção demonstra um pouco da forma de como foi implementada a arquitetura para um ambiente de testes.

3.4.1 Descrição tecnológica

Conforme descrito ao longo do capítulo 3, optou-se por uma abordagem de uma arquitetura representada em MVC e como forma de apoio à concepção da arquitetura, foi elegido um módulo ”genérico”de gerenciamento de carrinho, que apresenta uma aplicação de CRUD (*Create, Read, Update, Delete*) no sistema a ser testado e avaliado.

Com base no módulo, foi elaborado de forma simples e superficial, um diagrama de classes para apresentação da arquitetura do ponto de vista da informação, de forma que agrega e facilita o entendimento de como o ambiente de testes pode estar interagindo com o sistema a ser testado em si. A figura 6 traz esta representação:

Figura 6: Ponto de Vista de Informação: Módulo de gerenciamento de Carrinho



Fonte: Próprio autor

Pensado ainda em uma forma de deixar de uma maneira mais palpável o que está sendo dito, e visto que o *back-end* elegido utiliza *NodeJS* e o *front-end React*, foram escolhidos *Frameworks* padrão de mercado, como por exemplo *Jest*, *SonnarQube* e *Jenkins*. Ainda seguindo por esta direção, o módulo de carrinho possui relevância para negócios, uma vez que é o início do processo de conversão de venda, ou seja, de geração de receitas.

Considerando o diagrama de classes apresentado, os testes foram modelados de forma com que a resposta da parte do *back-end* fosse totalmente *mockada* e preparada para respeitar os possíveis cenários identificados como parte do requisito, incluindo também os cenários de erros.

3.4.2 Resultados

No contexto da arquitetura MVC adotada neste trabalho, a sistemática de teste foi desenvolvida com base em padrões amplamente aceitos na indústria de software, como teste de integração e uso de *mock objects*, para assegurar a validação de interações entre os componentes Modelo, Visão e Controlador.

Os testes de integração foram realizados para verificar a comunicação entre as diferentes camadas da arquitetura MVC. Por exemplo, foi validado que as requisições do *front-end* ao *back-end* retornam os dados esperados no formato correto.

Mock objects foram implementados para simular comportamentos do *back-end* durante o teste das funcionalidades do *front-end*. Esse padrão foi essencial para avaliar a consistência do *front-end*, mesmo em cenários onde o *back-end* estava indisponível ou retornava erros intencionais.

Para validar o comportamento do módulo de gerenciamento de carrinho, o padrão de teste de integração foi aplicado em conjunto com *mock objects*, simulando respostas da API para cenários de sucesso, falha e dados incompletos. Isso garantiu que a camada Controlador lidasse corretamente com diferentes tipos de respostas antes de enviá-las para a camada de Visão.

O uso de padrões de teste como *mock objects* e testes de integração não apenas assegurou a confiabilidade das interações no ambiente de teste, mas também proporcionou uma base reutilizável para a validação de futuras funcionalidades, adaptações do sistema e escalabilidade para distintas plataformas.

4 CONSIDERAÇÕES FINAIS

Ao longo deste trabalho, buscou-se desenvolver e documentar uma proposta de ambiente de teste simulado baseado em uma arquitetura MVC, voltado para sistemas que adotam a abordagem *headless*. Embora o objetivo inicial de explorar os fundamentos e propor uma estrutura escalável tenha sido cumprido, é importante reconhecer que o nível de detalhamento e implementação foi limitado, focando mais na concepção teórica e na validação conceitual do ambiente.

Durante o estudo, foram destacadas as vantagens de adotar arquiteturas modulares, como o MVC, para ambientes de teste, mas a aplicação de *frameworks* e ferramentas específicas foi tratada de forma superficial. Isso reflete as limitações do trabalho, que, devido a restrições de tempo e escopo, priorizou a elaboração de um modelo conceitual em detrimento de uma implementação prática mais abrangente.

De toda forma, a utilização de um módulo CRUD como exemplo aplicado demonstrou-se uma abordagem valiosa para ilustrar o conceito de arquitetura de teste no contexto da monografia. A implementação de um módulo de gerenciamento de carrinho permitiu explorar, de maneira prática e simplificada, como as interações típicas entre *front-end* e *back-end* podem ser representadas e testadas em um ambiente simulado. Essa escolha facilitou a demonstração de como um ambiente de teste pode reproduzir cenários reais de uso, mesmo que em menor escala, reforçando a viabilidade da proposta e evidenciando os benefícios da arquitetura MVC para modularidade e escalabilidade. Apesar das limitações inerentes ao escopo do trabalho, essa aplicação prática agregou valor ao estudo, contribuindo para a validação parcial do modelo teórico apresentado.

Para trabalhos futuros, sugere-se investigar estratégias para garantir que os dados simulados (*mock data*) estejam sempre atualizados em relação ao estado real do *back-end* em desenvolvimento. Essa é uma questão crítica, pois *mocks* desatualizados podem comprometer a eficácia dos testes ao introduzir inconsistências que não refletem o comportamento real do sistema. Algumas possíveis abordagens incluem a automatização do processo de sincronização de *mocks* por meio de *pipelines* de integração contínua e entrega contínua, a integração com ferramentas que geram dados simulados a partir de esquemas de API atualizados, ou o uso de contratos automatizados para validar a compatibilidade entre os *mocks* e o *back-end*. Essa linha de pesquisa pode contribuir significativamente para a confiabilidade de ambientes de teste

simulados, sobretudo para cenários similares ao apresentando no presente trabalho.

REFERÊNCIAS

Associação Brasileira de Comércio Eletrônico (ABComm). **Participação no Varejo**. 2024. Acesso em: 15 out. 2024. Disponível em: <https://dados.abcomm.org/participacao-no-varejo>.

ENGINE, W. **The State of Headless 2024 - Defining the Future of Digital Engagement**. 2024. Disponível em: <https://wpengine.com/blog/state-of-headless-2024/>.

PRESSMAN, R. S. **Engenharia de Software: Uma Abordagem Prática**. 9ª. ed. São Paulo: McGraw-Hill, 2014.

SOMMERVILLE, I. **Engenharia de Software**. 10ª. ed. São Paulo: Pearson Education, 2020.

THU, D. T. H.; QUANG, L. D.; NGUYEN, D.-A.; HUNG, P. N. A method of automated mock data generation for restful api testing. In: **2022 RIVF International Conference on Computing and Communication Technologies (RIVF)**. [S.l.: s.n.], 2022. p. 376–381.

TIWARI, D.; MONPERRUS, M.; BAUDRY, B. Mimicking production behavior with generated mocks. **IEEE Transactions on Software Engineering**, v. 50, n. 11, p. 2921–2946, 2024.

VALENTE, M. T. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**. [S.l.]: Editora: Independente, 2020.