



CENTRO DE PESQUISA E DESENVOLVIMENTO TECNOLÓGICO EM INFORMÁTICA E
ELETROELETRÔNICA DE ILHÉUS
CEPEDI



Relatório de Desenvolvimento : E-commerce ShoppingStore

João Vitor Nascimento Ramos

Ilhéus-BA

2024



CENTRO DE PESQUISA E DESENVOLVIMENTO TECNOLÓGICO EM INFORMÁTICA E
ELETROELETRÔNICA DE ILHÉUS
CEPEDI

Relatório de Desenvolvimento : E-commerce ShoppingStore

Trabalho apresentado como requisito de avaliação do módulo de Java Avançado da Residência de Software TIC18.

Autor: João Vitor Nascimento Ramos

Professor: Rogério Oliveira

Tutor: Marcelo Silva

Ilhéus-BA

2024

Lista de ilustrações

Figura 1 – Primeira versão do banco dados relacional	13
Figura 2 – Levantamento do Product backlog	14
Figura 3 – Divisão Inicial das atividades por <i>sprint</i>	16
Figura 4 – Modelo do banco de dados após a sexta sprint	27
Figura 5 – Diagrama de sequência para registro de usuário	31
Figura 6 – Diagrama de sequência para registro de usuário	32
Figura 7 – Diagrama de sequência para login de usuário	33
Figura 8 – Diagrama de sequência para envio de email para recuperação de senha	34
Figura 9 – Diagrama de sequência para redefinição de senha	35
Figura 10 – Diagrama de sequência para registrar nova brand	36
Figura 11 – Diagrama de sequência para buscar uma brand por ID	37
Figura 12 – Diagrama de sequência para buscar todas as brands	38
Figura 13 – Diagrama de sequência para atualizar brand	39
Figura 14 – Diagrama de sequência para desabilitar brand	40
Figura 15 – Diagrama de sequência para registrar nova categoria	41
Figura 16 – Diagrama de sequência para buscar uma brand por ID	42
Figura 17 – Diagrama de sequência para buscar todas as categorias	43
Figura 18 – Diagrama de sequência para atualizar category	44
Figura 19 – Diagrama de sequência para desabilitar category	45
Figura 20 – Diagrama de sequência para registrar novo atributo de categoria	47
Figura 21 – Diagrama de sequência para registrar nova compra	54
Figura 22 – Exemplo da estrutura de <i>packages</i>	59
Figura 23 – Exemplo de estrutura do <i>package</i> db.migration	60
Figura 24 – Exemplo de estrutura do <i>package</i> db.data	60
Figura 25 – Exemplo de onde devem estar as validações	68

Lista de tabelas

Tabela 1 – Distribuição das Subequipes	9
Tabela 2 – Cronograma das Sprints	15
Tabela 3 – Legenda da Figura 3	16
Tabela 4 – Frequência dos colaboradores na primeira <i>sprint planning meet</i>	17
Tabela 5 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 1</i>	17
Tabela 6 – Frequência dos colaboradores na segunda <i>sprint planning meet</i>	19
Tabela 7 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 2</i>	19
Tabela 8 – Frequência dos colaboradores na terceira <i>sprint planning meet</i>	21
Tabela 9 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 3</i>	21
Tabela 10 – Frequência dos colaboradores na quarta <i>sprint planning meet</i>	22
Tabela 11 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 4</i>	22
Tabela 12 – Frequência dos colaboradores na quinta <i>sprint planning meet</i>	24
Tabela 13 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 5</i>	24
Tabela 14 – Frequência dos colaboradores na sexta <i>sprint planning meet</i>	25
Tabela 15 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 6</i>	26
Tabela 16 – Frequência dos colaboradores na sétima <i>sprint planning meet</i>	28
Tabela 17 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 7</i>	28
Tabela 18 – Frequência dos colaboradores na oitava <i>sprint planning meet</i>	29
Tabela 19 – Distribuição das Tarefas entre as Subequipes na <i>Sprint 8</i>	29

Lista de Exemplos de Códigos

A.1	Configurações do Banco de Dados e Segurança	58
A.2	Exemplo de ErrorHandler	62
A.3	Exemplo de Entidade Patient com Lombok e Anotações JPA	63
A.4	Exemplo de Construtor de Entidade Product com Dados Adicionais	64
A.5	Exemplo de DTO para Registro de Categoria	64
A.6	Exemplo de DTO para Detalhes de Categoria	64
A.7	Exemplos de Repositórios	65
A.8	Exemplo de Serviço para a Entidade <i>Patient</i>	66
A.9	Serviço de Agendamento	67
A.10	Exemplo de interface de validação	68
A.11	Classe de Validação para Atualização de Paciente	69
A.12	Exemplo de controller	70
A.13	Exemplo de teste de unidade utilizando o Faker	72
A.14	Exemplo de teste de repositório com anotações	74
A.15	Exemplo de teste de serviço com mocks usando o Mockito	75
A.16	Exemplo de teste de controllers com o MockMvc	76

Sumário

	Lista de Exemplos de Códigos	5
1	INTRODUÇÃO	1
1.1	Apresentação dos Integrantes da Equipe	1
1.2	Apresentação do Problema	3
1.3	Objetivos	3
1.4	Identificação do pontos negativos no código legado	4
1.5	Planejamento de regra de negócios com melhoria	4
2	GERENCIAMENTO DE PROJETO	6
2.1	Metodologia de gerenciamento de projeto	6
2.2	Ferramenta de gerenciamento de projeto	6
2.2.1	Definição das etiquetas da ferramenta para gerenciamento de projeto	7
2.3	Estratégia para Versionamento do Sistema	7
2.3.1	Gestão de Riscos	7
2.4	Definição de Regras para Interoperabilidade de Grupo	8
2.4.1	Definição de estratégia para divisão das atividades em subequipes	8
3	DESENVOLVIMENTO DO PROJETO	10
3.1	Planejamento Inicial	10
3.1.1	Banco de Dados Legado	10
3.1.2	Alterações nas Entidades ao Migrar para o PostgreSQL	12
3.1.3	Levantamento do Product Backlog	13
3.1.4	Planejamento das <i>sprints</i>	14
3.2	Execução	16
3.2.1	Sprint 1	17
3.2.1.1	Sprint Planning Meeting	17
3.2.1.2	Resultados	17
3.2.2	Sprint 2	18
3.2.2.1	Sprint Planning Meeting	18
3.2.2.2	Resultados	19
3.2.3	Sprint 3	20
3.2.3.1	Sprint Planning Meeting	20
3.2.3.2	Resultados	21
3.2.4	Sprint 4	22
3.2.4.1	Sprint Planning Meeting	22

3.2.4.2	Resultados	23
3.2.5	<i>Sprint 5</i>	24
3.2.5.1	Sprint Planning Meeting	24
3.2.5.2	Resultados	24
3.2.6	<i>Sprint 6</i>	25
3.2.6.1	Sprint Planning Meeting	25
3.2.6.2	Resultados	26
3.2.7	<i>Sprint 7</i>	27
3.2.7.1	Sprint Planning Meeting	27
3.2.7.2	Resultados	28
3.2.8	<i>Sprint 8</i>	29
3.2.8.1	Sprint Planning Meeting	29
3.2.9	Resultados	29
4	FUNCIONALIDADES DESENVOLVIDAS	31
4.1	Segurança	31
4.1.1	Registro de usuário	31
4.1.2	Ativação do usuário	32
4.1.3	Login	33
4.1.4	Recuperação de senha	34
4.1.4.1	Envio de email para recuperação	34
4.1.4.2	Redefinição de senha	35
4.2	Brand	36
4.2.1	Registrar nova Brand	36
4.2.2	Buscar Brand por ID	37
4.2.3	Buscar todas as Brands	38
4.2.4	Atualizar Brand	39
4.2.5	Desabilitar Brand	40
4.3	Category	41
4.3.1	Registrar nova Category	41
4.3.2	Buscar Category por ID	42
4.3.3	Buscar todas as categories	43
4.3.4	Atualizar Category	44
4.3.5	Desabilitar Category	45
4.3.6	Buscar categorias por nome	46
4.4	PossibleFacets	47
4.4.1	Registrar Atributo de categoria	47
4.4.2	Buscar Todas as PossibleFacets	48
4.4.3	Buscar PossibleFacets por ID	48
4.4.4	Atualizar PossibleFacets	48

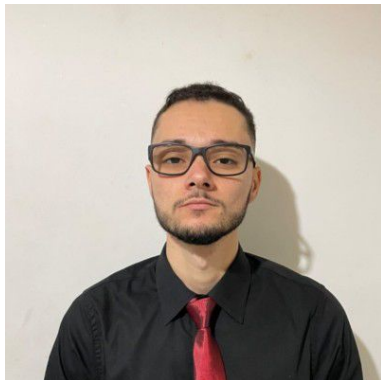
4.4.5	Desabilitar PossibleFacets	48
4.4.6	Buscar possibleFacets por categoria	48
4.5	Product	49
4.5.1	Registrar Produto	49
4.5.2	Listar Produtos	49
4.5.3	Buscar Produto por ID	49
4.5.4	Buscar Produtos por Categoria	49
4.5.5	Atualizar Produto	49
4.5.6	Desabilitar Produto	50
4.6	ProductAttributes	50
4.6.1	Registrar ProductAttribute	50
4.6.2	Listar ProductAttributes	50
4.6.3	Listar ProductAttributes Desabilitados	50
4.6.4	Buscar ProductAttributes por ID	50
4.6.5	Buscar ProductAttributes por Produto	51
4.6.6	Buscar ProductAttributes Desabilitados por Produto	51
4.6.7	Atualizar ProductAttribute	51
4.6.8	Desabilitar ProductAttribute	51
4.7	ProductRating	52
4.7.1	Registrar Avaliação de Produto	52
4.7.2	Listar Avaliações de Produto	52
4.7.3	Buscar Avaliação de Produto por ID	52
4.7.4	Buscar Avaliações por Produto	52
4.7.5	Buscar Avaliações por Usuário	52
4.7.6	Atualizar Avaliação de Produto	53
4.7.7	Desabilitar Avaliação de Produto	53
4.8	Purchase	54
4.8.1	Realizar Compra	54
4.8.2	Cancelar Compra	55
4.9	Payments	56
4.9.1	Registrar Pagamento	56
4.9.2	Listar Todos os Pagamentos	56
4.9.3	Listar Pagamentos por Usuário	56
4.9.4	Desabilitar Pagamento	56
5	CONCLUSÃO	57
	APÊNDICE A – REGRAS PARA ESCRITA E ORGANIZAÇÃO DE	
	CÓDIGO	58
A.1	Informações sensíveis	58

A.2	Estrutura de <i>Packages</i>	58
A.3	<i>Migrations</i>	59
A.4	<i>Seeders</i>	60
A.5	Auditoria	61
A.6	Controle de Exceções	61
A.7	Cacheable	62
A.8	Entidades	63
A.8.1	Construtores de Entidades	63
A.8.2	Construtores de Entidades que possuem composição	63
A.9	Mensagens de erro nos DTO's	64
A.10	Repositorys	65
A.11	Services	65
A.11.1	Services de Classes com composição	67
A.11.2	Localização das Classes de Validação	67
A.11.3	Interface de validação	67
A.11.4	Classes que Implementam as Interfaces de Validação	68
A.12	Controllers	69
A.13	Testes	71
A.13.1	Configurações do ambiente de teste	71
A.14	Padrões de teste	72
A.14.1	Testes de Repository	73
A.14.2	Testes de Service	74
A.14.3	Testes de Controllers	75

1 Introdução

1.1 Apresentação dos Integrantes da Equipe

- João Vitor Nascimento Ramos (Líder Técnico)



- Graduado em Engenharia Elétrica pela UESC- Universidade Estadual de Santa Cruz
- Comprometido em explorar profundamente os princípios e práticas de design de REST APIs, visando criar interfaces de comunicação eficientes e flexíveis entre sistemas.

- Ian Rodrigues Alexandrino



- Cursando Sistemas de Informação na UNEX - Itabuna
- Focado em me aprimorar no Back-End e aprender cada vez mais sobre REST APIs

- Leonardo Ribeiro Barbosa Santos



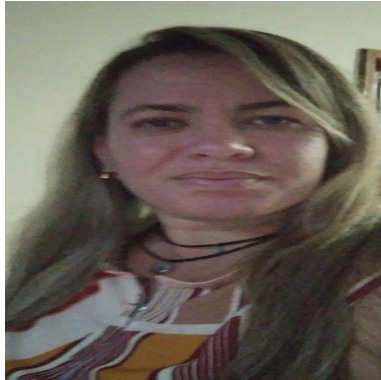
- Graduado em Superior Tecnólogo em Redes de Computadores pela Pitágoras Unopar Anhanguera, concluído em 2023.
- Atualmente cursando Técnico em Manutenção e Suporte de Informática pelo IFSuldeMinas.

- Murilo Carlos Novais



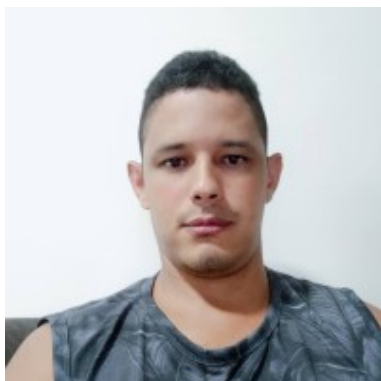
- Engenheiro Eletricista, formado pela Universidade Estadual de Santa Cruz - Ilhéus
- Desenvolvedor java

- Renata C. de Oliveira Moreno



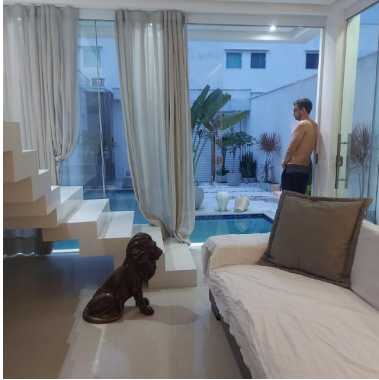
- Formada em Análise e Desenvolvimento de Sistemas
- Entusiasta de tecnologia e inovação, busco novos desafios na indústria de TI, acreditando que o sucesso vem para aqueles que nunca param de aprender.

- Danilo da Conceição Santos



- Profissional formado como Técnico em Informática pelo Instituto Federal da Bahia (IFBA).
- Atualmente cursando Análise e Desenvolvimento de Sistemas pela Estácio de Sá..

- Alexandre de Souza Amaral



– Não foram apresentadas informações..

1.2 Apresentação do Problema

O problema consiste na atualização de um sistema de *e-commerce*, estabelecido há cinco anos, que atualmente lida com a comercialização de relógios, *laptops* e *smartphones*. Essa solicitação surge em consonância com as novas diretrizes do modelo de negócios da empresa e visa manter sua competitividade no mercado. A modernização proposta implica uma transição estratégica, direcionando o sistema para a venda de produtos tecnológicos de alto valor agregado, como os servidores das marcas Dell e Positivo.

Diante desse contexto, torna-se crucial reestruturar e atualizar o sistema de *e-commerce* para atender aos padrões tecnológicos contemporâneos. Essas melhorias incluirão a atualização da versão do *framework* Java, a otimização do *backend* e a remodelagem do banco de dados. Garantindo, dessa forma, a segurança do sistema, utilizando ferramentas que fornecem mais escalabilidade, o que expande a competitividade no dinâmico cenário de tecnologia B2B.

1.3 Objetivos

- Atualizar, adaptar e otimizar um *e-commerce* desenvolvido há cinco anos
- Estimular o trabalho em equipe, promover a comunicação eficaz e desafiar os participantes a resolver problemas complexos
- Demonstrar profundo entendimento dos conceitos avançados de *Java* (abordados na residência)
- Adotar boas práticas de desenvolvimento e mostrar capacidade para atender a novas demandas tecnológicas
- Demonstrar criatividade e proatividade na busca por soluções inovadoras

1.4 Identificação do pontos negativos no código legado

- Versão do *Spring Boot*

A versão atual do *framework* no projeto é a 2.1.1. Embora tenha sido estável e amplamente utilizada no passado, desde dezembro de 2019 não recebe mais atualizações de segurança e correções de *bugs*. Isso implica que o projeto pode estar vulnerável a falhas conhecidas que foram solucionadas em versões mais recentes do *framework*.

Além disso, as versões mais recentes do *Spring Boot* apresentam uma série de melhorias e novos recursos que podem enriquecer o projeto. Estas atualizações englobam melhorias de desempenho, suporte a tecnologias e bibliotecas emergentes, bem como uma integração mais aprimorada com os ecossistemas modernos de desenvolvimento de software. Outro ponto relevante a se considerar é a compatibilidade com as bibliotecas e *frameworks* de terceiros que são utilizados no projeto.

- Uso de banco de dados Mongo DB

A falta de um esquema estruturado e a flexibilidade no modelo de dados *NoSQL* podem resultar em dificuldades na manutenção e no entendimento da estrutura dos dados. Isso pode levar a problemas de consistência e dificuldades na realização de consultas complexas que exigem operações de junção e agregação.

Além disso, a ausência de transações *ACID* em muitos bancos de dados *NoSQL* pode comprometer a integridade dos dados em ambientes onde a consistência é crítica.

- Versões desatualizadas de dependências

A utilização de versões desatualizadas de dependências em um projeto legado pode acarretar consequências graves. Em primeiro lugar, tais versões frequentemente deixam de receber atualizações de segurança e correções de *bugs*, deixando o sistema vulnerável a ataques de invasores. Além disso, versões desatualizadas podem provocar incompatibilidades com outras partes do sistema ou com bibliotecas mais recentes, resultando em falhas e instabilidades.

1.5 Planejamento de regra de negócios com melhoria

Para esse projeto foram levantadas as seguintes melhorias:

- Atualização da versão do *Spring Boot*

- Atualização das dependências.
- Re-modelagem do sistema para um banco de dados relacional PostgreSQL.
- Inserção de novas dependências ao projeto
- Aplicação de padrões de projeto nas verificações de dados em transições de cadastro, *update* e *delete*.
- Aplicação de um sistema de auditoria com *IpAddressInterceptor* para ter uma tabela de *logs* no banco de dados.
- Expansão do uso de *spring security* para funcionalidades de segurança do sistema.

2 Gerenciamento de projeto

2.1 Metodologia de gerenciamento de projeto

Foi optado pela metodologia *Scrum* para o desenvolvimento do projeto, principalmente devido à sua abordagem ágil e iterativa. O *Scrum* é uma estrutura de gerenciamento de projetos que enfatiza a entrega contínua de valor ao cliente, priorizando a colaboração da equipe, a flexibilidade e a capacidade de resposta às mudanças.

Existem algumas razões específicas pelas quais foi escolhido o *Scrum*:

- **Flexibilidade:** Permite adaptação fácil às mudanças de requisitos através de iterações curtas, o que é crucial quando se tem pouco tempo para entrega do produto e a equipe nunca trabalhou junta antes.
- **Foco no cliente:** Garante entrega de valor regularmente, atendendo às necessidades do cliente, mesmo em situações de pressão de tempo.
- **Transparência e colaboração:** Promove transparência e colaboração através de reuniões regulares, facilitando a integração da equipe e a identificação rápida de possíveis problemas.
- **Melhoria contínua:** Estimula reflexão e aprendizado constante através de retrospectivas, permitindo que a equipe ajuste seu processo de trabalho para se tornar mais eficiente ao longo do tempo, apesar da falta de experiência prévia de trabalho conjunto.
- **Controle e previsibilidade:** Oferece estrutura clara para controle e previsibilidade do desenvolvimento, ajudando a equipe a gerenciar e priorizar o trabalho de forma eficaz, mesmo em um contexto de novidade e prazos apertados.

2.2 Ferramenta de gerenciamento de projeto

A ferramenta escolhida para o gerenciamento do projeto foi o *Trello*, devido à sua simplicidade, flexibilidade e capacidade de promover a colaboração da equipe. O *Trello* oferece uma interface intuitiva que facilita a criação, organização e acompanhamento de tarefas, tornando-o ideal para uma equipe que enfrenta prazos apertados e está se familiarizando com novas práticas de trabalho.

2.2.1 Definição das etiquetas da ferramenta para gerenciamento de projeto

Foram definidas as seguintes etiquetas para o gerenciamento de projeto, cada uma associada a uma cor específica:

- **Criação**: Utilizada para tarefas relacionadas à criação de arquivos, como classes, records, interfaces, etc.
- **Teste**: Reservada para atividades de teste dos arquivos criados.
- **Planejamento**: Destinada ao planejamento de atividades do projeto.
- **Reunião**: Designada para as reuniões diárias de acompanhamento do progresso do projeto.
- **Relatório**: Utilizada para tarefas relacionadas à criação do documento de relatório.

Estas cores foram escolhidas para facilitar a identificação e organização das atividades ao longo do projeto.

2.3 Estratégia para Versionamento do Sistema

Foi definido o uso do *Git* como sistema de controle de versão, com a criação de um repositório no *GitHub*. Todos os participantes foram adicionados como colaboradores no repositório, permitindo que contribuíssem para o projeto. A estratégia adotada envolve a criação de *branches* individuais para cada tarefa ou funcionalidade, que serão posteriormente '*mergeadas*' na *branch* "desenvolvimento". Após revisão e testes, as alterações serão então '*mergeadas*' na *branch* "main", que representa a versão estável do sistema.

Essa estratégia proporciona um fluxo de trabalho organizado e colaborativo, permitindo o desenvolvimento de novas funcionalidades de forma isolada e segura, garantindo a estabilidade do código na *branch* principal do repositório.

2.3.1 Gestão de Riscos

Após uma análise minuciosa de riscos que possam interferir no desenvolvimento do trabalho, elencamos alguns pontos críticos e elaboramos medidas de contingência:

- **Interoperabilidade entre grupo**: Como forma de garantir que qualquer membro da equipe consiga dar continuidade ao trabalho de outro, sem que haja desperdício de tempo com análise de código, foi estabelecido um *Framework Interop Group* e adotado padrões de escrita de código.

- **Exceder prazo de entrega:** A metodologia de gerenciamento de projeto adotada no desenvolvimento desta aplicação permite que a execução das atividades sejam divididas em *sprints*. Com base nisso, adotamos a estratégia de programação em pares dividindo nossa força de trabalho em sub-equipes de 2 pessoas como forma de agilizar a execução das atividades e contingenciar a entrega caso um membro da equipe fique indisponível ou simplesmente não consiga executar a tarefa. Somado a isso, são realizadas reuniões diárias onde cada um dos participantes do grupo relata como foi o desenvolvimento de suas atividades e se houve alguma dificuldade ou impedimento na execução da tarefa.

2.4 Definição de Regras para Interoperabilidade de Grupo

Nesta seção, serão apresentadas as regras de interoperabilidade aplicadas ao grupo. Essas regras têm como objetivo estabelecer um padrão de desenvolvimento para promover consistência, qualidade e colaboração no projeto.

- Todos os membros devem seguir as convenções de codificação definidas para garantir consistência no estilo de código.
- As revisões de código devem ser realizadas regularmente para garantir a qualidade do código e identificar possíveis melhorias.
- Todas as contribuições devem ser documentadas adequadamente para facilitar a compreensão e manutenção do código no futuro.

Regras de escrita e organização de código foram estabelecidas para assegurar que os contribuidores mantenham um padrão de codificação consistente, permitindo uma transição suave entre diferentes partes do código e facilitando a compreensão e continuidade do desenvolvimento, independentemente do autor original. Essas regras estão detalhadas no Apêndice [A](#).

2.4.1 Definição de estratégia para divisão das atividades em subequipes

Foi decidido adotar a prática de programação em pares devido aos diversos benefícios que ela oferece. Entre esses benefícios estão o aprendizado compartilhado, a redução de erros, o aprimoramento do design de código e a eficiente resolução de problemas. Além disso, essa abordagem proporciona maior flexibilidade na distribuição de tarefas entre os membros da equipe.

Se um membro de uma subequipe não puder realizar uma atividade específica, eles têm a capacidade de redistribuí-la entre si. Por fim, caso ambos os membros de uma

subequipe encontrem dificuldades significativas, eles podem buscar orientação adicional do líder técnico.

Para implementar essa abordagem, foram definidas quatro subequipes, sendo as três primeiras compostas por dois integrantes cada e a quarta composta apenas pelo líder técnico. A distribuição das subequipes está apresentada na Tabela 1.

A distribuição das tarefas entre as subequipes foi realizada de maneira dinâmica e baseada na performance de cada uma delas ao longo do projeto. As equipes foram avaliadas regularmente, conforme sua eficiência e produtividade. A alocação de novas atividades foi feita de maneira a otimizar o progresso geral do trabalho. Este modelo permite ajustes flexíveis, garantindo que cada subequipe contribua de acordo com suas capacidades e habilidades, maximizando assim a eficácia do projeto.

Tabela 1 – Distribuição das Subequipes

Sub-equipe	Integrantes	
1	Leonardo Ribeiro Barbosa Santos	Alexandre de Souza Amaral
2	Danilo da Conceição Santos	Murilo Carlos Novais
3	Ian Rodrigues Alexandrino	Renata C. de Oliveira Moreno
4	João Vitor Nascimento Ramos	

3 Desenvolvimento do Projeto

Nesse Capítulo será apresentado a evolução temporal do desenvolvimento do projeto desde o planejamento até a execução de cada sprint, bem como as alterações do plano inicial e suas adaptações.

3.1 Planejamento Inicial

Assim que nos deparamos com o sistema legado, foi feita uma análise das entidades e da regra de negócio para entender seu funcionamento. Partindo disso, fizemos um planejamento das novas funcionalidades do produto e criamos modelos que adequam o sistema a nova realidade.

3.1.1 Banco de Dados Legado

O sistema legado, utilizava uma abordagem *NoSQL* com o software *MongoDB*, um sistema de gerenciamento de banco de dados orientado a documentos. Identificamos as entidades, e observamos como estavam armazenadas. A seguir apresentamos as entidades e seus respectivos atributos:

- **Category:** Entidade para armazenar as categorias de produtos.

```
private Long id; // Identificador único da categoria
private String name; // Nome da categoria
private List<String> possibleFacets; // Lista de possíveis facetas
// associadas à categoria
```

- **ElasticSearchProduct:** Entidade para armazenar produtos indexados no ElasticSearch.

```
private String id; // Identificador único do produto
private String name; // Nome do produto
private String description; // Descrição detalhada do produto
private BigDecimal price; // Preço do produto
private String sku; // Código SKU do produto
private String imageUrl; // URL da imagem do produto
private Category category; // Categoria do produto
private List<ProductAttribute> productAttributeList; // Lista de atributos
```

```

// do produto
private Integer quantity;           // Quantidade do produto em estoque
private String manufacturer;        // Fabricante do produto
private boolean featured;           // Indica se o produto é destaque
private List<ProductRating> productRating; // Lista de avaliações do produto

```

- **Mail:** Entidade para armazenar informações de emails.

```

private String from;                // Remetente do email
private String to;                  // Destinatário do email
private String content;             // Conteúdo do email
private String subject;             // Assunto do email

```

- **Product:** Entidade para armazenar produtos.

```

private String id;                  // Identificador único do produto
private String name;                // Nome do produto
private String description;          // Descrição detalhada do produto
private BigDecimal price;           // Preço do produto
private String sku;                 // Código SKU do produto
private String imageUrl;            // URL da imagem do produto
private Category category;          // Categoria do produto
private Long categoryId;            // Identificador da categoria
private List<ProductAttribute> productAttributeList; // Lista de atributos
// do produto
private Integer quantity;           // Quantidade do produto em estoque
private String manufacturer;        // Fabricante do produto
private boolean featured;           // Indica se o produto é destaque
private List<ProductRating> productRating; // Lista de avaliações do produto

```

- **ProductAttribute:** Entidade para armazenar atributos dos produtos.

```

private String attributeName;       // Nome do atributo
private String attributeValue;      // Valor do atributo

```

- **ShoppingCart:** Entidade para armazenar carrinhos de compras.

```

private String id;                  // Identificador único do carrinho
// de compras
private Set<ShoppingCartItem> shoppingCartItems; // Conjunto de itens no
// carrinho de compras

```

```
private BigDecimal cartTotalPrice;    // Preço total do carrinho
private int numberOfItems;           // Número de itens no carrinho
private String username;              // Nome do usuário proprietário do
                                     // carrinho
```

- **ShoppingCartItem:** Entidade para armazenar itens no carrinho de compras.

```
private String name;                 // Nome do produto no carrinho
private BigDecimal price;            // Preço do produto no carrinho
```

- **User:** Entidade para armazenar usuários.

```
private String id;                   // Identificador único do usuário
private String email;                // Email do usuário
private String username;             // Nome de usuário
private String password;             // Senha do usuário
private String passwordConfirmation; // Confirmação da senha do usuário
private boolean enabled;             // Indica se o usuário está ativo
```

- **VerificationToken:** Entidade para armazenar tokens de verificação de usuários.

```
private String id;                   // Identificador único do token
private String token;                // Token de verificação
private User user;                   // Usuário associado ao token
private Instant expiryDate;          // Data de expiração do token
```

3.1.2 Alterações nas Entidades ao Migrar para o PostgreSQL

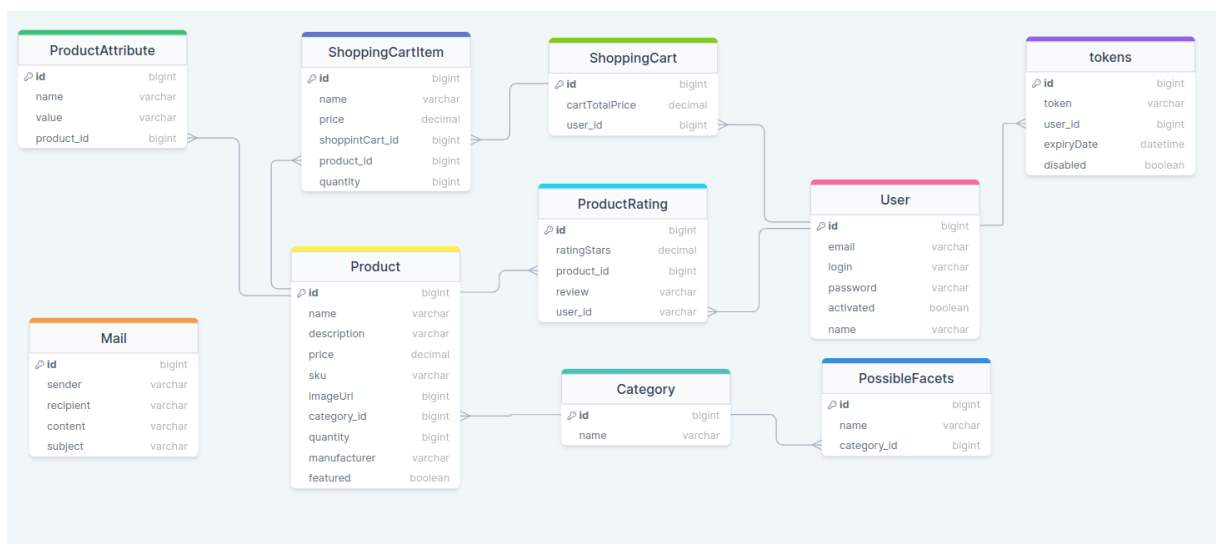
Ao migrar do *MongoDB* para o PostgreSQL, foram realizadas alterações em cada entidade para adequar a base de dados ao modelo relacional e também viabilizar a expansão do sistema para novos produtos no futuro.

- **Category:** O atributo name da entidade Category foi mantido, porém a lista possibleFacets foi removida. Esses dados foram armazenados na tabela PossibleFacets com uma chave estrangeira.
- **ElasticSearchProduct:** Inicialmente, esta entidade não foi modelada no novo banco de dados, pois sua implementação foi prevista para ser realizada posteriormente, caso houvesse tempo disponível.
- **Mail:** Permaneceu sem alterações.

- **Product:** Os atributos foram mantidos e as listas das entidades ProductAttribute e ProductRating tiveram suas próprias tabelas referenciando Product.
- **ProductAttribute:** Manteve sua estrutura original, agora referenciando um produto.
- **ShoppingCart:** Foi transformada em uma entidade fraca em relação a um User. Além disso, deixou de armazenar o número de itens diretamente, visto que essa informação pode ser facilmente obtida por meio de uma busca de contagem no banco de dados pelos itens referenciados em ShoppingCartItem.
- **ShoppingCartItem:** Foi remodelada para armazenar o nome do produto no momento da compra, o preço, a quantidade e referenciar o produto.
- **User:** O atributo passwordConfirmation foi removido, pois essa validação deve ser feita no *frontend*.
- **VerificationToken:** A entidade foi renomeada para tokens e foi inserida uma referencia ao usuário, além de um atributo para desabilitar o token.

O modelo do banco de dados relacional está ilustrado na Figura 1.

Figura 1 – Primeira versão do banco dados relacional










































































3.1.3 Levantamento do Product Backlog

O *backlog* do produto é a lista priorizada de trabalho para a equipe de desenvolvimento, derivado do roteiro do produto e seus requisitos. Uma lista de pendências ágil bem organizada não só torna o planejamento da liberação e da iteração (*sprint*) mais fácil, como também transmite tudo em que sua equipe pretende trabalhar e faz com que o tempo da equipe seja fixo.

Nesse contexto, foi realizado um levantamento de ideias para as atividades necessárias em formato *brainstorming*. Em seguida, baseado nos requisitos e no roteiro de desenvolvimento do sistema, foi realizada uma filtragem desse conjunto de ideias e construído um *product backlog* elencando as tarefas a serem realizadas para conclusão do projeto. A Figura 2 ilustra uma abstração dessas ideias com as seguintes etiquetas.

Figura 2 – Levantamento do Product backlog

Configurar Spring Doc 	Implementar package AuditLog  	Implementar Entity e Repository Mail  	Implementar Entity e Repository Category  	Inserir dependencias 
Implementar Package Security  	Implementar Entity e Repository PossibleFacets  	Criar Entity e Repository ProductAttribute  	Criar Entity e Repository Product  	Criar Entity e Repository ShoppingCart  
Criar Entity e Repository ProductRating  	Criar Entity e Repository ShoppingCartItem  	Implementar HandleException  	Criar e testar Service ShoppingCart  	Criar e testar Service ProductAttribute  
Criar e testar Service ShoppingCartItem  	Criar e testar Service ProductRating  	Criar e testar Service Mail  	Criar e testar Service Category  	Criar e testar Service Product  
Criar e testar Service PossibleFacets  	Criar e testar Controller shoppingCartItem  	Criar e testar Controller ShoppingCart  	Criar e testar Controller Category  	Criar e testar Controller ProductAttribute  
Criar e testar Controller Product  	Criar e testar Controller PossibleFacets  	Criar e testar Controller ProductRating  	Implementar Envio de Email Para registro do usuario  	Implementar Envio de Email Para recuperação de senha do usuario  
Implementar Persistencia dos emails enviados  	Comentarios Spring Doc nos Controllers 	Implementar ValidationMessages 	Popular banco de Dados 	Criar e testar Entity Payment  
Criar e testar Service Payment  	Criar e testar Controller Payment  	Configurar Permissão nos controllers  		

3.1.4 Planejamento das *sprints*

O projeto foi apresentado à equipe no dia 13 de maio, marcando o início de uma série de reuniões e planejamentos. A primeira reunião ocorreu às 10 horas do dia 14 de maio, durante a qual o *backlog* foi definido e as *sprints* foram estabelecidas. O planejamento detalhado incluiu a execução de 7 *sprints*, com o objetivo de concluir a solução até o dia 24 de maio.

Após a reunião inicial, que terminou às 12 horas, ficou decidido que seriam realizadas reuniões diárias (*daily*s) em dias úteis, às 14 horas, com previsão de duração de uma hora. Essas reuniões serviram para discutir o progresso, enfrentar impedimentos e planejar os próximos passos. As *daily*s foram fundamentais para manter a equipe alinhada e garantir a continuidade eficiente do projeto, coincidindo com o início de cada nova *sprint*.

Cada *sprint* foi programada para encerrar às 11 horas do dia seguinte, com exceção da primeira, permitindo ao líder técnico revisar o código, analisar a qualidade do trabalho realizado e corrigir eventuais erros, os quais seriam abordados nas reuniões subsequentes.

A Tabela 2 apresenta o número de cada *sprint* e os horários de início e término de cada uma, fornecendo um panorama claro do cronograma estabelecido para o projeto:

Tabela 2 – Cronograma das Sprints

Sprint	Data de Início	Data de Fim
1	14 de maio, 12:00	16 de maio, 11:00
2	16 de maio, 15:00	17 de maio, 11:00
3	17 de maio, 15:00	20 de maio, 11:00
4	20 de maio, 15:00	21 de maio, 11:00
5	21 de maio, 15:00	22 de maio, 11:00
6	22 de maio, 15:00	23 de maio, 11:00
7	23 de maio, 15:00	24 de maio, 11:00

O interstício entre os dias 17 e 20 foi planejado para evitar reuniões durante o fim de semana, conforme a equipe se organizou para não necessitar de encontros nesses dias.

Para garantir a qualidade do trabalho desenvolvido, optamos por realizar sprints curtas, com duração de 24 horas cada. Dessas 24 horas, 20 horas eram dedicadas à produção pelos subgrupos. As 3 horas seguintes eram reservadas para o líder técnico corrigir eventuais bugs e planejar as sprints seguintes, e a última hora era dedicada à reunião diária (*daily*). Essa estratégia foi necessária pelo prazo curto para entrega do produto e para melhor avaliar o desempenho dos integrantes da equipe.

Durante essas 3 horas, o líder técnico também decidia quais assuntos seriam levados para discussão nas reuniões. Isso incluía problemas encontrados na escrita do código, razões para a não entrega de determinadas tarefas e a necessidade de realocação de tarefas entre os membros da equipe. Além disso, era um momento para avaliar o progresso geral do projeto, identificar quaisquer bloqueios que pudessem estar afetando a equipe e ajustar o planejamento conforme necessário para garantir que todos estivessem alinhados com os objetivos da *sprint*.

As atividades do *backlog* foram inicialmente divididas em *sprints*, identificadas por cores na Figura 3, com a legenda correspondente na Tabela 3. A sprint 8 foi designada com menos atividades, criando um espaço de contingência para permitir flexibilidade. Isso garante que eventuais atrasos possam ser tratados sem comprometer o prazo final.

Figura 3 – Divisão Inicial das atividades por *sprint*

Configurar Spring Doc [Progress Bar]	Implementar package AuditLog [Progress Bar]	Implementar Entity e Repository Mail [Progress Bar]	Implementar Entity e Repository Category [Progress Bar]	Inserir dependencias [Progress Bar]
Implementar Package Security [Progress Bar]	Implementar Entity e Repository PossibleFacets [Progress Bar]	Criar Entity e Repository ProductAtributte [Progress Bar]	Criar Entity e Repository Product [Progress Bar]	Criar Entity e Repository ShoppingCart [Progress Bar]
Criar Entity e Repository ProductRating [Progress Bar]	Criar Entity e Repository ShoppingCartItem [Progress Bar]	Implementar HandleException [Progress Bar]	Criar e testar Service ShoppingCart [Progress Bar]	Criar e testar Service ProductAtributte [Progress Bar]
Criar e testar Service ShoppingCartItem [Progress Bar]	Criar e testar Service ProductRating [Progress Bar]	Criar e testar Service Mail [Progress Bar]	Criar e testar Service Category [Progress Bar]	Criar e testar Service Product [Progress Bar]
Criar e testar Service PossibleFacets [Progress Bar]	Criar e testar Controller shoppingCartItem [Progress Bar]	Criar e testar Controller ShoppingCart [Progress Bar]	Criar e testar Controller Category [Progress Bar]	Criar e testar Controller ProductAtributte [Progress Bar]
Criar e testar Controller Product [Progress Bar]	Criar e testar Controller PossibleFacets [Progress Bar]	Criar e testar Controller ProductRating [Progress Bar]	Implementar Envio de Email Para registro do usuario [Progress Bar]	Implementar Envio de Email Para recuperação de senha do usuario [Progress Bar]
Implementar Persistencia dos emails enviados [Progress Bar]	Comentarios Spring Doc nos Controllers [Progress Bar]	Implementar ValidationMessages [Progress Bar]	Popular banco de Dados [Progress Bar]	Criar e testar Entity Payment [Progress Bar]
Criar e testar Service Payment [Progress Bar]	Criar e testar Controller Payment [Progress Bar]	Configurar Permissão nos controllers [Progress Bar]		

Tabela 3 – Legenda da Figura 3

Sprint	Cor
1	[Red]
2	[Yellow]
3	[Green]
4	[Blue]
5	[Orange]
6	[Purple]
7	[Light Purple]
8	[Grey]

3.2 Execução

Conforme planejado, a execução das tarefas foi distribuída em *sprints*. Nesta seção, detalharemos o desenvolvimento de cada *sprint*, incluindo as tarefas concluídas, os desafios enfrentados e as soluções implementadas. Além disso, analisaremos a performance da equipe, destacando pontos fortes e áreas de melhoria. Também discutiremos os tópicos mais importantes que foram abordados nas reuniões diárias, como ajustes no planejamento, impedimentos encontrados e a colaboração entre os membros da equipe.

3.2.1 Sprint 1

3.2.1.1 Sprint Planning Meeting

Durante a reunião de planejamento geral realizada em 14 de maio, foi conduzida também a primeira *sprint* planning meeting (reunião de planejamento de *sprint*). A Tabela 4 apresenta a frequência dos colaboradores durante essa reunião.

Tabela 4 – Frequência dos colaboradores na primeira *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino	x	
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

Até o presente momento, não foi possível entrar em contato com Alexandre, pois ele não me procurou como fizeram outros membros da equipe. Além disso, o número de telefone que eu possuía não estava mais em uso.

Após repassar aos colaboradores as regras para a interoperabilidade do grupo, conforme descrito na Sessão 2.4, e a distribuição das equipes, conforme descrito na Tabela 1, foram alocadas as tarefas conforme a Tabela 5.

Tabela 5 – Distribuição das Tarefas entre as Subequipes na *Sprint* 1

1	2	3	4
Implementar e testar package AuditLog	Inserir dependências	Implementar e testar Entity e Repository Category	Implementar e testar o package Security
Configurar Spring Doc	Implementar e testar Entity e Repository Mail	Implementar e testar Entity e Repository PossibleFacets	

3.2.1.2 Resultados

Após o término da primeira sprint, durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

- **Subequipe 1:** Devido à falta de contato por parte de Alexandre, Leonardo assumiu as responsabilidades. A implementação do AuditLog foi realizada com sucesso, no entanto, o construtor que recebe um DTO como parâmetro não foi incluído. Além disso, o Spring Doc foi configurado com êxito.

- **Subequipe 2:** Danilo assumiu sozinho todas as responsabilidades, uma vez que Murilo ainda não havia iniciado sua participação no código. Murilo justificou que estava com problemas de tempo devido a avaliações no mestrado. Sob a liderança de Danilo, todas as tarefas foram concluídas. No entanto, foi constatado que os testes para a entidade *Mail* e a criação e teste do *repository* correspondente não foram realizados. Além disso, as dependências discutidas na reunião foram devidamente integradas. Destacam-se o SLF4J, que proporciona um logging flexível e eficiente, o Auth0 - JWT para uma autenticação segura baseada em JWT, o Faker para a geração de dados falsos em testes e desenvolvimento, o Rest Assured para a automatização de testes de APIs REST, o Spring DOC para uma documentação eficaz de APIs Spring e o Starter Mail para uma fácil integração de e-mails em aplicativos Spring.
- **Subequipe 3:** Devido a problemas de comunicação, Renata tomou a frente e realizou as tarefas antes da participação de Ian. No entanto, após conversarem, concordaram em equilibrar as responsabilidades na próxima *sprint*, com Ian cuidando das pendências dessa subequipe. Quanto à implementação da entidade *Category*, foi realizada com sucesso. No entanto, não foi incluída a anotação `@Table` para definir o nome da tabela no banco de dados. Em relação a entidade *Possible Facets*, o relacionamento entre ele e a *Category* foi realizado de forma inadequada.
- **Subequipe 4:** A Subequipe 4 entregou com êxito e dentro do prazo o desenvolvimento do package security.

Todos os erros identificados por parte de cada sub-equipe nessa *sprint* foram corrigidos pelo líder técnico. Os testes não implementados foram realocados como tarefas para próxima *sprint*.

3.2.2 Sprint 2

3.2.2.1 Sprint Planning Meeting

Durante a *daily* do dia 16 de maio, onde constou a análise da primeira *sprint*, também foi conduzida a segunda *sprint planning meeting*. A Tabela 6 apresenta a frequência dos colaboradores durante essa reunião.

Durante esta reunião, os colaboradores foram apresentados ao *plugin JaCoCo*, responsável por ilustrar a cobertura de testes. Além disso, foi demonstrado como instalar e utilizar esse *plugin* tanto no *IntelliJ* quanto no *Eclipse*. Também foi compartilhado que a cobertura atual de testes estava em 40%.

Devido à ausência dos testes necessários na primeira *sprint*, foi preciso ajustar o planejamento da segunda pois garantir a qualidade das camadas iniciais é crucial antes de avançar para novos componentes. Como resultado, algumas atividades da segunda *sprint*

Tabela 6 – Frequência dos colaboradores na segunda *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino	x	
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

foram destinadas à realização dos testes pendentes. Após explicar aos colaboradores a importância desses testes e a nova realidade da sprint, foram alocadas as tarefas para a segunda fase do projeto. É importante destacar que, mesmo com a ausência de Alexandre, a quantidade de tarefas para a equipe 1 não foi reduzida, uma vez que Leonardo demonstrou um desempenho satisfatório. A Tabela 7 ilustra a alocação de atividades para a segunda *sprint*.

Tabela 7 – Distribuição das Tarefas entre as Subequipes na *Sprint 2*

1	2	3	4
Testar Entity e Repository PossibleFacets	Implementar e testar Entity e Repository Product	Implementar e testar Entity e Repository ShoppingCart	Implementar e testar Entity e Repository ProductRating
Implementar e testar Entity e Repository ProductAttribute	Testar Entity e Repository Category	Testar Entity e Repository Mail	Implementar e testar Entity e Repository ShoppingCartItem
			Implementar e testar HandleException

Destaca-se nesta etapa a distribuição das atividades de teste para subequipes diferentes daquelas responsáveis pela implementação das classes. Esse procedimento foi realizado com o intuito de verificar a eficácia dos padrões de código adotados, que visavam facilitar a compreensão do código por outros membros da equipe. Além disso, objetivou-se evitar vícios nos testes, promovendo uma avaliação mais imparcial e abrangente.

3.2.2.2 Resultados

Antes da apresentação dos resultados, cabe destacar que durante o período da segunda *sprint*, Renata conseguiu finalmente entrar em contato com Alexandre e compartilhou o número de telefone com a equipe. O líder técnico tomou a iniciativa de entrar em contato com Alexandre, instruindo-o a visualizar os slides gerados durante a reunião e a entrar em contato com Leonardo. Ao mesmo tempo, solicitou a Leonardo que iniciasse o contato com Alexandre. O e-mail de Alexandre foi inserido no Trello, seu número de inserido no grupo *WhatsApp*, entretanto ele não forneceu seu GitHub à equipe e também não manteve comunicação.

Após o término da segunda *sprint*, durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

- **Subequipe 1:** Com a ausência de contato por parte de Alexandre, Leonardo continuou assumindo as responsabilidades. A implementação e os testes da entidade *ProductAttribute* foram realizados com sucesso, juntamente com os testes da entidade *PossibleFacets*. No entanto, os testes ainda não estavam sendo executados em ordem randômica.
- **Subequipe 2:** Com Murilo ainda ocupado com suas avaliações de mestrado, Danilo assumiu todas as atividades. Ele implementou e testou a entidade e o *repository* *Product*, além de testar a entidade e o *repository* *Category*. O trabalho foi executado com muita competência, faltando apenas a inclusão de um construtor na classe *Product* que recebesse um *DTO*.
- **Subequipe 3:** Conforme acordado pelos membros da subequipe, Ian assumiu as atividades para essa sprint. Ele implementou e testou a *Entity* e o *Repository* para *ShoppingCart*, além de testar a *Entity* e o *Repository* para *Mail*. Embora tenha entregado com duas horas de atraso, isso não interferiu no andamento do trabalho.
- **Subequipe 4:** As entregas foram realizadas com sucesso: a implementação e os testes da *Entity* e do *Repository* para *ProductRating*, *ShoppingCartItem*, além da implementação e dos testes para *HandleException*.

É importante destacar que, devido aos padrões adotados, foi possível que equipes diferentes pudessem facilmente testar entidades que não haviam escrito. Por exemplo, Danilo, da Subequipe 2, testou a entidade *Category*, que foi criada pela Subequipe 3. Além disso, como acordado, o líder técnico corrigiu todos os erros de codificação. Outro aspecto relevante proveniente desta *sprint* foi o aumento da nossa cobertura de testes, passando de 40% para 65%, evidenciando uma considerável melhoria na qualidade do código.

3.2.3 Sprint 3

3.2.3.1 Sprint Planning Meeting

Durante a *daily* do dia 17 de maio, onde constou a análise da segunda *sprint*, também foi conduzida a terceira *sprint planning meeting*. A Tabela 8 apresenta a frequência dos colaboradores durante essa reunião.

Após a conclusão do desenvolvimento das camadas de *entities* e *repositories*, deu-se início à implementação da camada de *services*. Durante esta reunião, foram introduzidos os padrões de escrita para os *services*, os quais estão detalhados no tópico A.11. Assim, avançou-se para a distribuição das atividades da terceira sprint, conforme mostrado na

Tabela 8 – Frequência dos colaboradores na terceira *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino		
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

Tabela 9. É importante destacar que, nesta *sprint*, foram solicitados apenas os métodos para registro e busca de dados, além das classes de validação.

Tabela 9 – Distribuição das Tarefas entre as Subequipes na *Sprint 3*

1	2	3	4
Implementar e testar Service Category	Implementar e testar Service Product	Implementar e testar Service ProductAttribute	Implementar e testar Service ProductCart
Implementar e testar Service PossibleFacets	Implementar e testar Service ProductRating	Implementar e testar Service Mail	Implementar e testar Service ProductCartItem
			Buscar aumentar cobertura de testes

3.2.3.2 Resultados

Após o término da terceira *sprint*, durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

- **Subequipe 1:** Leonardo implementou corretamente o *CategoryService*, entretanto em relação ao *PossibleFacets* ficou no aguardo de Alexandre, que não realizou a entrega.
- **Subequipe 2:** Nessa *sprint* Murilo se apresentou para realização da atividade, entretanto teve dificuldade. A tarefa que lhe foi destinada foi o *ProductRatingService*, que foi realizada por Danilo que também estava responsável pelo *ProductService*. Danilo além dos métodos de *register* e buscas, também fez métodos de *update* e *disabled*. Entretanto, no *ProductRatingService*, que foi feito de última hora, ficou faltando alguns métodos de busca, explicitados no *card* do *Trello* dessa *sprint*.
- **Subequipe 3:** Renata concluiu com êxito a implementação do *MailService*, garantindo que todos os aspectos estivessem corretos. Por sua vez, Ian trabalhou no *ProductAttribute*, apresentando uma pequena falha nos nomes dos métodos das interfaces, que precisavam ser mais genéricos.

- **Subequipe 4:** O *ShoppingCartService* e *ShoppingCartItemService* foram implementados corretamente. Além disso, foi criado um *PurchaseService* para gerenciar ambos. Adicionalmente, alguns métodos de teste foram desenvolvidos para garantir a qualidade das implementações, contribuindo para o aumento da cobertura de teste nas classes desenvolvidas anteriormente.

É importante destacar que nessa *sprint* a cobertura de testes subiu de 65% para 71%, evidenciando uma melhoria na qualidade do código. Além disso, como acordado, o líder técnico corrigiu todos os erros de codificação, além de desenvolver as atividades que faltaram devido ao curto prazo.

3.2.4 Sprint 4

3.2.4.1 Sprint Planning Meeting

Durante a *daily* do dia 20 de maio, onde constou a análise da terceira *sprint*, também foi conduzida a quarta *sprint planning meeting*. A Tabela 10 apresenta a frequência dos colaboradores durante essa reunião.

Tabela 10 – Frequência dos colaboradores na quarta *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino		Teve que ir levar o animal de estimação do veterinário
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

Para esta *sprint*, foi destinado o prosseguimento na criação dos métodos dos *services*, introduzindo os métodos de *update* e *disable*, conforme descrito no tópico A.11. A divisão das tarefas foi realizada de acordo com a Tabela 11.

Tabela 11 – Distribuição das Tarefas entre as Subequipes na *Sprint 4*

1	2	3	4
Implementar e testar Service Category	Implementar e testar Service Product	Implementar e testar Service ProductAttribute	Implementar e testar Service ProductCart
Implementar e testar Service PossibleFacets	Implementar e testar Service ProductRating	Implementar e testar Service Mail	Implementar e testar Service ProductCartItem
			Buscar aumentar cobertura de testes

Como mencionado anteriormente, Danilo já havia implementado esses métodos, no entanto, não havia implementado a exclusão lógica. Este método apenas altera uma *flag* no banco de dados para marcar um dado como excluído ou não. Para isso, antes da *sprint*, o líder técnico introduziu novas *migrations* para adicionar essa nova coluna nas tabelas do banco de dados e incluiu o atributo nas classes, padronizando o processo.

Nessa reunião, foi decidido, em conjunto com os membros da sub-equipe 2, que Danilo e Murilo desenvolveriam as atividades em parceria para garantir a entrega das tarefas. Eles manteriam a comunicação via *discord*, o que permitiria a Danilo esclarecer as dúvidas de Murilo em tempo real.

3.2.4.2 Resultados

Após a conclusão da quarta *sprint*, constatou-se que os integrantes da equipe já haviam incorporado as regras de escrita de código, resultando em uma redução significativa dos erros.

É importante destacar que o líder técnico informou à subequipe 3, algumas horas após o início da *sprint*, que não seria necessário implementar os métodos de atualização e desativação para o *mailService*, já que o sistema não iria alterar nem deletar informações de *email*.

Durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

- **Subequipe 1:** Leonardo implementou corretamente os novos métodos para *CategoryService* e *PossibleFacetsService*.
- **Subequipe 2:** Murilo e Danilo implementaram corretamente os novos métodos para *ProductRatingService* e *ProductService*.
- **Subequipe 3:** Em relação ao *ProductAttributeService*, ficou como responsabilidade de Ian, que ainda não havia encerrado a implementação, alegando que estava desenvolvendo novos métodos para a classe.
- **Subequipe 4:** Foi desenvolvido métodos de *disabled* para *PurchaseService*, que gerencia o *ShoppingCartService* e o *shoppingCartItemService*. O método de *update* não foi implementado alegando que caso um usuário inicie uma compra errada, basta excluí-la e iniciar outra.

Vale ressaltar que nesta *sprint*, apesar do aumento na quantidade de linhas de código, a cobertura de testes permaneceu em 71%. Como não foram mais detectados erros significativos na estrutura do código, o líder técnico direcionou seus esforços para implementar as validações restantes necessárias para a execução dos métodos dos *services*.

3.2.5 Sprint 5

3.2.5.1 Sprint Planning Meeting

Durante a *daily* do dia 21 de maio, onde constou a análise da quarta *sprint*, também foi conduzida a quinta *sprint planning meeting*. A Tabela 12 apresenta a frequência dos colaboradores durante essa reunião.

Tabela 12 – Frequência dos colaboradores na quinta *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino	x	
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

Com os *services* parcialmente prontos, chegou o momento de iniciar o desenvolvimento dos *controllers*. Foram reiteradas aos integrantes as regras de escrita de código, de acordo com o que foi descrito no tópico A.12. Em seguida, as novas tarefas foram subdivididas, conforme apresentado na Tabela 13. Devido ao Ian estar implementando métodos adicionais em seu *service*, a demanda da *sprint* anterior para ele permaneceu inalterada para esta *sprint*.

Inicialmente, não foram solicitados métodos de teste para esses *controllers*, pois a prioridade era construí-los e os integrantes ainda não estavam familiarizados com os testes específicos para *controllers*. A decisão de reduzir a carga de trabalho de Leonardo por uma *sprint* foi tomada devido à sua situação de estar sem um companheiro de subequipe. Isso permitiu que ele descansasse e focasse em pendências de outras disciplinas da residência de software.

Tabela 13 – Distribuição das Tarefas entre as Subequipes na *Sprint 5*

1	2	3	4
Implementar Controller Category	Implementar Controller Product	Implementar Service ProductAttribute	Pesquisar como preencher o banco de dados
	Implementar Controller ProductRating	Implementar Controller PossibleFacets	Implementar o serviço de e-mail no controlador responsável por disparar e-mails.

3.2.5.2 Resultados

Durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

- **Subequipe 1:** Leonardo implementou corretamente o *ContollerCategory*.
- **Subequipe 2:** Murilo e Danilo implementarem corretamente o *ContollerProduct*, entretanto o *ContollerProductRating* ainda não estava concluído.
- **Subequipe 3:** Ian implementou o *ServiceProductAtributte* com excelência, trazendo uma grande quantidade de métodos diferentes de busca, além de implementar novos métodos de teste para classes que possuíam baixo nível de cobertura. Enquanto que Renata implementou o *ControllerPossibleFacets*.
- **Subequipe 4:** Foi implementado o serviço de *mail* no *controllador* responsável por disparar *e-mails*, registrando eles no banco. Entretanto, o preenchimento do banco de dados ainda não foi realizado.

Vale ressaltar que nesta *sprint*, apesar do aumento dos métodos de teste por parte de todos, e principalmente do Ian, a cobertura de testes caiu para 62%, o que é um indicativo que era necessário aumentar a cobertura de testes na próxima *sprint*.

3.2.6 *Sprint 6*

3.2.6.1 Sprint Planning Meeting

Durante a *daily* do dia 22 de maio, onde constou a análise da quinta *sprint*, também foi conduzida a sexta *sprint planning meeting*. A Tabela 14 apresenta a frequência dos colaboradores durante essa reunião.

Tabela 14 – Frequência dos colaboradores na sexta *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino	x	
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

Nesta *sprint*, optamos por dividir nossos esforços em três partes distintas. Primeiramente, decidimos que a SubEquipe 1 concentraria seus esforços em ampliar a cobertura de testes em todo o código. Em seguida, as subequipes 2 e 4 ficariam responsáveis pela implementação dos *controllers*. Quanto à SubEquipe 3, foi atribuída a Renata a tarefa de implementar o *ValidationMessages.properties*. Já Ian considerando a relevância dos métodos trazidos na *sprint* anterior, ficou responsável pela implementação de novos métodos de busca em todos os *services* desenvolvidos anteriormente. Em suma, essas informações estão apresentadas de forma mais concisa na Tabela 15.

Tabela 15 – Distribuição das Tarefas entre as Subequipes na *Sprint 6*

1	2	3	4
Aumentar Cobertura de Testes	Implementar Controller ProductRating	Implementar novos métodos de busca em todos os services	Implementar Controller Para o Shopping-Cart
	Implementar Controller ProductAtributte	Implementar ValidationMessages.properties	Implementar Controller Para o Shopping-CartItem.
		Implementar comentários Spring Doc nos controllers	

3.2.6.2 Resultados

Durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

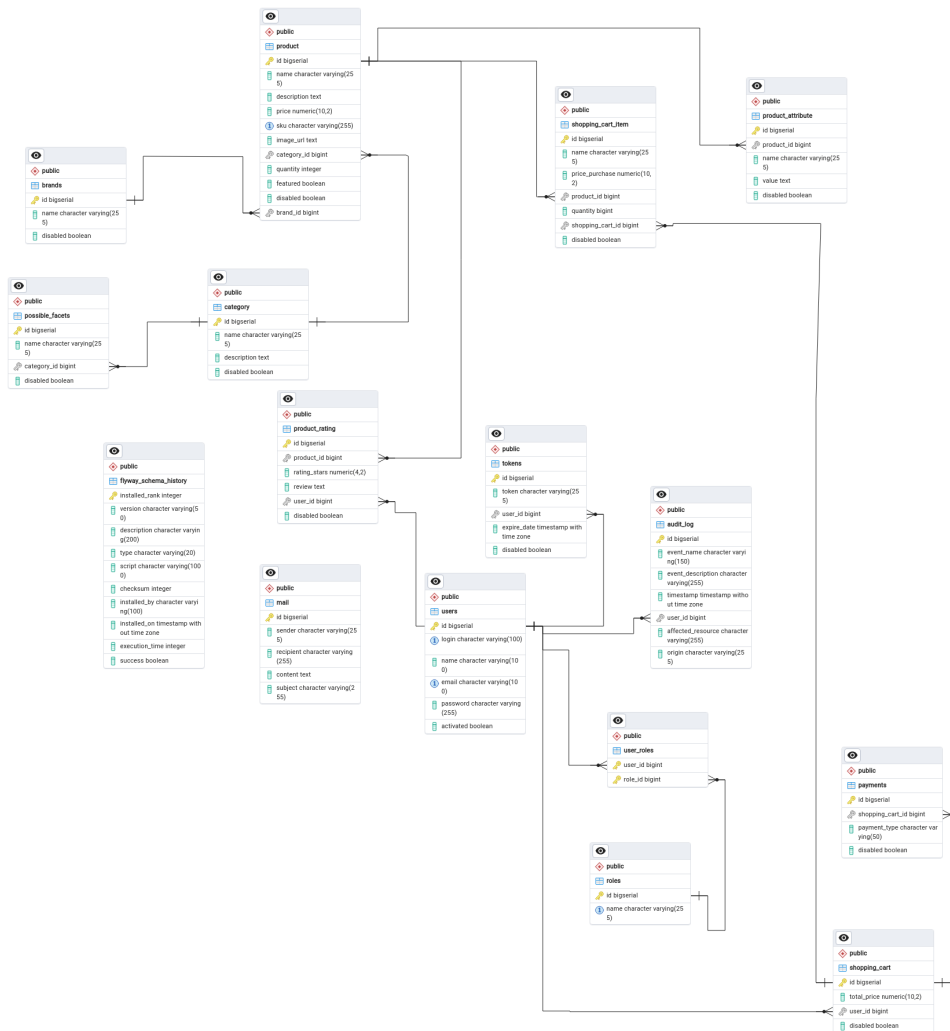
- **Subequipe 1:** Leonardo implementou 40 novos testes.
- **Subequipe 2:** Murilo e Danilo implementarem corretamente o *ContollerProductRating* e *ContollerProductAtributte*.
- **Subequipe 3:** Ian implementou novos métodos de busca em todos os *services*, enquanto Renata concluiu o arquivo *ValidationMessages.properties*. No entanto, a subequipe não entregou os comentários Spring Doc, planejando realizá-los na próxima sprint para todos os *controllers*.
- **Subequipe 4:** Foi desenvolvido o *controller Purchase* para gerenciar o serviço de compra, abrangendo também os *services shoppingCart* e *shoppingCartItem*. Além disso, foi realizado o desenvolvimento completo, desde as *entities* até os *controllers*, da classe *Brand* para as marcas dos produtos e *Payment* para registro de pagamentos.

Ademais, durante a reunião, foi realizada uma demonstração abrangente do funcionamento dos métodos implementados. Utilizamos o *Postman* para enviar requisições e exemplificar como cada método opera em diferentes cenários. Essa abordagem permitiu que todos compreendessem não apenas a implementação técnica, mas também a interação prática com os serviços desenvolvidos. Através dessa demonstração, garantimos que todos os membros da equipe estivessem alinhados e totalmente capacitados para utilizar e entender os novos recursos adicionados ao projeto.

Por conseguinte, a Figura 4 apresenta a versão do banco de dados após a conclusão dessa *sprint*. Esta ilustração foi gerada utilizando o PgAdmin, destacando as evoluções e refinamentos alcançados ao longo do processo.

Em relação aos testes, apesar de termos aumentado a quantidade de testes realizados, a cobertura de código continuou caindo, fomos de 65% para 58%. Isso se deve ao crescimento no número de classes criadas, que expandiu significativamente o código.

Figura 4 – Modelo do banco de dados após a sexta sprint



3.2.7 Sprint 7

3.2.7.1 Sprint Planning Meeting

Durante a *daily* do dia 23 de maio, onde constou a análise da sexta *sprint*, também foi conduzida a sétima *sprint planning meeting*. A Tabela 16 apresenta a frequência dos colaboradores durante essa reunião.

Para esta *sprint*, que seria a última no planejamento devido ao fato de ser o último dia, o foco foi direcionado ao aumento da cobertura de testes, visto que o sistema já apresentava as funcionalidades necessárias para um *Minimum Viable Product* (Produto Mínimo Viável). Esse foco adicional nos testes visava identificar e corrigir possíveis *bugs* no

Tabela 16 – Frequência dos colaboradores na sétima *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino	x	
Renata C. de Oliveira Moreno	x	
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

código, garantindo assim a entrega de um produto final de alta qualidade. A distribuição das atividades para essa *sprint* está na Tabela 17.

Tabela 17 – Distribuição das Tarefas entre as Subequipes na *Sprint 7*

1	2	3	4
Aumentar cobertura de testes nos <i>validations</i>	Aumentar cobertura de testes nos <i>controllers</i>	Aumentar cobertura de testes nos <i>services</i>	Aplicar seeders
Aumentar cobertura de testes nas <i>entitys</i>			Aplicar controle de permissão aos <i>controllers</i> .

3.2.7.2 Resultados

Durante a reunião de avaliação, foram levantados alguns pontos referentes ao desempenho das subequipes:

- **Subequipe 1:** Leonardo implementou corretamente o testes para *validations* e para *entitys*.
- **Subequipe 2:** Murilo e Danilo implementarem corretamente o novos testes para *controllers*.
- **Subequipe 3:** Ian e Renata implementaram novos testes para *services*.
- **Subequipe 4:** Foi aplicado *seeders* parar todos os produtos da Positivo e *DELL*, conforme solicitado pelos *stakeholders*, além da aplicação correta do controle de permissão aos endpoints.

Durante essa *sprint*, conseguimos elevar a cobertura de 58% para 66%, evidenciando um progresso significativo na qualidade do código. Além disso, é importante destacar que, durante o decorrer da sétima *sprint*, foi comunicado aos residentes sobre um adiamento no prazo de entrega do projeto, estendendo-o até o dia 28. Esse ajuste não impactou de forma significativa o planejamento, uma vez que já havíamos alcançado o (MVP).

3.2.8 Sprint 8

3.2.8.1 Sprint Planning Meeting

Em virtude do adiamento do prazo, discutimos com a equipe, durante a reunião de análise da sétima *sprint*, quais seriam os próximos passos, considerando que o sistema já estava funcional. Decidimos concentrar nossos esforços na melhoria dos testes, delegando essa tarefa às subequipes 1, 2 e 3, enquanto a subequipe 4 assumiu a responsabilidade de utilizar esse tempo para elaborar o presente relatório. A Tabela 18 apresenta a frequência dos colaboradores durante essa reunião. A sprint em questão teve seu planejamento inicial para encerrar no dia 27 de maio, levando em consideração que não temos reuniões aos finais de semana.

Tabela 18 – Frequência dos colaboradores na oitava *sprint planning meet*

Colaborador	Presença	Justificativa
João Vitor Nascimento Ramos	x	
Danilo da Conceição Santos	x	
Leonardo Ribeiro Barbosa Santos	x	
Ian Rodrigues Alexandrino		Trabalhou até tarde da noite realizando testes.
Renata C. de Oliveira Moreno		Tinha um compromisso
Murilo Carlos Novais	x	
Alexandre de Souza Amaral		

A distribuição das atividades para essa *sprint* está na Tabela 19.

Tabela 19 – Distribuição das Tarefas entre as Subequipes na *Sprint 8*

1	2	3	4
Aumentar cobertura de testes nos validations	Aumentar cobertura de testes nos <i>controllers</i>	Aumentar cobertura de testes nos <i>services</i> remanescentes	Finalizar relatório
Aumentar cobertura de testes no ServiceShoppingCartItem	Aumentar cobertura de testes no Service ProductRating		

3.2.9 Resultados

Em relação a esta *sprint*, foram levantados alguns pontos sobre o desempenho das subequipes:

- **Subequipe 1:** Leonardo implementou corretamente 40 novos métodos de testes.
- **Subequipe 2:** Murilo e Danilo implementaram corretamente 44 novos métodos de testes.

- **Subequipe 3:** Ian e Renata implementaram corretamente 35 novos métodos de testes.
- **Subequipe 4:** finalizou o relatório.

Durante essa *sprint*, conseguimos elevar a cobertura de 66% para 84%, evidenciando um progresso significativo na qualidade do código.

4 Funcionalidades Desenvolvidas

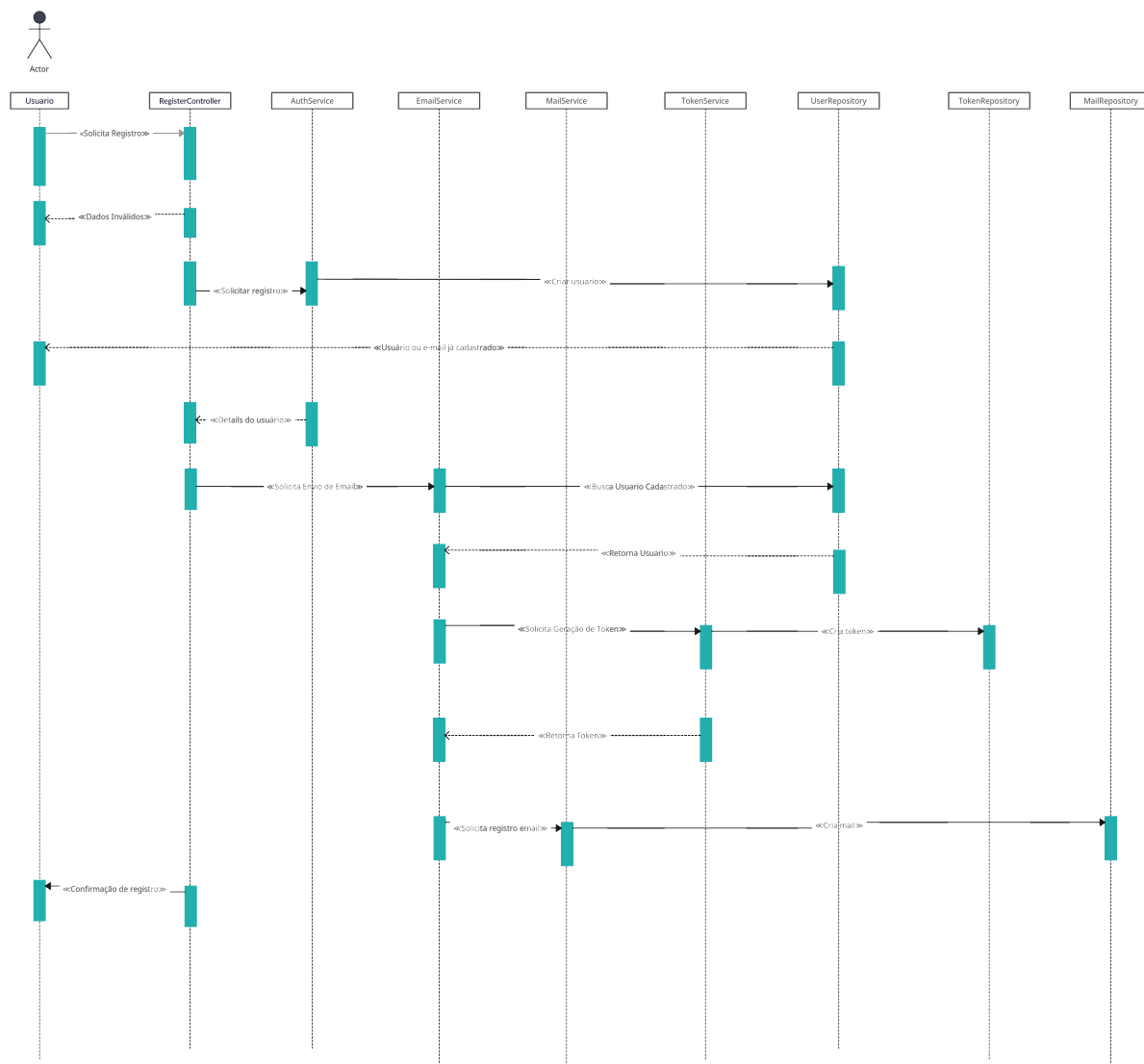
Neste capítulo, serão apresentados as funcionalidades do sistema. Vale ressaltar que o *usuario* retratado nas imagens dessa seção é o *front-end* ou a aplicação que esta consumindo a API..

4.1 Segurança

4.1.1 Registro de usuário

O diagrama da Figura 5 ilustra o caso de uso completo para o registro de usuário, cobrindo desde o envio da solicitação até o envio do *e-mail* de ativação.

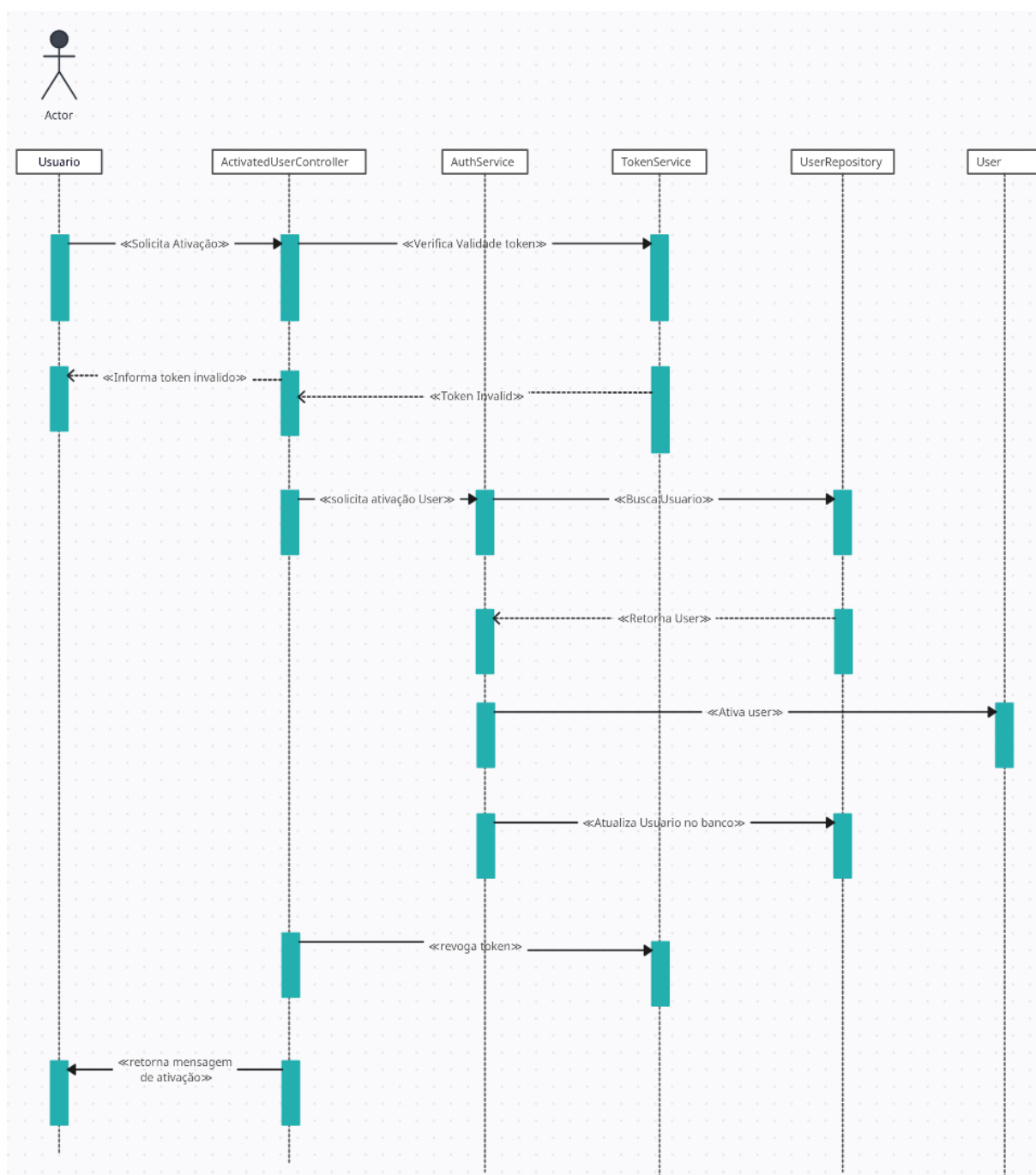
Figura 5 – Diagrama de sequência para registro de usuário



4.1.2 Ativação do usuário

Este diagrama de sequência ilustra o processo de ativação do usuário após o registro. O processo começa quando o usuário clica no link de ativação enviado por *e-mail* e prossegue até a confirmação da ativação, conforme representado na Figura 6.

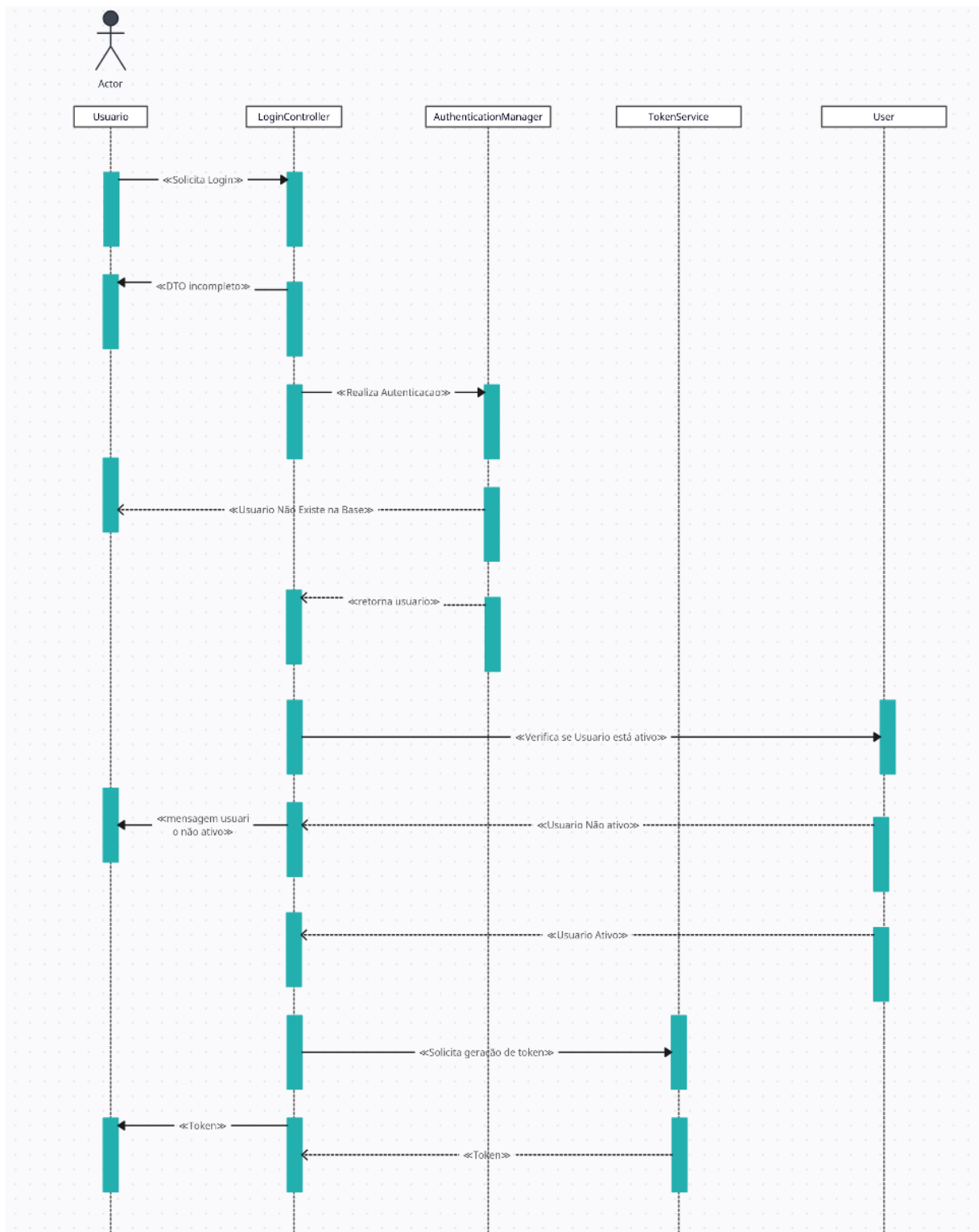
Figura 6 – Diagrama de sequência para registro de usuário



4.1.3 Login

O diagrama da Figura 7 ilustra o processo de login do usuário. Este processo começa quando o usuário insere suas credenciais no sistema e prossegue com a verificação dessas credenciais e a concessão de acesso, caso sejam válidas.

Figura 7 – Diagrama de sequência para login de usuário

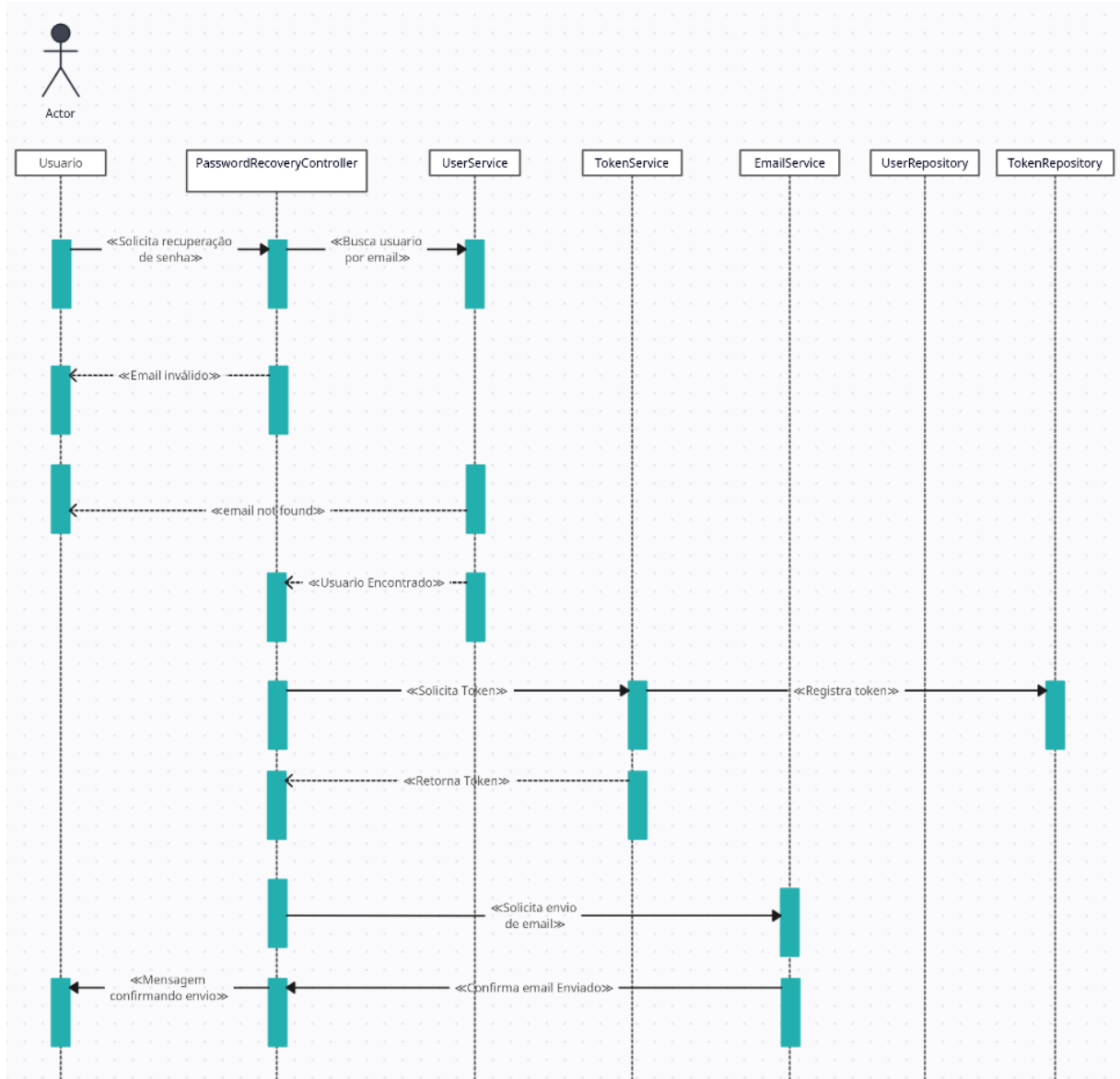


4.1.4 Recuperação de senha

4.1.4.1 Envio de email para recuperação

O diagrama da Figura 8 ilustra o processo de envio de e-mail para recuperação de senha. Este processo começa quando o usuário solicita a recuperação de senha e prossegue com o envio de um *e-mail* contendo um link para redefinir a senha.

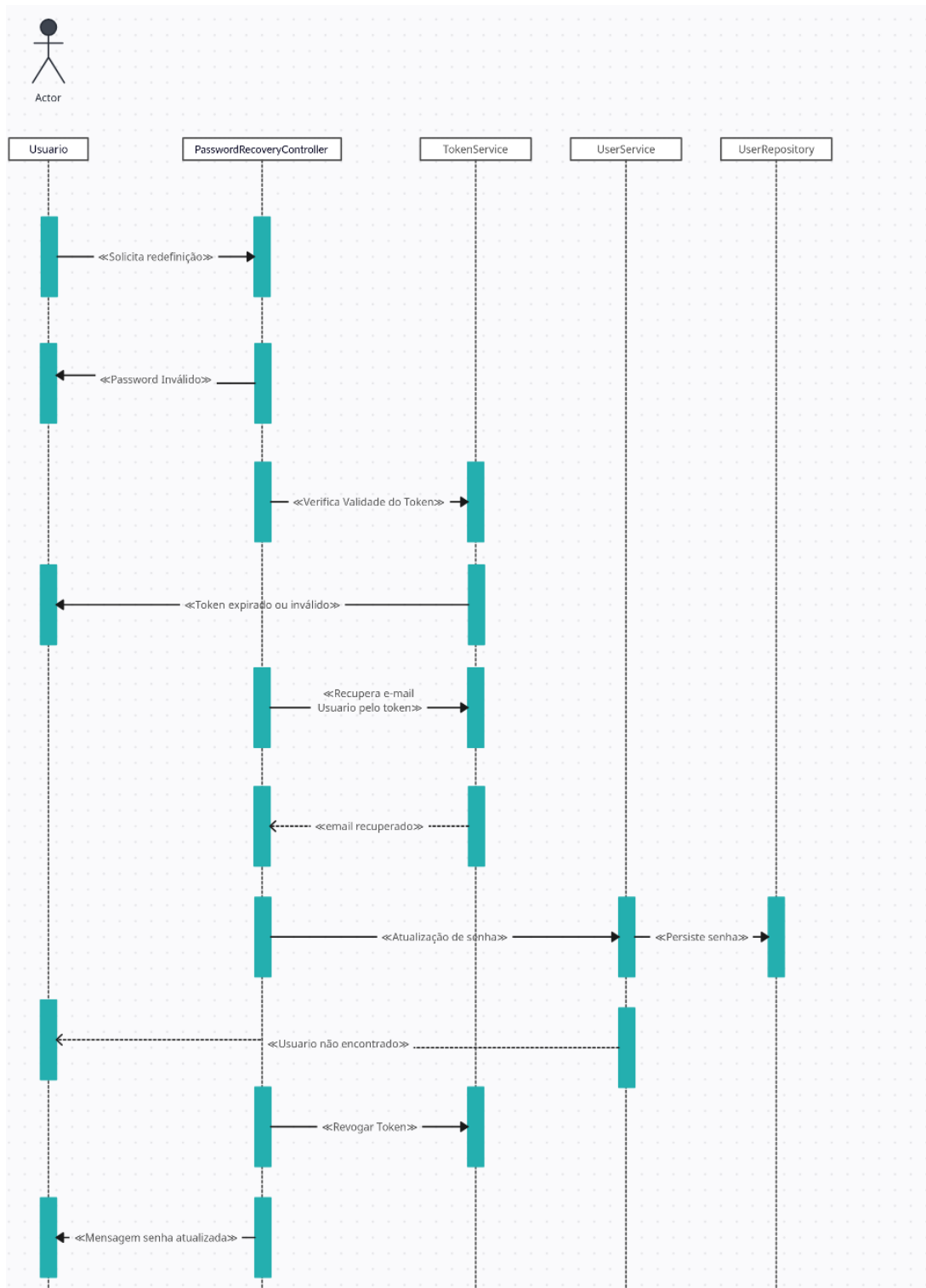
Figura 8 – Diagrama de sequência para envio de email para recuperação de senha



4.1.4.2 Redefinição de senha

O diagrama da Figura 9 ilustra o processo de envio de e-mail para recuperação de senha. Este processo começa quando o usuário solicita a recuperação de senha e prossegue com o envio de um *e-mail* contendo um link para redefinir a senha.

Figura 9 – Diagrama de sequência para redefinição de senha



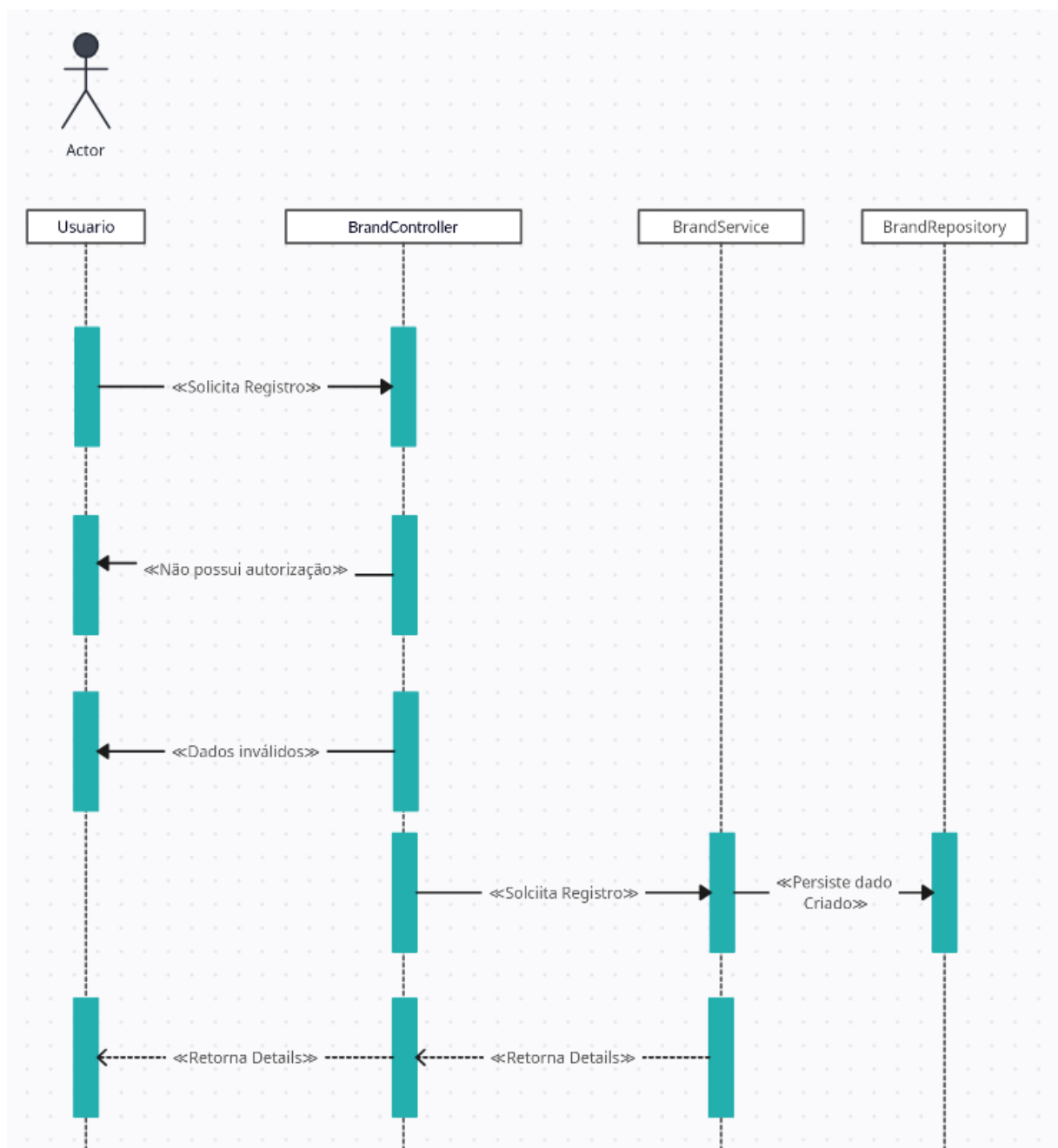
4.2 Brand

Esta seção descreve as operações essenciais para gerenciar marcas dentro do sistema.

4.2.1 Registrar nova Brand

O registro de uma nova brand permite adicionar uma marca ao sistema. O diagrama de sequência na Figura 10 mostra o fluxo para registrar uma nova marca, incluindo a validação e armazenamento dos dados no banco de dados.

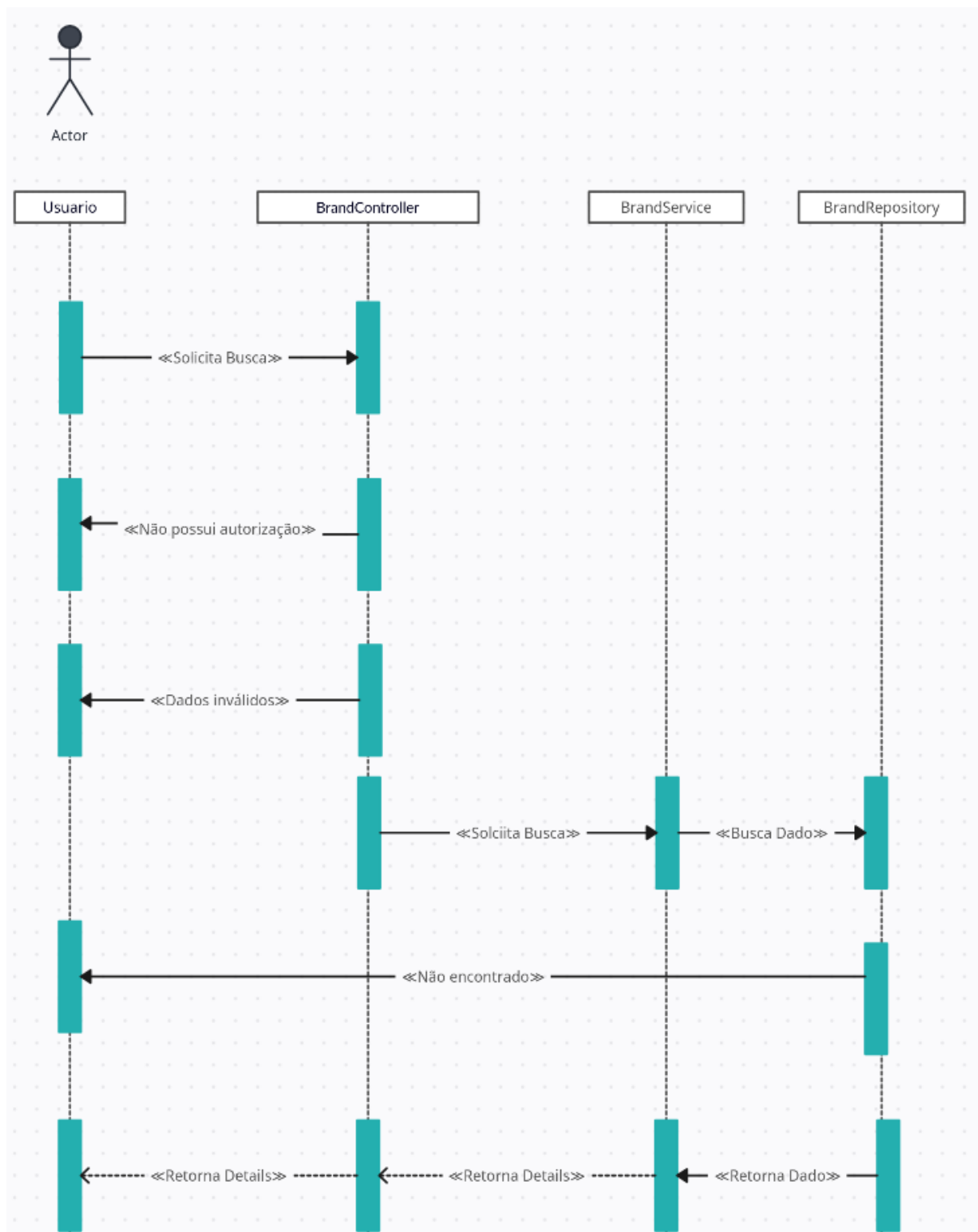
Figura 10 – Diagrama de sequência para registrar nova brand



4.2.2 Buscar Brand por ID

A busca por ID permite obter detalhes de uma marca específica através de seu identificador único. O sistema consulta a base de dados e retorna as informações da marca correspondente. O diagrama de sequência na Figura 11 mostra o fluxo para buscar uma marca por ID.

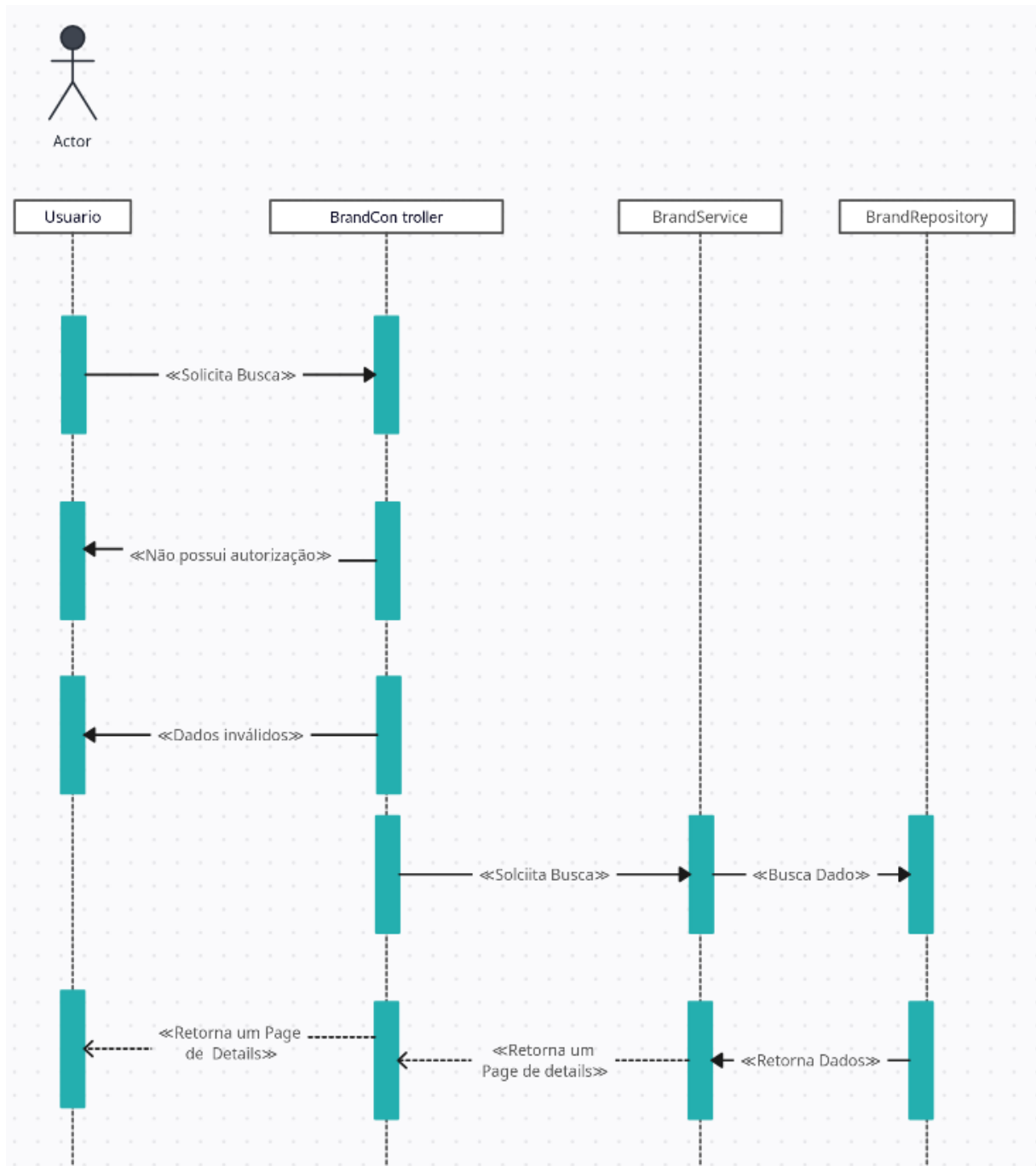
Figura 11 – Diagrama de sequência para buscar uma brand por ID



4.2.3 Buscar todas as Brands

Esta funcionalidade recupera a lista completa de marcas cadastradas no sistema. O usuário pode ver todas as marcas. O diagrama de sequência na Figura 12 mostra o fluxo para buscar uma marca por ID.

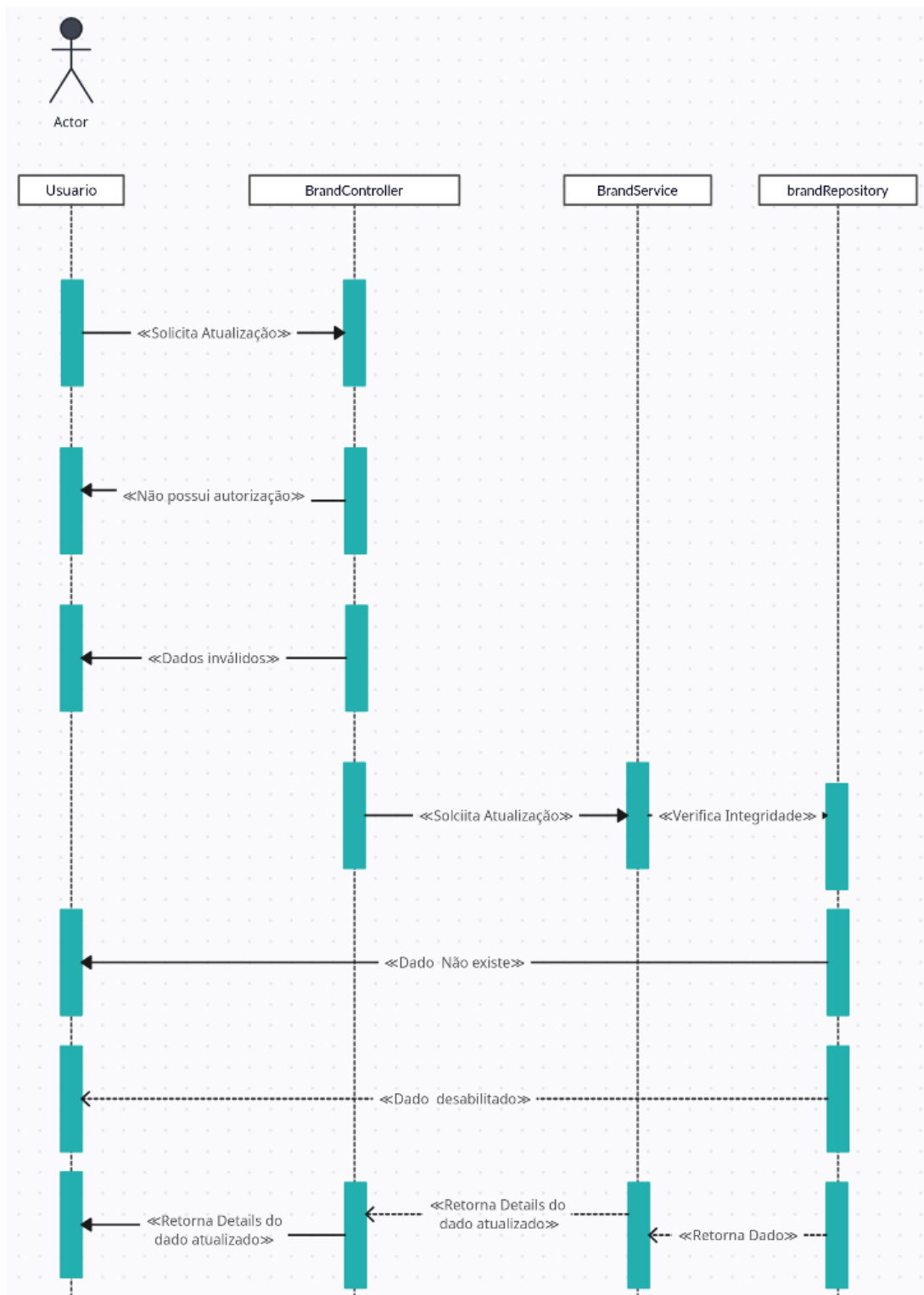
Figura 12 – Diagrama de sequência para buscar todas as brands



4.2.4 Atualizar Brand

Atualizar uma *brand* permite modificar os dados de uma marca existente. O sistema valida e atualiza as informações fornecidas, mantendo os registros atualizados. O diagrama de sequência na Figura 13 mostra o fluxo para atualizar uma brand.

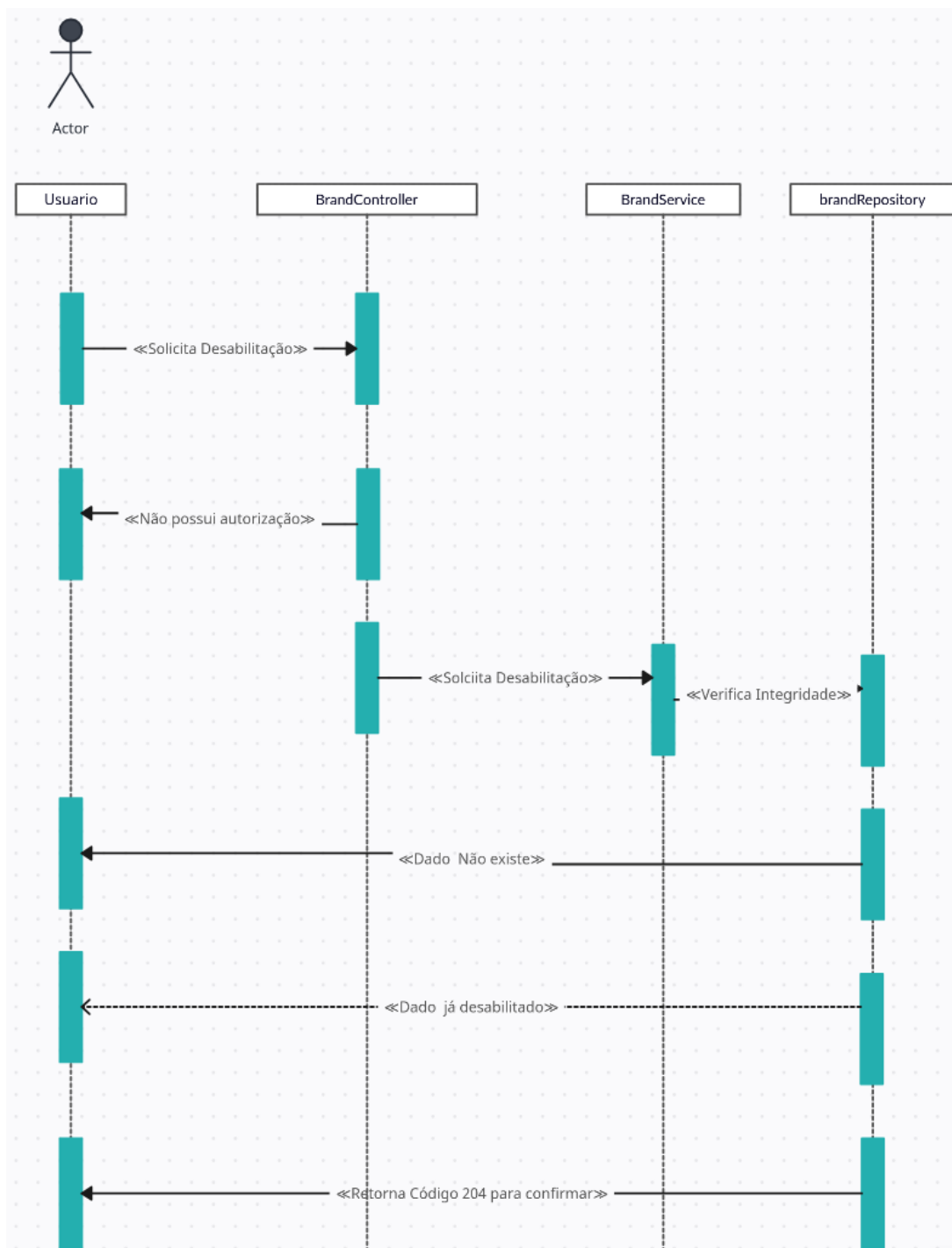
Figura 13 – Diagrama de sequência para atualizar brand



4.2.5 Desabilitar Brand

Desabilitar uma *brand* altera seu status para desativado, mantendo seus dados no sistema. Isso é útil para marcas que não estão em uso ativo, mas que devem ser preservadas para referência futura. O diagrama de sequência na Figura 14 mostra o fluxo para desabilitar uma brand.

Figura 14 – Diagrama de sequência para desabilitar brand



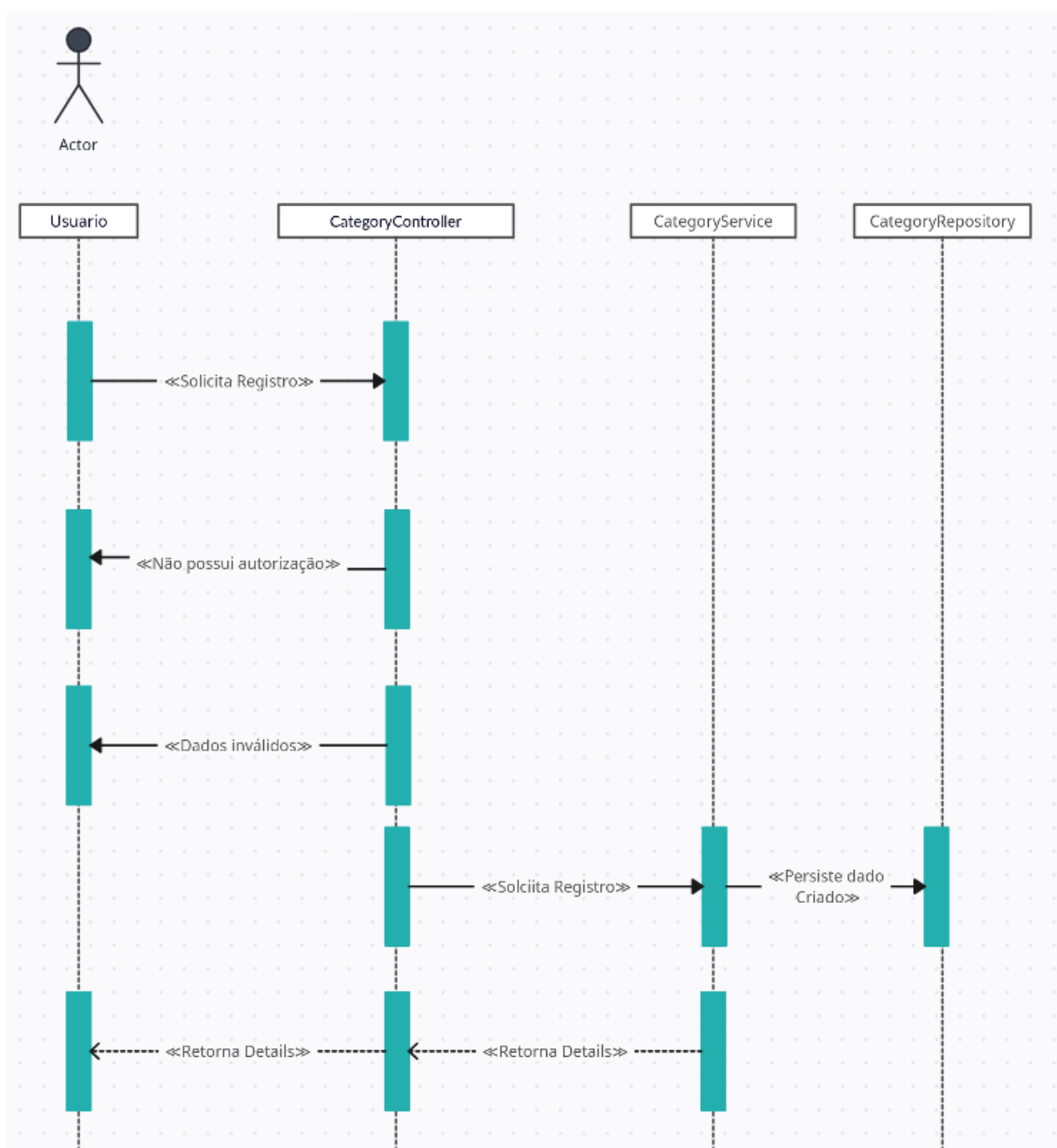
4.3 Category

Esta seção descreve as operações essenciais para gerenciar categorias dentro do sistema.

4.3.1 Registrar nova Category

Registrar uma nova categoria envolve a validação e armazenamento das informações fornecidas. O diagrama de sequência na Figura 15 mostra o fluxo para registrar uma nova marca, incluindo a validação e armazenamento dos dados no banco de dados.

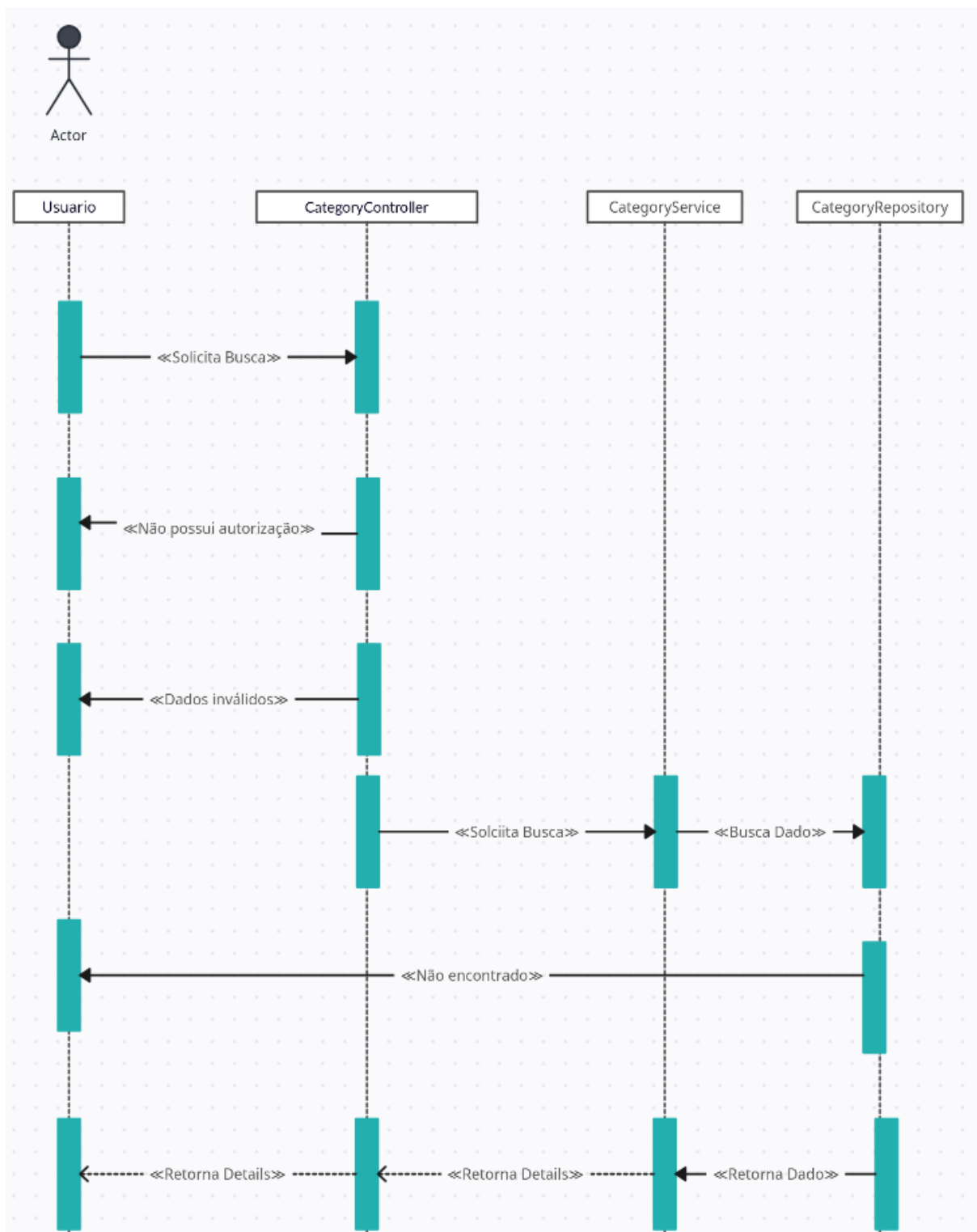
Figura 15 – Diagrama de sequência para registrar nova categoria



4.3.2 Buscar Category por ID

Buscar uma categoria por ID permite recuperar os detalhes de uma categoria específica usando seu identificador único. Isso é útil para obter informações detalhadas sobre uma categoria específica. O diagrama de sequência na Figura 16 mostra o fluxo para buscar uma marca por ID.

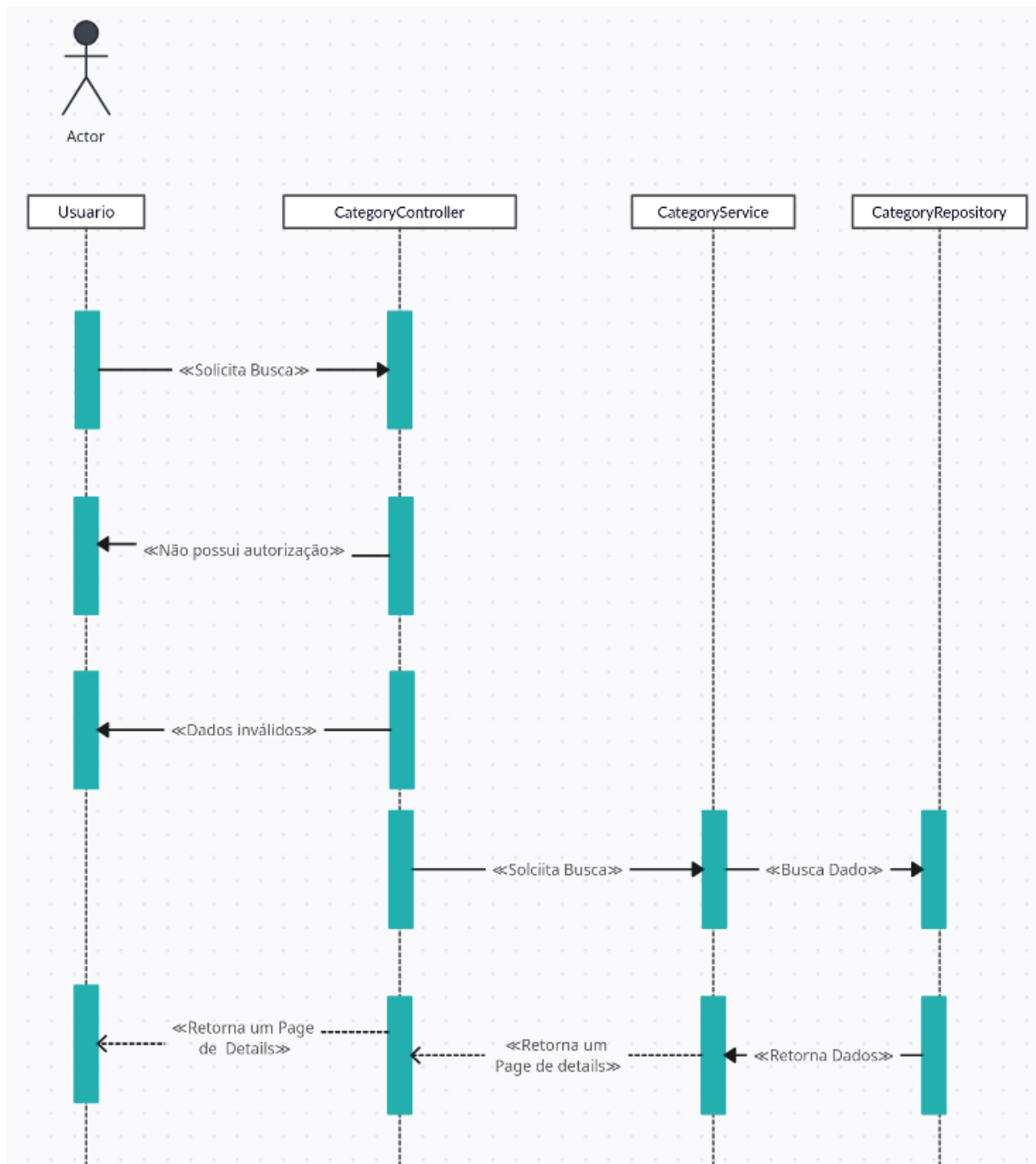
Figura 16 – Diagrama de sequência para buscar uma brand por ID



4.3.3 Buscar todas as categories

Buscar todas as categorias disponíveis no sistema oferece uma visão geral das opções disponíveis. Isso permite aos usuários explorar todas as categorias existentes sem a necessidade de saber seus nomes específicos. O diagrama de sequência na Figura 17 mostra o fluxo para buscar uma marca por ID.

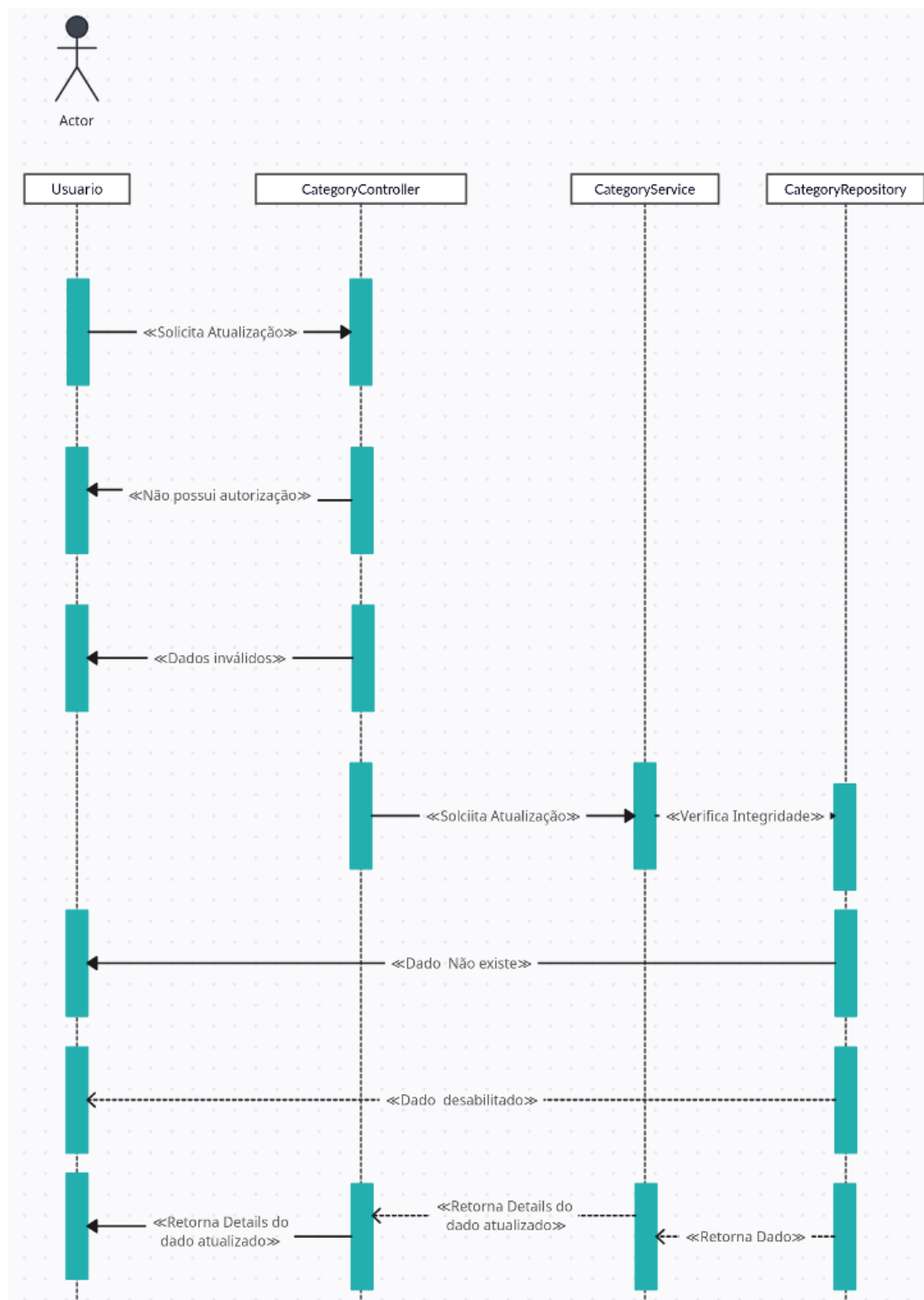
Figura 17 – Diagrama de sequência para buscar todas as categorias



4.3.4 Atualizar Category

Atualizar uma categoria permite modificar suas informações existentes. Isso é útil para corrigir erros ou atualizar detalhes sobre uma categoria, garantindo que os dados estejam sempre precisos e atualizados. O diagrama de sequência na Figura 18 mostra o fluxo para atualizar uma category.

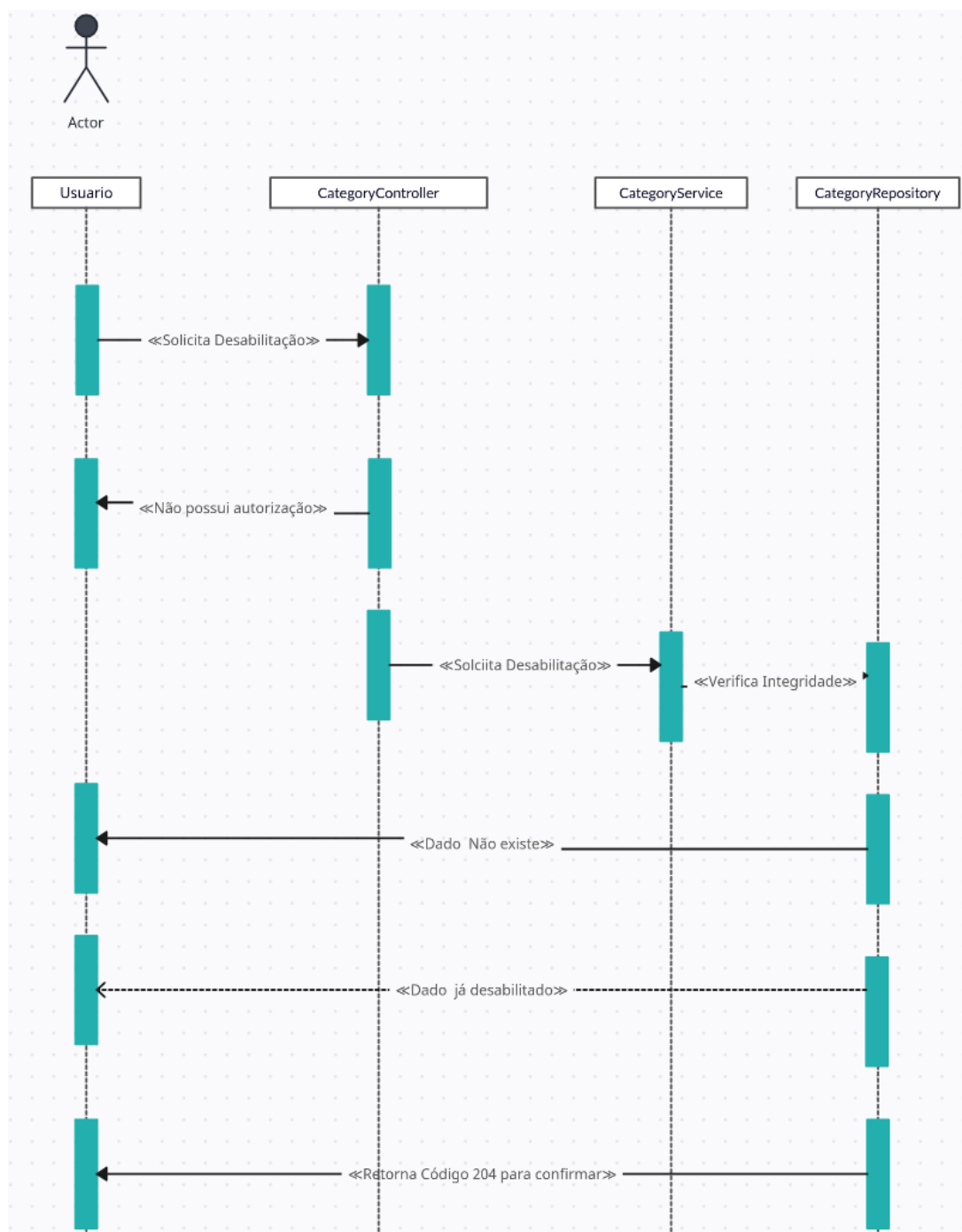
Figura 18 – Diagrama de sequência para atualizar category



4.3.5 Desabilitar Category

Desabilitar uma categoria altera seu status para desativado, removendo-a das opções ativas. Isso é útil quando uma categoria não está mais em uso ou precisa ser temporariamente removida do sistema. O diagrama de sequência na Figura 19 mostra o fluxo para desabilitar uma category.

Figura 19 – Diagrama de sequência para desabilitar category



4.3.6 Buscar categorias por nome

A busca por categorias por nome é uma funcionalidade essencial para simplificar a navegação e pesquisa dentro do sistema, permitindo aos usuários localizar categorias específicas ou conjuntos semelhantes. Embora não incluído aqui, o diagrama de sequência para essa operação é omitido devido à sua simplicidade e similaridade com outras operações de busca, mantendo o foco nas operações mais distintas e complexas do sistema de gerenciamento de categorias.

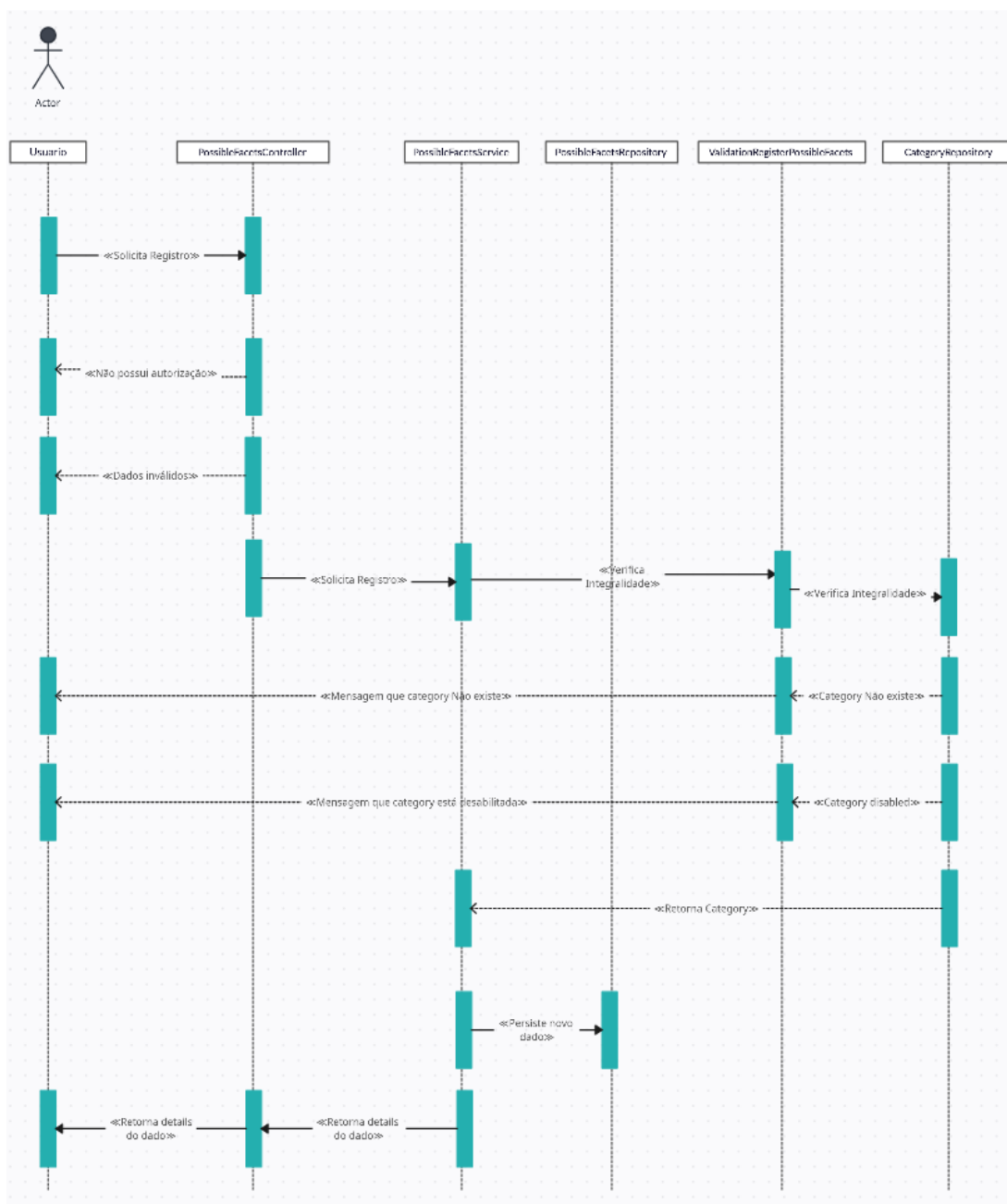
4.4 PossibleFacets

Esta seção descreve as operações essenciais para gerenciar os atributos de categorias dentro do sistema.

4.4.1 Registrar Atributo de categoria

O diagrama da Figura 20 ilustra o processo de registro de um novo atributo de categoria. Este processo inicia quando um usuário solicita o registro de um novo atributo e prossegue com a validação e armazenamento dessas informações no sistema.

Figura 20 – Diagrama de sequência para registrar novo atributo de categoria



4.4.2 Buscar Todas as PossibleFacets

O comportamento desta operação é semelhante ao de outras consultas específicas, como buscar todas as marcas e categorias (4.2.3, ??), que já foram detalhadas anteriormente. Sua inclusão aqui serve apenas como referência à funcionalidade, mantendo o foco nas operações essenciais para o gerenciamento de atributos de categorias.

4.4.3 Buscar PossibleFacets por ID

Esta operação segue o mesmo padrão de outras consultas por ID, como buscar uma marca por ID (4.2.2) e buscar uma categoria por ID (4.3.2). Não é necessário detalhar novamente seu comportamento, visto que ele foi explicado em seções anteriores. Sua presença aqui é apenas para fins de referência, mantendo o conteúdo conciso e direcionado.

4.4.4 Atualizar PossibleFacets

Esta operação segue o mesmo padrão de atualização de outras entidades, como marcas (4.2.4) e categorias (4.3.3). No entanto, se diferencia ao incluir um conjunto adicional de validações, similar ao processo de registro de atributos de categoria (4.4.1). As validações específicas para esta operação incluem:

- Verificar se o PossibleFacets existe.
- Verificar se o PossibleFacets está desabilitado.
- Verificar se a categoria associada ao PossibleFacets existe.
- Verificar se a categoria associada ao PossibleFacets está desabilitada.

4.4.5 Desabilitar PossibleFacets

Esta operação segue o mesmo padrão de desabilitação de outras entidades, como marcas (4.2.5) e categorias (4.3.5). As validações para esta operação incluem:

- Verificar se o PossibleFacets existe.
- Verificar se o PossibleFacets já está desabilitado.

4.4.6 Buscar possibleFacets por categoria

Esta operação permite buscar todos os PossibleFacets associados a uma categoria específica. O comportamento desta operação é similar ao de outras consultas específicas e, portanto, não será mostrado o diagrama de sequência aqui.

4.5 Product

As operações para gerenciar produtos seguem padrões semelhantes aos das operações de outras entidades, como marcas e categorias, já apresentadas anteriormente. Portanto, os diagramas de sequência para essas operações não serão detalhados novamente. Esta seção se concentrará nas descrições das operações e nas validações específicas necessárias para cada uma.

4.5.1 Registrar Produto

Para registrar um novo produto, são necessárias as seguintes validações:

- Verificar se a marca associada ao produto existe e está habilitada.
- Verificar se a categoria associada ao produto existe e está habilitada.

4.5.2 Listar Produtos

A operação de listar produtos permite obter uma lista paginada de produtos registrados no sistema. Esta funcionalidade é essencial para a visualização e gerenciamento dos produtos disponíveis.

4.5.3 Buscar Produto por ID

A operação de buscar um produto por ID permite obter os detalhes de um produto específico, utilizando seu identificador único. Esta funcionalidade é crucial para acessar informações detalhadas sobre um produto específico.

4.5.4 Buscar Produtos por Categoria

Esta operação permite buscar todos os produtos associados a uma categoria específica. O comportamento desta operação é similar ao de outras consultas específicas por categoria ou outra entidade e, portanto, não será detalhado novamente.

4.5.5 Atualizar Produto

Para atualizar um produto, são necessárias as seguintes validações:

- Verificar se o produto existe.
- Verificar se o produto está desabilitado.
- Verificar se a marca associada ao produto existe e está habilitada.
- Verificar se a categoria associada ao produto existe e está habilitada.

4.5.6 Desabilitar Produto

Para desabilitar um produto, são necessárias as seguintes validações:

- Verificar se o produto existe.
- Verificar se o produto está habilitado.

4.6 ProductAttributes

As operações para gerenciar atributos de produtos seguem padrões semelhantes aos das operações de outras entidades, como produtos, marcas e categorias, já apresentadas anteriormente. Portanto, os diagramas de sequência para essas operações não serão detalhados novamente. Esta seção se concentrará nas descrições das operações e nas validações específicas necessárias para cada uma.

4.6.1 Registrar ProductAttribute

Para registrar um novo atributo de produto, são necessárias as seguintes validações:

- Verificar se o produto associado ao atributo existe.
- Verificar se o produto associado ao atributo está habilitado.

4.6.2 Listar ProductAttributes

A operação de listar atributos de produtos permite obter uma lista paginada de atributos registrados no sistema. Esta funcionalidade é essencial para a visualização e gerenciamento dos atributos disponíveis.

4.6.3 Listar ProductAttributes Desabilitados

A operação de listar atributos de produtos desabilitados permite obter uma lista paginada de atributos que foram desativados. Esta funcionalidade é útil para a administração e possíveis reativações de atributos.

4.6.4 Buscar ProductAttributes por ID

A operação de buscar um atributo de produto por ID permite obter os detalhes de um atributo específico, utilizando seu identificador único. Esta funcionalidade é crucial para acessar informações detalhadas sobre um atributo específico.

4.6.5 Buscar ProductAttributes por Produto

Esta operação permite buscar todos os atributos associados a um produto específico. O comportamento desta operação é similar ao de outras consultas específicas por entidade e, portanto, não será detalhado novamente.

4.6.6 Buscar ProductAttributes Desabilitados por Produto

Esta operação permite buscar todos os atributos desabilitados associados a um produto específico. O comportamento desta operação é similar ao de outras consultas específicas por entidade e, portanto, não será detalhado novamente.

4.6.7 Atualizar ProductAttribute

Para atualizar um atributo de produto, são necessárias as seguintes validações:

- Verificar se o atributo de produto existe.
- Verificar se o atributo de produto está desabilitado.
- Verificar se o produto associado ao atributo existe.
- Verificar se o produto associado ao atributo está habilitado.

4.6.8 Desabilitar ProductAttribute

Para desabilitar um atributo de produto, são necessárias as seguintes validações:

- Verificar se o atributo de produto existe.
- Verificar se o atributo de produto está habilitado.

4.7 ProductRating

As operações para gerenciar avaliações de produtos seguem padrões semelhantes aos das operações de outras entidades, como produtos, marcas e categorias, já apresentadas anteriormente. Portanto, os diagramas de sequência para essas operações não serão detalhados novamente. Esta seção se concentrará nas descrições das operações e nas validações específicas necessárias para cada uma.

4.7.1 Registrar Avaliação de Produto

Para registrar uma nova avaliação de produto, são necessárias as seguintes validações:

- Verificar se o produto associado à avaliação existe e está habilitado.
- Verificar se o usuário que está realizando a avaliação existe e está habilitado.

4.7.2 Listar Avaliações de Produto

A operação de listar avaliações de produtos permite obter uma lista paginada de avaliações registradas no sistema. Esta funcionalidade é essencial para a visualização e gerenciamento das avaliações disponíveis.

4.7.3 Buscar Avaliação de Produto por ID

A operação de buscar uma avaliação de produto por ID permite obter os detalhes de uma avaliação específica, utilizando seu identificador único. Esta funcionalidade é crucial para acessar informações detalhadas sobre uma avaliação específica.

4.7.4 Buscar Avaliações por Produto

Esta operação permite buscar todas as avaliações associadas a um produto específico. O comportamento desta operação é similar ao de outras consultas específicas por entidade e, portanto, não será detalhado novamente.

4.7.5 Buscar Avaliações por Usuário

Esta operação permite buscar todas as avaliações realizadas por um usuário específico. O comportamento desta operação é similar ao de outras consultas específicas por entidade e, portanto, não será detalhado novamente.

4.7.6 Atualizar Avaliação de Produto

Para atualizar uma avaliação de produto, são necessárias as seguintes validações:

- Verificar se a avaliação de produto existe.
- Verificar se a avaliação de produto está desabilitada.
- Verificar se o produto associado à avaliação existe e está habilitado.
- Verificar se o usuário que está realizando a avaliação existe e está habilitado.

4.7.7 Desabilitar Avaliação de Produto

Para desabilitar uma avaliação de produto, são necessárias as seguintes validações:

- Verificar se a avaliação de produto existe.
- Verificar se a avaliação de produto está habilitada.

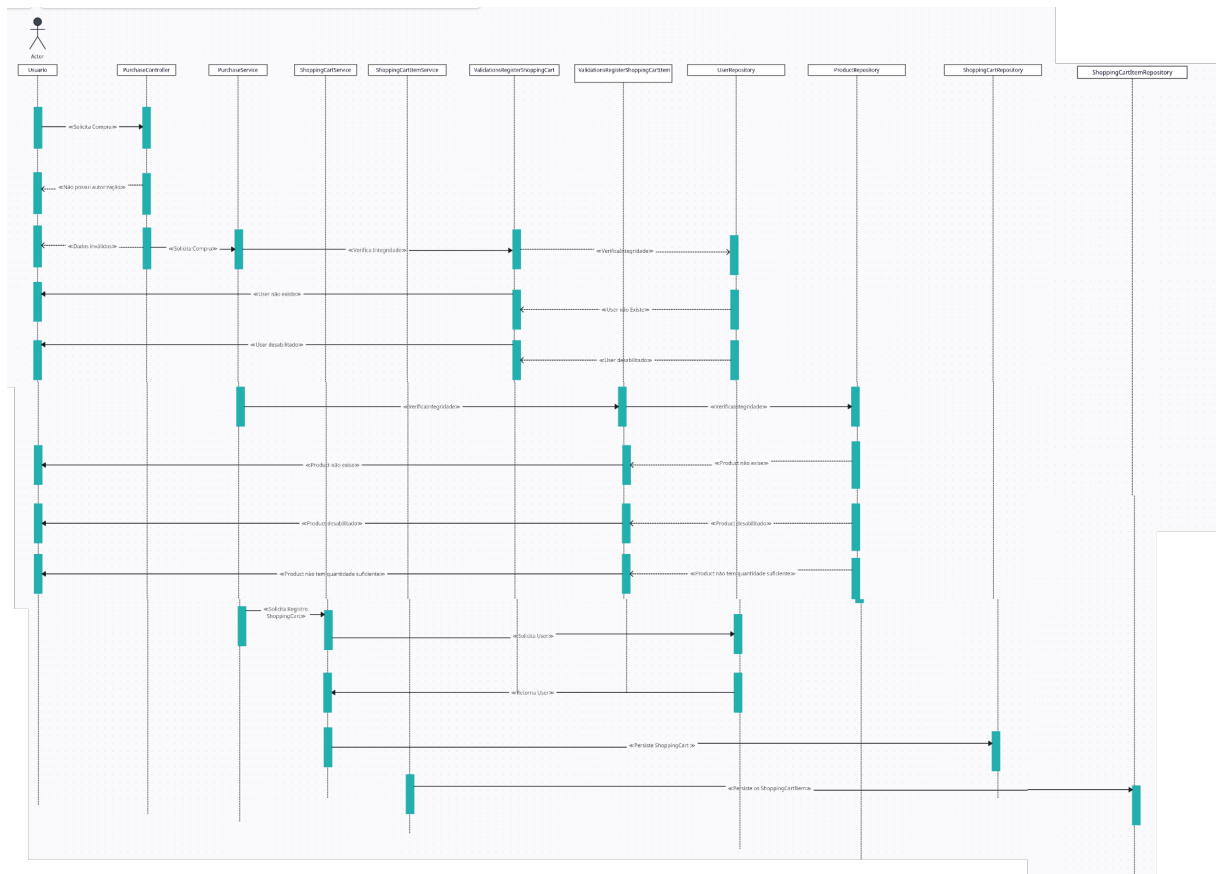
4.8 Purchase

4.8.1 Realizar Compra

O diagrama de sequência para registrar uma nova compra está representado na Figura 21. Observa-se que as distorções na imagem ocorreram devido ao seu tamanho, comprometendo parcialmente a legibilidade. No entanto, a estrutura geral do processo ainda é compreensível.

Foi omitido o retorno do detalhes da compra para manter a simplicidade e a clareza do diagrama. Incluir detalhes adicionais de retorno poderia sobrecarregar visualmente o diagrama, tornando-o menos eficaz como uma ferramenta de comunicação visual.

Figura 21 – Diagrama de sequência para registrar nova compra



4.8.2 Cancelar Compra

A operação de cancelamento de compra permite aos usuários cancelarem uma compra previamente realizada. Esta funcionalidade é essencial para lidar com situações onde os clientes desejam desistir de uma compra por algum motivo específico.

As validações necessárias para o cancelamento de compra são as seguintes:

- Verificar se o carrinho de compras associado à compra existe.
- Verificar se o carrinho de compras associado à compra está desabilitado.
- Verificar se existe um pagamento ativo associado à compra.

4.9 Payments

As operações para gerenciar pagamentos seguem padrões semelhantes aos das operações de outras entidades já apresentadas anteriormente. Portanto, os diagramas de sequência para essas operações não serão detalhados novamente. Esta seção se concentrará nas descrições das operações e nas validações específicas necessárias para cada uma.

4.9.1 Registrar Pagamento

Para registrar um novo pagamento, são necessárias as seguintes validações:

- Verificar se os dados do pagamento são válidos e estão corretos.
- Verificar se o *ShoppingCart* associado ao pagamento existe e está ativo.
- Verificar se não existe nenhum pagamento ativo associado ao mesmo *ShoppingCart*.

4.9.2 Listar Todos os Pagamentos

A operação de listar todos os pagamentos permite obter uma lista paginada de todos os pagamentos registrados no sistema. Esta funcionalidade é essencial para a visualização e gerenciamento dos pagamentos disponíveis.

4.9.3 Listar Pagamentos por Usuário

Esta operação permite recuperar uma lista paginada de pagamentos associados a um usuário específico. O controlador recebe o ID do usuário como parâmetro da solicitação e chama o serviço responsável por listar os pagamentos desse usuário.

4.9.4 Desabilitar Pagamento

A operação de desabilitar pagamento permite que pagamentos existentes sejam desabilitados no sistema. Ao receber uma solicitação para desabilitar um pagamento específico, o controlador chama o serviço correspondente, que executa a desativação do pagamento.

Antes de desabilitar um pagamento, são realizadas as seguintes validações:

- Verificar se o pagamento existe.
- Verificar se o pagamento está ativo.

5 Conclusão

Este trabalho apresenta a atualização, adaptação e otimização de um *e-commerce* desenvolvido a cinco anos atrás. A implementação do sistema foi feita em equipe, adotando a metodologia ágil de gerenciamento de projetos *scrum* que além de estimular o trabalho em grupo promove uma comunicação eficaz com os colaboradores por meio das reuniões diárias e facilita a solução de problemas complexos. Para construir a aplicação, utilizamos o *framework spring boot* 3.2.5 com a linguagem Java na versão 17 e o sistema gerenciador de banco de dados relacional *PostgreSQL*. Exploramos conceitos avançados tanto da linguagem quanto do *framework* e padronizamos a implementação utilizando a arquitetura de 3 camadas (*Controller, Service, Repository*) e criando regras para escrita de código e nomenclatura de arquivos.

Portanto, podemos concluir que a aplicação atende a necessidade do cliente e engloba toda a regra de negócios estabelecida pelo demandante.

APÊNDICE A – Regras para Escrita e Organização de Código

A.1 Informações sensíveis

Informações sensíveis, como nomes de usuário, senhas e chaves secretas, devem ser armazenadas em variáveis de ambiente e nunca expostas diretamente no código-fonte para garantir a segurança e facilitar o gerenciamento de configuração. Isso evita a exposição acidental dessas informações, além de permitir que configurações sejam facilmente alteradas sem modificar o código, especialmente em diferentes ambientes de desenvolvimento, teste e produção.

Por exemplo, o código abaixo demonstra como referenciar variáveis de ambiente em vez de expor diretamente informações sensíveis (veja a Listagem A.1):

Listing A.1 – Configurações do Banco de Dados e Segurança

```
1 spring.datasource.url=jdbc:postgresql://localhost:5432/shoppingStore
2 spring.datasource.driver-class-name=org.postgresql.Driver
3 spring.datasource.username=${POSTGRES_DATASOURCE_USER}
4 spring.datasource.password=${POSTGRES_DATASOURCE_PASSWORD}
5
6 api.security.token.secret=${JWT_SECRET}
```

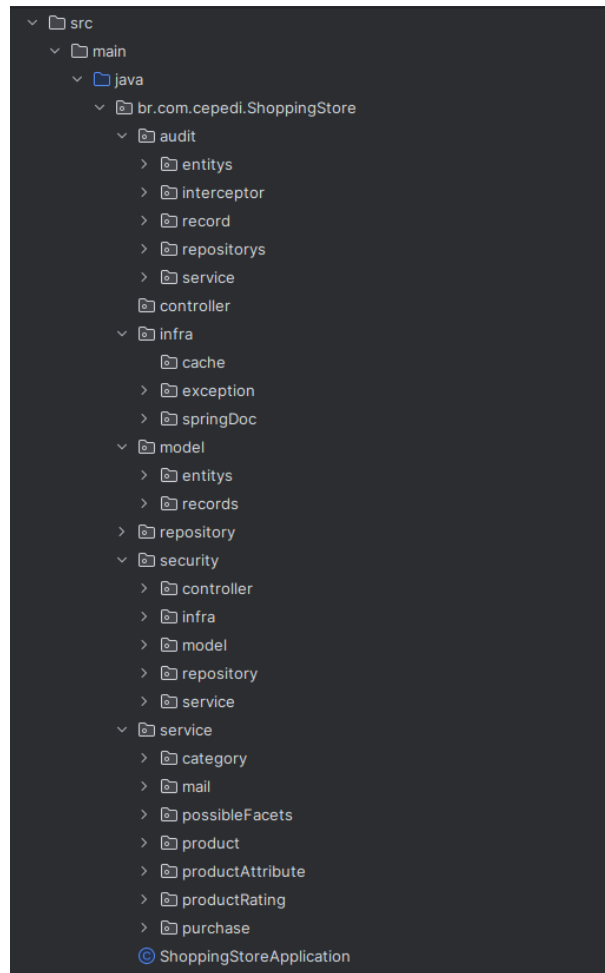
A.2 Estrutura de *Packages*

Foi estabelecida uma estrutura básica de *packages* com o intuito de organizar e manter cada classe em seu devido lugar, facilitando assim a localização.

Basicamente, toda entidade tem um *DTO* de *input* e *output* chamados de ‘*register*’ e ‘*details*’, que definem o que é necessário o usuário passar para realizar um registro e o que deve ser retornado ao *frontend*. Cada entidade possui um *repository*, um *service* e, em alguns casos, um *controller*. A Figura 22 ilustra essa organização.

O **repository** no Spring Boot é responsável por interagir diretamente com o banco de dados, executando operações de CRUD (Create, Read, Update, Delete). O **service** contém a lógica de negócio da aplicação, chamando os métodos do *repository* e realizando as operações necessárias antes de retornar os dados ao *controller*. O **controller** é a camada

Figura 22 – Exemplo da estrutura de *packages*



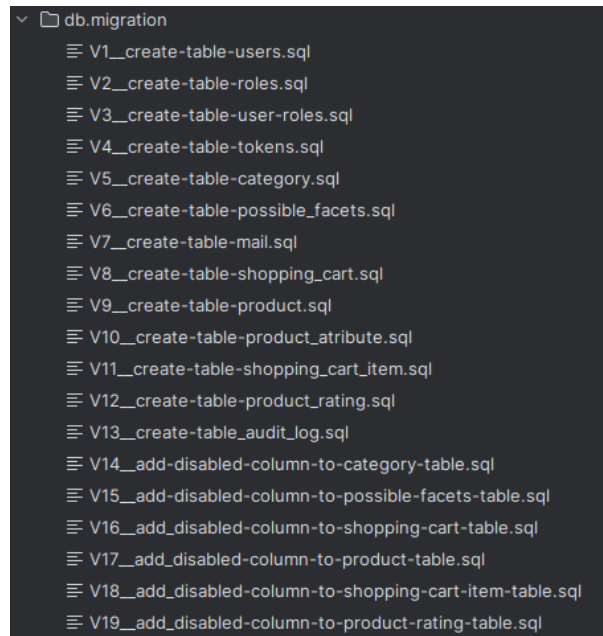
que lida com as requisições HTTP, recebendo dados do *frontend*, chamando os métodos do *service* e retornando as respostas adequadas.

Além disso, o *package* de segurança (*security*) é isolado devido à possibilidade de construí-lo com baixo acoplamento, permitindo que ele seja reutilizado em outros projetos com mínimas modificações. Isso facilita a manutenção, além de promover a possibilidade de reutilização de código e a padronização das práticas de segurança.

A.3 Migrations

Cada modificação no banco de dados, como a criação de tabelas ou qualquer comando DDL, requer uma nova *migration*. Essas *migrations* são nomeadas seguindo uma sequência numérica, começando com 'S(O número correspondente)___', garantindo a ordem e unicidade das instruções. Um exemplo desse padrão é ilustrado na Figura 23.

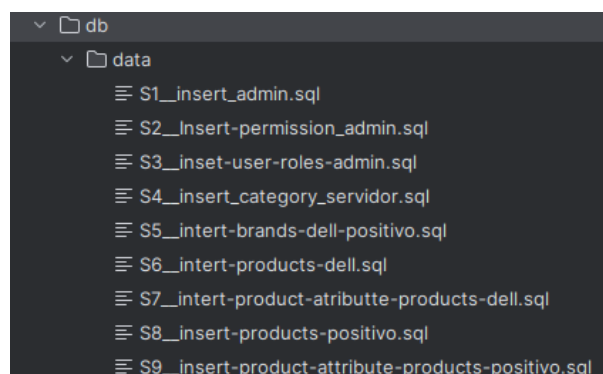
Figura 23 – Exemplo de estrutura do *package* db.migration



A.4 Seeders

Os *seeders* desempenham um papel crucial na inicialização e no preenchimento inicial do banco de dados com dados pré-definidos. Localizados no pacote de dados (*package data*) em `db/resources`, esses arquivos seguem uma convenção de nomenclatura que inclui uma sequência numérica, começando com 'S(O número correspondente)___'. Um exemplo visual da estrutura do diretório *package* db.data é ilustrado na Figura 24.

Figura 24 – Exemplo de estrutura do *package* db.data



Para executar os *scripts* de *seeders* durante o processo de inicialização do banco de dados, é necessário adicionar as seguintes instruções no arquivo `application.properties`:

```
spring.sql.init.mode=always
spring.sql.init.platform=postgres
spring.sql.init.data-locations=classpath:db/data/S_*.sql
```

A configuração `spring.sql.init.mode=always` garante que os scripts de *seeders* sejam executados sempre que a aplicação for iniciada, garantindo a consistência dos dados. O `spring.sql.init.platform=postgres` especifica a plataforma de banco de dados a ser usada durante a inicialização. Já `spring.sql.init.data-locations` indica o local onde os scripts de *seeders* estão localizados no classpath.

É importante notar que, após a primeira população do banco de dados, é recomendável alterar o modo de inicialização para `never` (`spring.sql.init.mode=never`) para evitar a sobregravação dos dados existentes durante futuras inicializações da aplicação. Isso garante que os dados já existentes no banco permaneçam intactos.

A.5 Auditoria

Na auditoria será utilizada a estratégia por *IpAddressInterceptor* combinada com a programação orientada a aspectos (AOP), o que garante menor escrita de código e maior modularidade. A utilização de AOP permite que preocupações transversais, como auditoria, sejam implementadas de forma isolada, sem a necessidade de poluir o código principal das aplicações.

Dessa forma, a auditoria se torna um componente reutilizável e facilmente gerenciável, podendo ser ajustada ou expandida conforme necessário sem grandes impactos no restante da aplicação.

A.6 Controle de Exceções

O controle de exceções é gerenciado pelo arquivo *ErrorHandler*, localizado no *package* `infra/exception`, e é marcado com a anotação `@RestControllerAdvice`. Este componente desempenha um papel crucial na aplicação, interceptando e tratando exceções que podem ocorrer durante sua execução. Ao utilizar a anotação `@RestControllerAdvice`, o *ErrorHandler* é globalmente aplicado em toda a aplicação Spring Boot, garantindo um tratamento consistente de exceções em todos os endpoints, evitando a repetição de código para o tratamento de erros em cada controlador individualmente.

Por meio do *ErrorHandler*, é possível definir diferentes comportamentos para diferentes tipos de exceções, como retornar mensagens de erro personalizadas e códigos de status HTTP apropriados. Abaixo segue um exemplo (veja a Listagem [A.2](#)).

Listing A.2 – Exemplo de ErrorHandler

```

1 @RestControllerAdvice
2 public class ErrorHandler {
3     private static final Logger logger = LoggerFactory.getLogger(
4         ErrorHandler.class);
5     @ExceptionHandler(EntityNotFoundException.class)
6     public ResponseEntity<Object> Error404() {
7         logger.error("EntityNotFoundException occurred.");
8         return ResponseEntity.notFound().build();
9     }
10    @ExceptionHandler(MethodArgumentNotValidException.class)
11    public ResponseEntity<Object> Error400(
12        MethodArgumentNotValidException exception) {
13        logger.error("MethodArgumentNotValidException occurred.",
14            exception);
15        List<FieldError> errors = exception.getFieldErrors();
16        return ResponseEntity.badRequest().body(errors.stream().map(
17            DataExceptionValidate::new).toList());
18    }
19    @ExceptionHandler(BadCredentialsException.class)
20    public ResponseEntity<Object> handleBadCredentialsError() {
21        logger.error("BadCredentialsException occurred.");
22        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("
23            Invalid credentials");
24    }
25    private record DataExceptionValidate(String value, String message) {
26        public DataExceptionValidate(FieldError error) {
27            this(error.getField(), error.getDefaultMessage());
28        }
29    }
30 }

```

A.7 Cacheable

O uso da anotação `@Cacheable` deve ser utilizado em consultas que retornam grandes volumes de dados é essencial para otimizar o desempenho e reduzir a carga no sistema. Ao armazenar os resultados dessas consultas em cache, evita-se a necessidade de consultas repetitivas ao banco de dados, reduzindo o tempo de resposta e minimizando o consumo de recursos do servidor. Isso não apenas melhora a experiência do usuário, proporcionando tempos de resposta mais rápidos, mas também aumenta a escalabilidade da aplicação, permitindo lidar com um maior número de solicitações simultâneas de forma mais eficiente.

A.8 Entidades

A.8.1 Construtores de Entidades

Para garantir consistência e facilitar o desenvolvimento, todas as entidades devem fazer uso de recursos como Lombok, a anotação `@Entity`, e o `@Table` para especificar o nome da tabela no banco de dados. Além disso, devem ser configurados para receber os objetos DTO correspondentes em seus construtores, como ilustrado abaixo (veja a Listagem A.3).

Listing A.3 – Exemplo de Entidade Patient com Lombok e Anotações JPA

```
1 @Entity
2 @Table(name = "patients")
3 @Getter
4 @Setter
5 @NoArgsConstructor
6 @AllArgsConstructor
7 @EqualsAndHashCode(of = "id")
8 @ToString
9 public class Patient {
10
11     ...atributos
12
13     public Patient(DataRegisterPatient data) {
14         this.name = data.name();
15         this.email = data.email();
16         this.phoneNumber = data.phoneNumber();
17         this.cpf = data.cpf();
18         this.address = new Address(data.dataAddress());
19         this.activated = true;
20     }
```

A.8.2 Construtores de Entidades que possuem composição

Para entidades que têm composição, ou seja, precisam de dados adicionais que não são fornecidos pelo DTO, os valores relevantes devem ser passados juntamente com o DTO, como exemplificado abaixo (veja a Listagem A.4):

Listing A.4 – Exemplo de Construtor de Entidade Product com Dados Adicionais

```
1 public Product(DataRegisterProduct data, Category category){
2     this.name = data.name();
3     this.description = data.description();
4     this.price = data.price();
5     this.sku = data.sku();
6     this.imageUrl = data.imageUrl();
7     this.quantity = data.quantity();
8     this.manufacturer = data.manufacturer();
9     this.featured = data.featured();
10    this.category = category;
11    this.disabled = false;
12 }
```

Segue um exemplo de um DTO de registro (veja a Listagem A.5):

Listing A.5 – Exemplo de DTO para Registro de Categoria

```
1 public record DataRegisterCategory(
2     @NotBlank
3     String name,
4     @NotBlank
5     String description
6 ) {
7 }
```

E também um exemplo de um DTO de detalhes (veja a Listagem A.6):

Listing A.6 – Exemplo de DTO para Detalhes de Categoria

```
1 public record DataCategoryDetails (
2     Long id,
3     String name,
4     String description
5 ) {
6     public DataCategoryDetails(Category category){ this(
7         category.getId(), category.getName(), category.getDescription());
8     }
9 }
```

A.9 Mensagens de erro nos DTO's

Padronizar as mensagens de erro nos DTOs é essencial para garantir consistência e facilitar a manutenção do código. Armazenar essas mensagens em um arquivo como `ValidationMessages.properties` ajuda na gestão e tradução delas conforme necessário. Veja o exemplo no código abaixo:

validation.required.name=0 nome é obrigatório.
validation.required.email=E-mail é obrigatório.
validation.length.min.name=0 nome deve ter pelo menos {0} caracteres.
validation.pattern.email=E-mail inválido. Insira um e-mail válido.

Essa prática promove uma experiência consistente para os usuários e simplifica a manutenção do sistema.

A.10 Repositorys

Para cada entidade, é necessário um repositório correspondente. Pode ser utilizado a sintaxe JPQL ou seguir a convenção de nomenclatura do JPA. Deve ser evitado a convenção JPA caso o nome resultante seja muito longo. Aqui estão exemplos tanto da conversão JPA quanto do uso de JPQL. Um exemplo consta na Listagem A.7.

Listing A.7 – Exemplos de Repositórios

```
1 public interface PatientRepository extends JpaRepository<Patient, Long>
2 {
3     // Exemplo da convenção JPA
4     Page<Patient> findAllByActivatedTrue(Pageable pageable);
5
6     // Exemplo da utilização de JPQL
7     @Query("""
8         SELECT p.activated FROM Patient p WHERE p.id = :id
9         """)
10    Boolean findActivatedById(@Param("id") Long id);
11 }
```

A.11 Services

Todas as entidades devem possuir um serviço responsável por implementar sua regra de negócios. Esses serviços devem estar localizados em um *package* específico para a entidade, com o nome da entidade seguido de **Service**. Por exemplo, para a entidade *Patient*, o serviço correspondente será **PatientService** e ficará dentro do *package patient* dentro do *package service*.

A implementação de um serviço deve seguir um fluxo específico para cada operação: o serviço recebe um DTO de entrada, comunica-se com o repositório para executar a operação desejada e, em seguida, retorna um DTO contendo os detalhes da entidade. Além disso, o serviço deve ser anotado com **@Service**, injetar os repositórios necessários e possuir listas de validações que serão injetadas por inversão de dependência.

Abaixo, apresentamos um exemplo prático com várias operações de um serviço para a entidade *Patient*. Um exemplo está na Listagem [A.8](#).

Listing A.8 – Exemplo de Serviço para a Entidade *Patient*

```
1 @Service
2 public class PatientService {
3
4     @Autowired
5     private PatientRepository repository;
6     @Autowired
7     private List<ValidationUpdatePatient> validationUpdatePatient;
8     @Autowired
9     private List<ValidationDisabledPatient> validationDisabledPatients;
10
11     public DataDetailsPatient register(DataRegisterPatient data) {
12         Patient patient = new Patient(data);
13         repository.save(patient);
14         return new DataDetailsPatient(patient);
15     }
16     public Page<DataDetailsPatient> list(Pageable pageable) {
17         return repository.findAllByActivatedTrue(pageable).map(
18             DataDetailsPatient::new);
19     }
20     public DataDetailsPatient details(Long id) {
21         Patient patient = repository.getReferenceById(id);
22         return new DataDetailsPatient(patient);
23     }
24     public DataDetailsPatient update(Long id, DataUpdatePatient data) {
25         validationUpdatePatient.forEach(v -> v.validation(id, data));
26         Patient patient = repository.getReferenceById(id);
27         patient.updateData(data);
28         return new DataDetailsPatient(patient);
29     }
30     public void disabled(Long id) {
31         validationDisabledPatients.forEach(v -> v.validation(id));
32         Patient patient = repository.getReferenceById(id);
33         patient.logicalDelete();
34     }
35 }
```

- **register:** Recebe um *DataRegisterPatient*, cria uma nova entidade *Patient*, salva no repositório e retorna os detalhes do paciente registrado.
- **list:** Retorna uma página de *DataDetailsPatient* contendo todos os pacientes ativados, paginados conforme os parâmetros fornecidos.

- **details:** Recebe um `Long id`, obtém a referência do paciente pelo ID e retorna os detalhes do paciente.
- **update:** Recebe um `Long id` e um `DataUpdatePatient`, valida as atualizações, atualiza os dados do paciente e retorna os detalhes do paciente atualizado.
- **disabled:** Recebe um `Long id`, valida a desativação, realiza a exclusão lógica do paciente.

A.11.1 Services de Classes com composição

Quando uma entidade possui composição de outras entidades, as operações de busca e validação das entidades de composição são realizadas dentro do serviço da entidade fraca (entidades que dependem de outras entidades para existir). Isso garante que todas as operações relacionadas à composição sejam tratadas de forma centralizada e consistente.

No exemplo da Listagem A.9, o serviço de agendamento (`Appointment`) está realizando o registro de um novo agendamento. Antes de persistir o agendamento, ele executa algumas validações e busca as entidades de composição (como o paciente e o médico) necessárias para criar o agendamento.

Listing A.9 – Serviço de Agendamento

```

1 public DataDetailsAppointment register(DataRegisterAppointment data ){
2     validators.forEach(validator -> validator.validation(data));
3     Patient patient = repositoryPatient.getReferenceById(data.idPatient
4         ());
5     Doctor doctor = chooseDoctor(data);
6     Appointment appointment = new Appointment(null, doctor, patient,
7         data.date(),null);
8     repository.save(appointment);
9     return new DataDetailsAppointment(appointment);
10 }

```

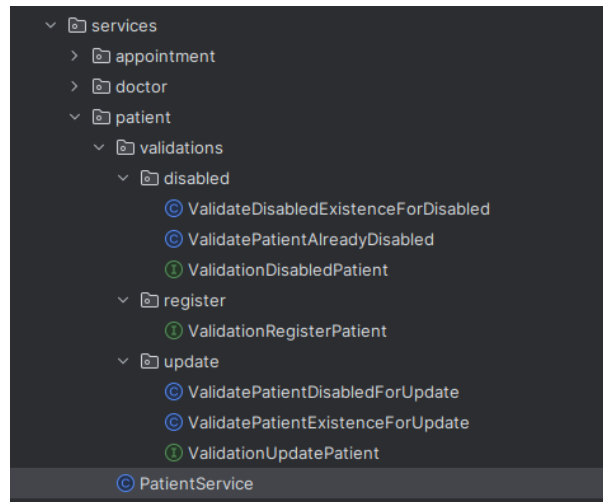
A.11.2 Localização das Classes de Validação

Dentro do *package* `service/nomeDaEntidade`, onde `nomeDaEntidade` é o nome escolhido pelo desenvolvedor, deve haver um *package* `validations`, que conterá *subpackages* específicos para cada tipo de validação, como `register`, `update` e `disabled`. Conforme ilustrado na Figura 25.

A.11.3 Interface de validação

Cada *subpackage* terá uma interface responsável por definir a regra do método de validação, como mostrado no exemplo a seguir (Listagem A.10):

Figura 25 – Exemplo de onde devem estar as validações



Listing A.10 – Exemplo de interface de validação

```
1 public interface ValidationUpdatePatient {
2     void validation(Long id, DataUpdatePatient data);
3 }
```

As classes que implementam esta interface devem realizar a validação e lançar uma exceção do tipo `ValidationException` caso a validação falhe. Essa validação será capturada pelo `ErrorsHandler` e retornada ao front-end.

A.11.4 Classes que Implementam as Interfaces de Validação

As classes devem ser marcadas com a *annotation* `@Component` para serem carregadas previamente pelo *Spring Boot* e seguir as regras da interface implementada, conforme exemplificado a seguir (Listagem A.11). Dessa forma, elas serão injetadas na lista de validações, conforme demonstrado no exemplo do serviço. É importante escolher nomes específicos e descritivos para as classes, de modo a evitar confusões com outras classes do projeto.

Listing A.11 – Classe de Validação para Atualização de Paciente

```
1 @Component
2 public class ValidatePatientExistenceForUpdate implements
   ValidationUpdatePatient {
3     @Autowired
4     private PatientRepository repository;
5     public void validation(Long id, DataUpdatePatient data) {
6         if (!repository.existsById(id)) {
7             throw new ValidationException("The required patient does not
               exist");
8         }
9     }
10 }
```

A.12 Controllers

Os controllers devem ser marcados com a anotação `@RestController` para indicar que são controladores REST. Isso é importante porque os controladores REST são especificamente projetados para lidar com requisições HTTP e retornar respostas no formato adequado para comunicação com APIs. Além disso, é necessário o uso de `@RequestMapping` para mapear as requisições para métodos específicos do controller. É recomendado que o `@RequestMapping` inclua uma versão da API para garantir uma gestão mais eficaz da evolução da API. Também é essencial incluir `@SecurityRequirement(name = "bearer-key")` para indicar que a autenticação será realizada por meio de um token de autorização do tipo "Bearer".

É recomendável incluir um Logger nos controllers para registrar informações relevantes sobre as operações realizadas. Isso facilita a depuração e o monitoramento do sistema, permitindo que os desenvolvedores identifiquem problemas e acompanhem o comportamento da aplicação em tempo real.

Os controllers devem ter os serviços necessários injetados por meio da anotação `@Autowired`. Isso permite que os controllers chamem os métodos dos serviços para realizar operações de negócios.

Os DTOs devem ser passados como `@RequestBody` nas requisições HTTP para que os dados sejam recebidos e processados pelo controller. O uso de `@Valid` junto com `@RequestBody` é recomendado para que as validações definidas nos DTOs sejam aplicadas automaticamente pelo Spring. No método de registro (`register`), é recomendável usar `UriComponentsBuilder` para construir a URI do recurso criado e incluí-lo na resposta. Isso permite que os clientes saibam onde encontrar o recurso recém-criado.

Os métodos que realizam modificações nos dados devem ser marcados com

`@Transactional` para garantir a consistência dos dados e a atomicidade das operações. Eles devem retornar os detalhes do recurso modificado para que os clientes possam confirmar o sucesso da operação e receber informações atualizadas.

Por fim, é recomendado que os *controllers* implementem métodos a fim de utilizar todos os métodos públicos dos serviços que estão chamando, garantindo assim uma cobertura completa das funcionalidades fornecidas pelos serviços.

No *Listing A.12* está um exemplo de *controller* implementando esses princípios:

Listing A.12 – Exemplo de controller

```
1 @RestController
2 @RequestMapping("v1/patients")
3 @SecurityRequirement(name = "bearer-key")
4 public class PatientControllerV1 {
5     private static final Logger log = LoggerFactory.getLogger(
6         PatientControllerV1.class);
7     @Autowired
8     private PatientService service;
9     @PostMapping
10    @Transactional
11    public ResponseEntity<DataDetailsPatient> register(@RequestBody
12        @Valid DataRegisterPatient data, UriComponentsBuilder uriBuilder)
13    {
14        log.info("Registering new patient...");
15        DataDetailsPatient details = service.register(data);
16        URI uri = uriBuilder.path("/patients/{id}").buildAndExpand(
17            details.id()).toUri();
18        log.info("New patient registered with ID: {}", details.id());
19        return ResponseEntity.created(uri).body(details);
20    }
21    @GetMapping
22    public ResponseEntity<Page<DataDetailsPatient>> listPatients(
23        @PageableDefault(size = 10, sort = {"name"}) Pageable pageable) {
24        log.info("Fetching list of patients...");
25        Page<DataDetailsPatient> page = service.list(pageable);
26        log.info("List of patients fetched successfully.");
27        return ResponseEntity.ok(page);
28    }
29    @GetMapping("/{id}")
30    public ResponseEntity<DataDetailsPatient> detailsDoctor(
31        @PathVariable Long id) {
32        log.info("Fetching details of patient with ID: {}", id);
33        DataDetailsPatient details = service.details(id);
34        log.info("Details of patient with ID {} fetched successfully.",
35            id);
36        return ResponseEntity.ok(details);
37    }
38 }
```

```

30     }
31     @PutMapping("/{id}")
32     @Transactional
33     public ResponseEntity<DataDetailsPatient> update(@PathVariable Long
        id, @RequestBody @Valid DataUpdatePatient data) {
34         log.info("Updating patient with ID: {}", id);
35         DataDetailsPatient details = service.update(id,data);
36         log.info("Patient with ID {} updated successfully.", id);
37         return ResponseEntity.ok(details);
38     }
39     @DeleteMapping("/{id}")
40     @Transactional
41     public ResponseEntity<Object> disabled(@PathVariable Long id) {
42         log.info("Disabling patient with ID: {}", id);
43         service.disabled(id);
44         log.info("Patient with ID {} disabled successfully.", id);
45         return ResponseEntity.noContent().build();
46     }
47 }

```

A.13 Testes

A.13.1 Configurações do ambiente de teste

Ter um ambiente de teste bem configurado é fundamental para garantir a qualidade, confiabilidade e a estabilidade do sistema. Um aspecto crítico é o uso de um banco de dados de teste que replica o ambiente de produção. Isso ajuda a garantir que os testes reflitam com precisão o comportamento do sistema em produção, identificando problemas antes que eles impactem os usuários finais.

Um arquivo de propriedades específico para o ambiente de teste, como o `application-test.properties`, permite configurar facilmente o ambiente de teste com o mesmo SGBD (Sistema de Gerenciamento de Banco de Dados) e as mesmas configurações do ambiente de produção. Isso simplifica o processo de teste e ajuda a manter a consistência entre os ambientes de desenvolvimento, teste e produção.

```

spring.config.activate.on-profile=test
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/shoppingStore_test
spring.datasource.username=${POSTGRES_DATASOURCE_USER}
spring.datasource.password=${POSTGRES_DATASOURCE_PASSWORD}
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

```



```
spring.jpa.show-sql=false
server.port=8081
```

Este exemplo demonstra como configurar o ambiente de teste para usar o PostgreSQL como banco de dados de teste. As configurações incluem o driver JDBC, a URL do banco de dados, o nome de usuário e senha, além de outras configurações específicas do Hibernate. Utilizar um ambiente de teste adequado, como esse exemplo ilustra, é essencial para garantir testes confiáveis e eficazes.

A.14 Padrões de teste

É recomendável aplicar a ordem randômica em testes de unidade para evitar dependências entre eles e garantir a independência dos resultados. Isso pode ser alcançado aplicando a anotação `@TestMethodOrder(MethodOrderer.Random.class)`. A vantagem dessa abordagem é que os testes serão executados em ordens diferentes a cada execução, ajudando a identificar possíveis falhas relacionadas à ordem de execução.

Ao escrever testes, é importante fornecer nomes significativos para os métodos de teste usando a anotação `@DisplayName`. Isso torna os testes mais legíveis e compreensíveis, facilitando a identificação dos cenários de teste e dos casos de uso cobertos pelos testes.

O uso do *Faker* é altamente recomendado sempre que possível para gerar dados de teste de forma dinâmica e realista. O *Faker* permite criar dados fictícios de maneira rápida e fácil, o que é especialmente útil para cenários de teste que requerem uma grande quantidade de dados. Isso ajuda a aumentar a cobertura dos testes e a garantir que diferentes cenários sejam testados de forma abrangente. Veja o exemplo do (Listing A.13):

Listing A.13 – Exemplo de teste de unidade utilizando o Faker

```
1 @TestMethodOrder(MethodOrderer.Random.class)
2 @DisplayName("Test entity Product")
3 class ProductTest {
4     private final Faker faker = new Faker();
5     @Test
6     @DisplayName("Inequality test")
7     public void testProductInequality() {
8         Product product1 = new Product(); product1.setId(faker.number().
9             randomNumber()); product1.setName(faker.commerce().productName
10            ());
11         Product product2 = new Product(); product2.setId(faker.number().
12            randomNumber()); product2.setName(faker.commerce().productName
13            ());
14         assertEquals(product1, product2);
15     }
16 }
```

A.14.1 Testes de Repository

Além das anotações mencionadas anteriormente, os testes de repositório devem incluir algumas outras anotações importantes para garantir que os testes sejam executados em um ambiente de teste isolado e controlado. Estas anotações são:

- `@DataJpaTest`: Essa anotação configura o ambiente de teste para testar camadas de persistência JPA. Ela carrega apenas as partes relevantes da aplicação relacionadas à JPA.
- `@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)`: Esta anotação impede que o Spring Boot substitua automaticamente o banco de dados de teste pelo da aplicação. Isso garante que os testes sejam executados no banco de dados configurado no **application-test.properties**.
- `@ExtendWith(SpringExtension.class)`: Essa anotação habilita a integração do Spring com os testes JUnit 5.
- `@ActiveProfiles("test")`: Define o perfil ativo como "test", permitindo a configuração específica para o ambiente de teste.

Essas anotações garantem que os testes de repositório sejam executados em um ambiente de teste isolado e controlado, utilizando o banco de dados de teste configurado e evitando a persistência de dados no banco de dados de produção. Além disso, é importante limpar a tabela utilizada no teste ao final de cada caso de teste para manter a consistência e a independência dos testes.

No Listing [A.14](#) está um exemplo de teste de repositório com as anotações mencionadas.

Listing A.14 – Exemplo de teste de repositório com anotações

```
1 @DataJpaTest
2 @TestMethodOrder(MethodOrderer.Random.class)
3 @ExtendWith(SpringExtension.class)
4 @ActiveProfiles("test")
5 @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.
    NONE)
6 class UserRepositoryTest {
7
8     @Autowired
9     private UserRepository userRepository;
10
11     @Autowired
12     private TestEntityManager entityManager;
13
14     @Test
15     @DisplayName("Test find by username")
16     public void testFindByUsername() {
17         // C digo do teste
18     }
19
20     // Outros m todos de teste...
21 }
```

A.14.2 Testes de Service

Os testes de *service* devem utilizar *mocks* para isolar o *service* que está sendo testado. Para isso, são usadas as seguintes anotações:

- `@ExtendWith(MockitoExtension.class)`
- `@InjectMocks`
- `@Mock`

Um exemplo de teste de serviço com *mocks* usando o *Mockito* é apresentado no Listing A.15:

Listing A.15 – Exemplo de teste de serviço com mocks usando o Mockito

```
1 @ExtendWith(MockitoExtension.class)
2 class AppointmentServiceTest {
3
4     @InjectMocks
5     private AppointmentService appointmentService;
6
7     @Mock
8     private AppointmentRepository appointmentRepository;
9
10    @Test
11    void testRegisterAppointment() {
12        Appointment appointment = new Appointment();
13        appointment.setId(1L);
14
15        Mockito.when(appointmentRepository.save(Mockito.any(Appointment.class))).thenReturn(appointment);
16
17        DataDetailsAppointment result = appointmentService.register(new DataRegisterAppointment());
18
19        assertNotNull(result);
20        assertEquals(1L, result.getId());
21    }
22
23    // Outros m todos de teste...
24 }
```

A.14.3 Testes de Controllers

Os testes de controllers são importantes para garantir o correto funcionamento das rotas e da lógica de controle da aplicação. Para testar os controllers, geralmente utilizamos o framework MockMvc, que simula requisições HTTP sem a necessidade de iniciar um servidor real.

Para configurar e executar testes de controllers com o MockMvc, utilizamos as seguintes anotações:

- **@SpringBootTest**: Essa anotação carrega a aplicação Spring Boot durante os testes, permitindo a inicialização do contexto da aplicação e a injeção de dependências.
- **@AutoConfigureMockMvc**: Essa anotação configura o MockMvc para ser injetado automaticamente no teste, permitindo a simulação de requisições HTTP e a validação das respostas.

Um exemplo de configuração e execução de testes de controllers com o MockMvc pode ser visto no Listing A.16:

Listing A.16 – Exemplo de teste de controllers com o MockMvc

```
1 @SpringBootTest
2 @AutoConfigureMockMvc
3 class UserControllerTest {
4
5     @Autowired
6     private MockMvc mockMvc;
7
8     @Test
9     void testGetUserById() throws Exception {
10         mockMvc.perform(get("/users/{id}", 1))
11                 .andExpect(status().isOk())
12                 .andExpect(jsonPath("$.id").value(1));
13     }
14
15     // Outros m todos de teste...
16 }
```

Neste exemplo, estamos testando o endpoint GET para obter um usuário por ID. Utilizamos o método `perform` do MockMvc para realizar uma requisição HTTP GET para o endpoint especificado. Em seguida, usamos as expectativas (`andExpect`) para validar o status da resposta (200 OK) e o conteúdo do JSON retornado.