

1. Use Cases:

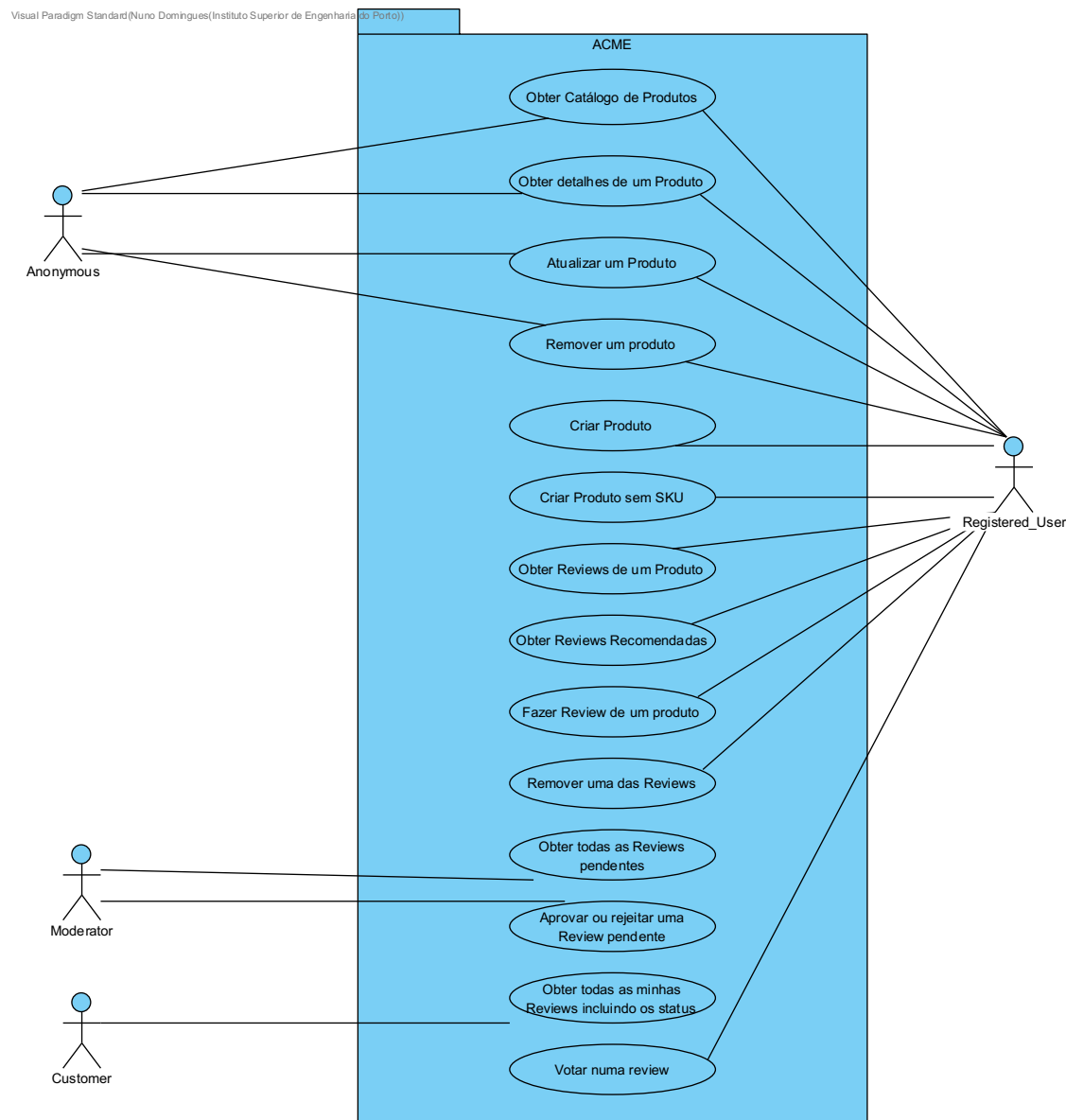


Figura 1 - Use Case Diagram

2. Vista Lógica (Modelo único de persistência):

- **Nível 2**

Neste diagrama estão demonstrados os três principais constituintes da aplicação.

A API HTTP, que permite fazer pedidos HTTP à aplicação, permite criar, remover ou atualizar informação. Está presente a aplicação que consome dados da API HTTP e da base de dados. No componente da base de dados(**DB**) é armazenada toda a informação.

Visual Paradigm Standard(1191013) (Copyright © Engenharia do Porto))

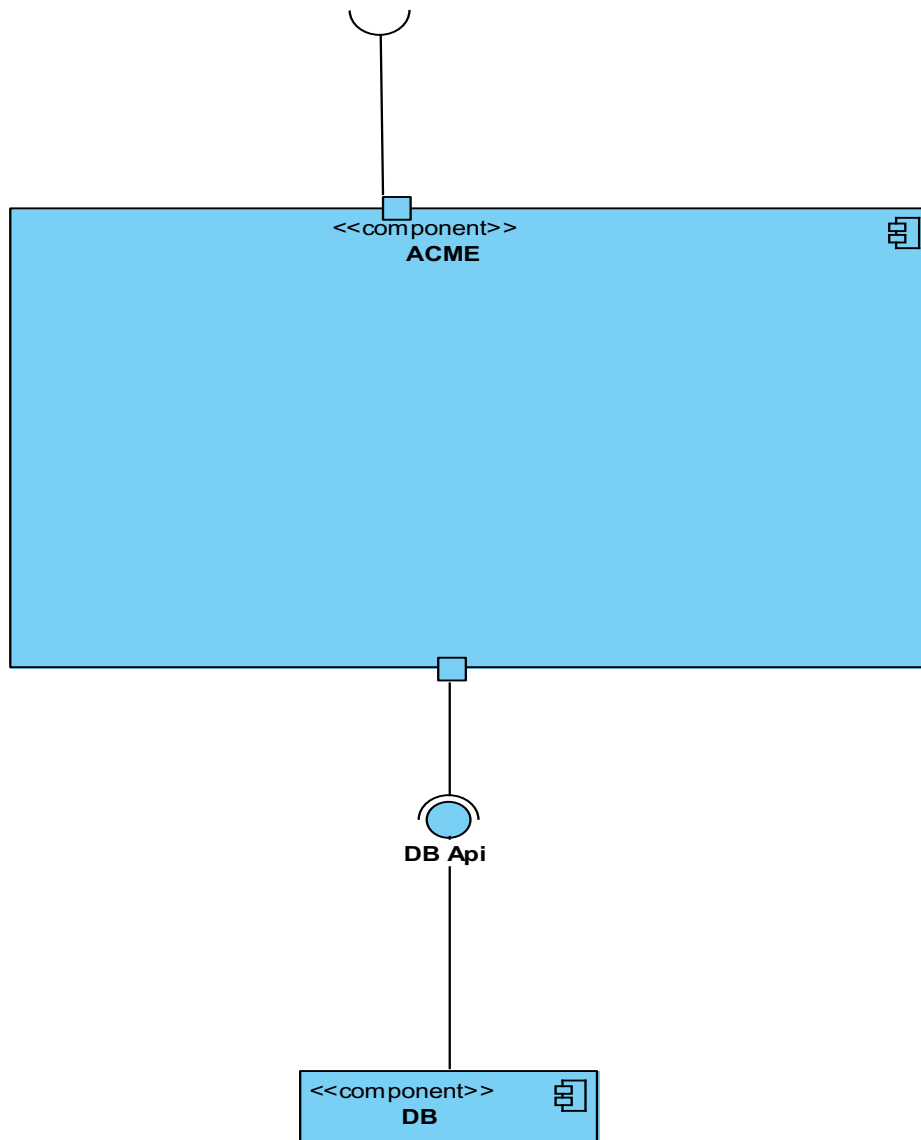


Figura 2 – Vista lógica Nível 2

- **Nível 3**

Neste diagrama de nível 3, está descrito mais detalhadamente o serviço. A API HTTP foi dividida em duas APIs, uma para a criação de produtos, reviews, votos, entre outros e outra para a autenticação, que permite fazer o login dos utilizadores com as diferentes roles.

O componente principal “ACME” foi dividido tendo por base os diferentes packages, em que estes estão ligados por APIs.

Estão também descritas as ligações à base de dados, tanto para os diferentes repositórios como para a autenticação.

Visual Paradigm Standard(1191018(Instituto Superior de Engenharia do Porto))

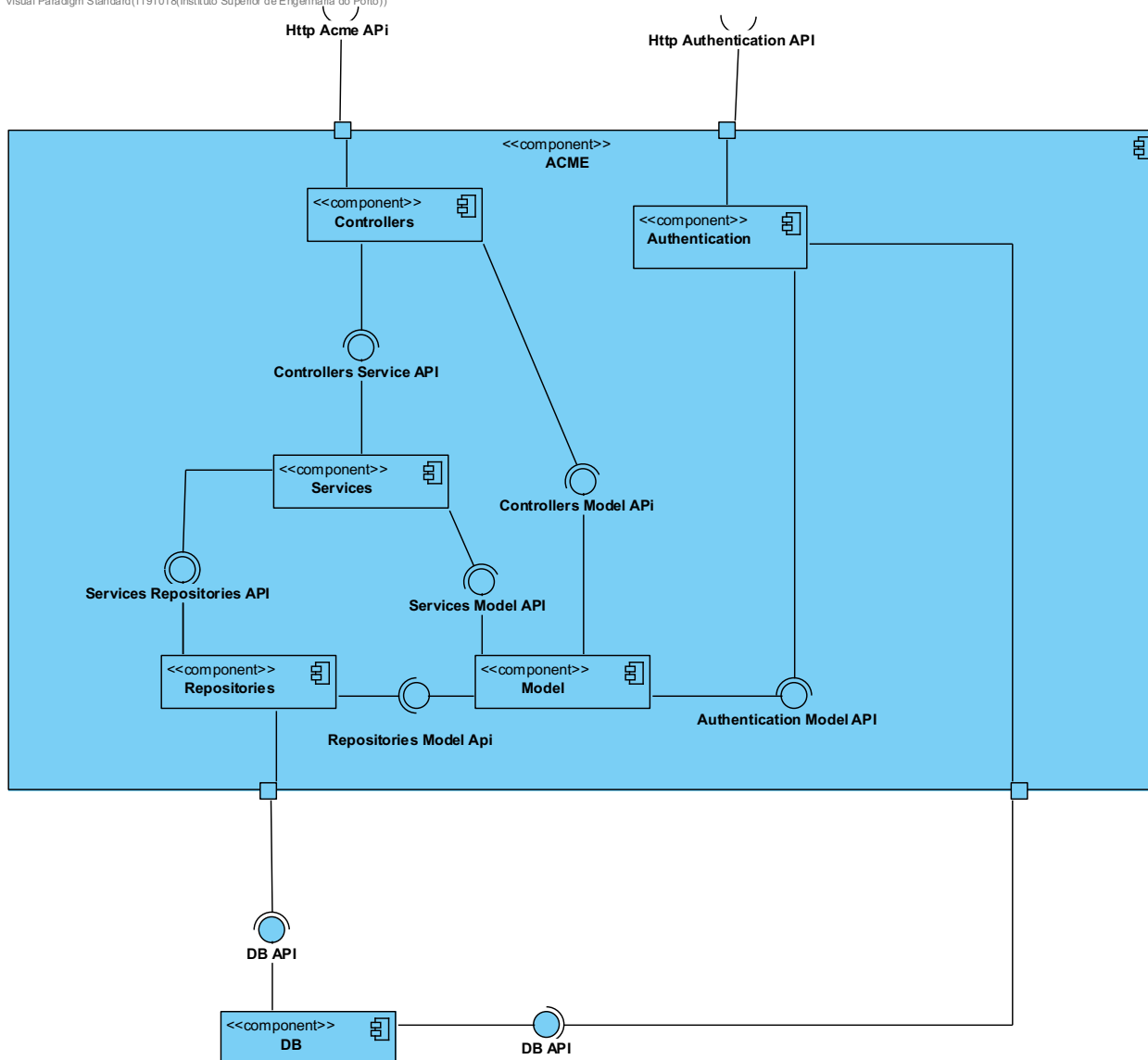


Figura 3 – Vista Lógica Nivel 3(Só uma BD)

3. Vista Lógica (Vários modelos de persistência):

Não foi representada novamente a vista lógica de nível 2, visto que não foram detetadas mudanças a este nível.

- **Nível 3**

Para os vários modelos de persistência foi necessário criar componentes para as diferentes bases de dados. Estas diferentes bases de dados têm ligação tanto ao repositório como à autenticação, visto que ambos consomem informação das mesmas.

Foi necessário adicionar um novo componente “Config” onde foram feitas as configurações necessárias para o Redis. A informação deste componente é consumida pelo componente “Services”.

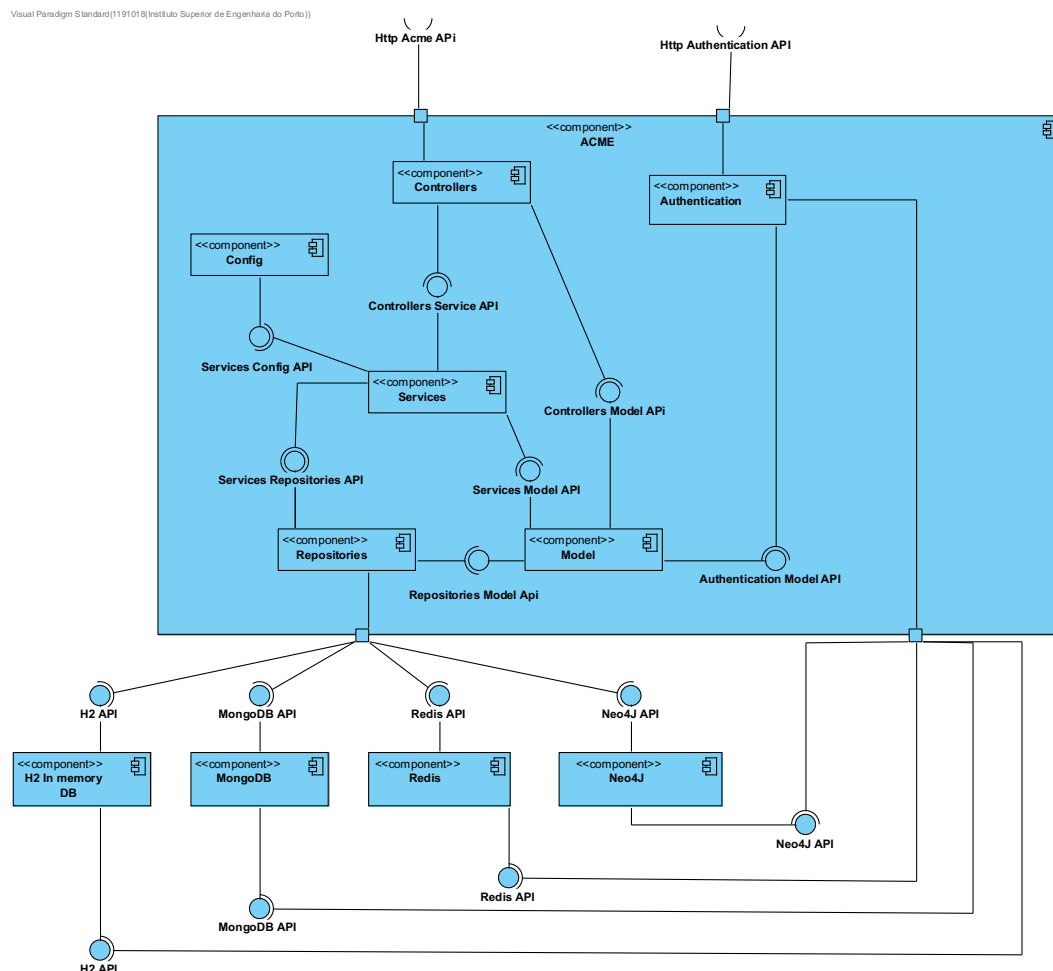


Figura 4 – Vista Lógica Nivel 3(várias BDs)

4. Diagramas de Sequência

4.1 Use Cases Iniciais

1. Criar Produto

Este Use Case permite ao utilizador criar um Produto, entidade que é necessária para a criação das restantes entidades como Review e Voto. O produto é a base do projeto.

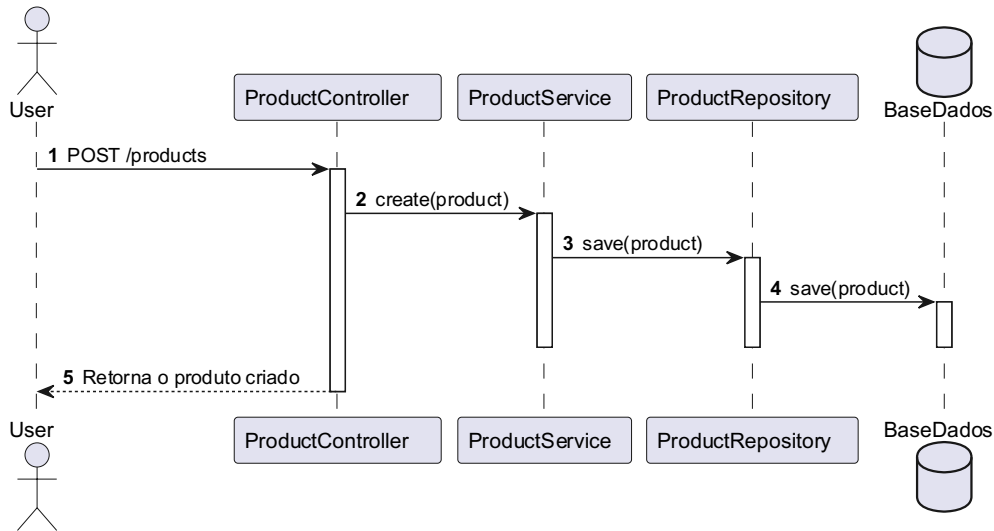


Figura 5- Diagrama Sequência Criar Produto

2. Apagar Produto

Tal como referido no UseCase anterior, o produto é a base do projeto e é importante demonstrar como este pode ser eliminado.

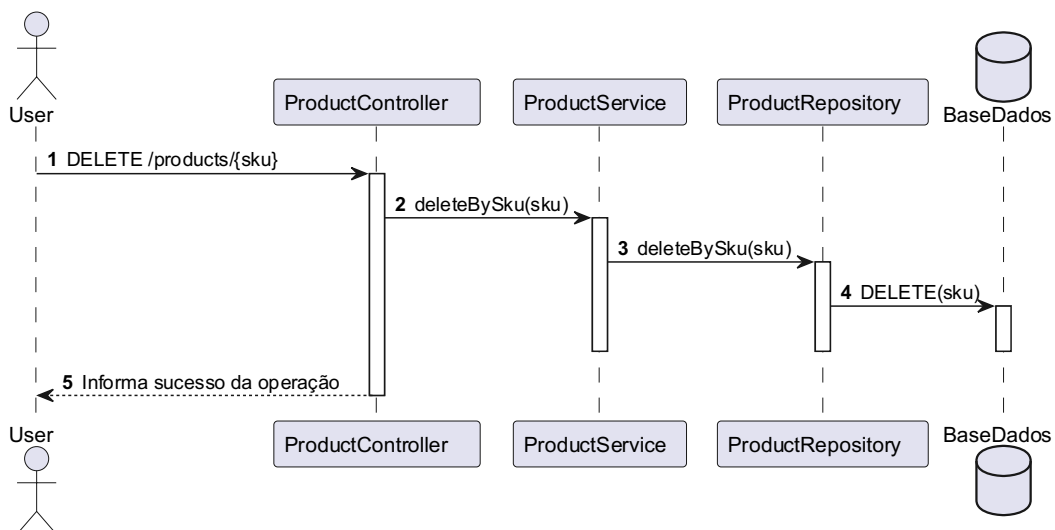


Figura 6 – Diagrama Sequência Apagar Produto

3. Encontrar Produto

Este caso de uso permite aos utilizadores a capacidade de escolher um produto com base em seu código SKU. O SKU é um identificador único associado a cada produto em nosso sistema, e é fundamental para a rastreabilidade e organização de nosso catálogo. Essa funcionalidade é crucial para permitir que os utilizadores localizem rapidamente e com precisão os produtos desejados no sistema.

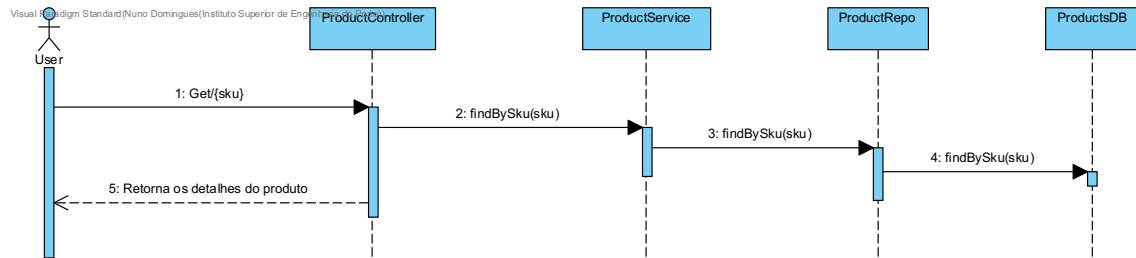


Figura 7 - Diagrama Sequência Encontrar Produto

4. Criar Review

Este caso de uso permite aos utilizadores registados criar Reviews para os produtos disponíveis na nossa plataforma. Para que uma Review seja criada é necessário que o utilizador escolha um produto e de seguida atribua uma classificação ao mesmo.

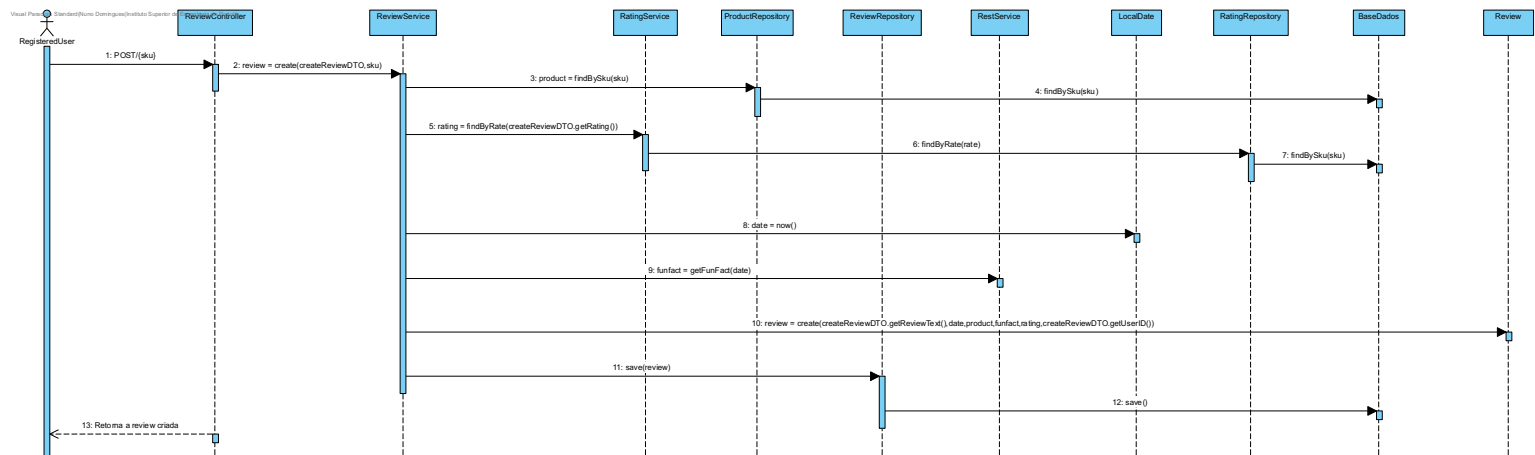


Figura 8 - Diagrama Sequência Criar Review

5. Obter Review

Este caso de uso permite aos utilizador encontrar uma lista de Reviews pelo SKU do produto e pelo status da Review. O SKU é um identificador único associado a cada produto no nosso sistema e o status da Review pode ser approved, reject ou pending. Desta forma, o utilizador procura as review específicas que pretende de um determinado produto.

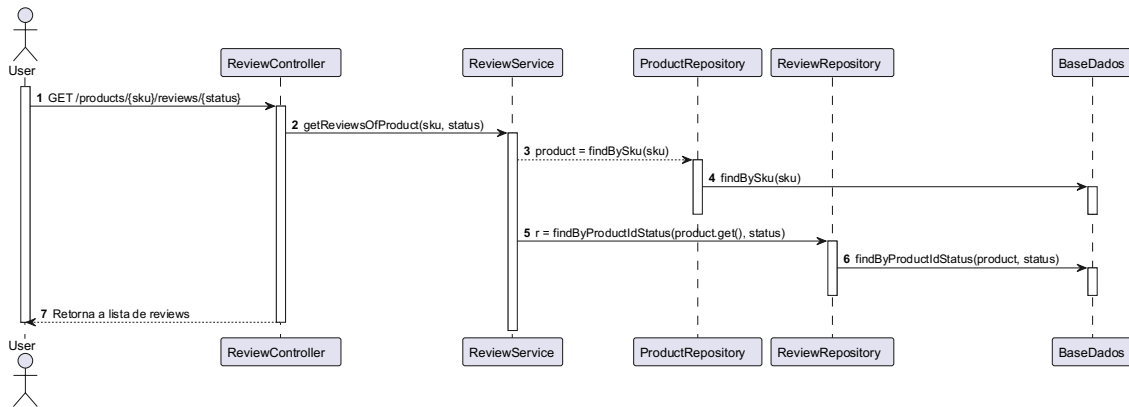


Figura 9 - Diagrama Sequência Obter Review

6. Apagar Review

Este caso de uso concede aos utilizadores registados o controle sobre as reviews que eles criaram, permitindo-lhes excluir as suas reviews quando necessário, contribuindo para uma experiência mais personalizada e flexível na nossa plataforma.

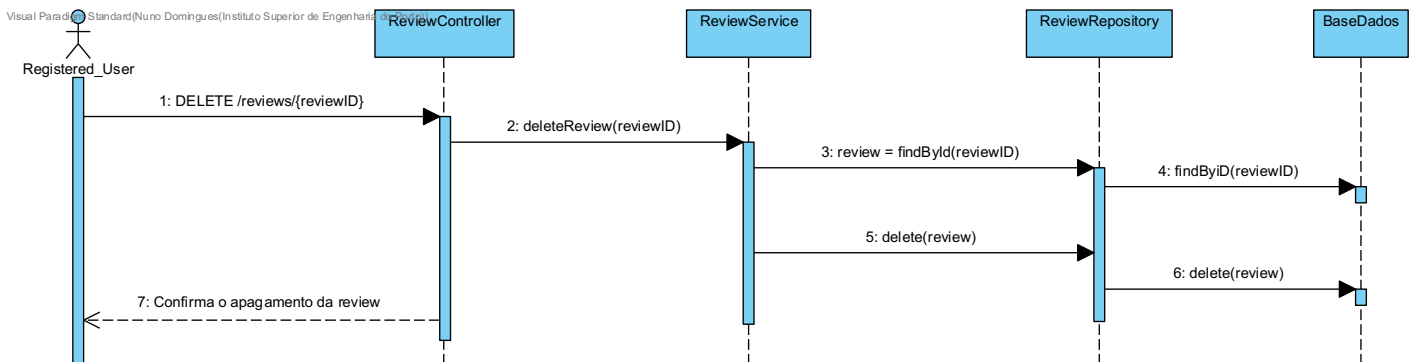


Figura 10 - Diagrama Sequência Apagar Review

7. Adicionar Voto

Este caso de utilização permite ao utilizador adicionar votos a uma review existente, utilizando o identificador único dessa review. O votos podem ser upVotes ou downVotes.

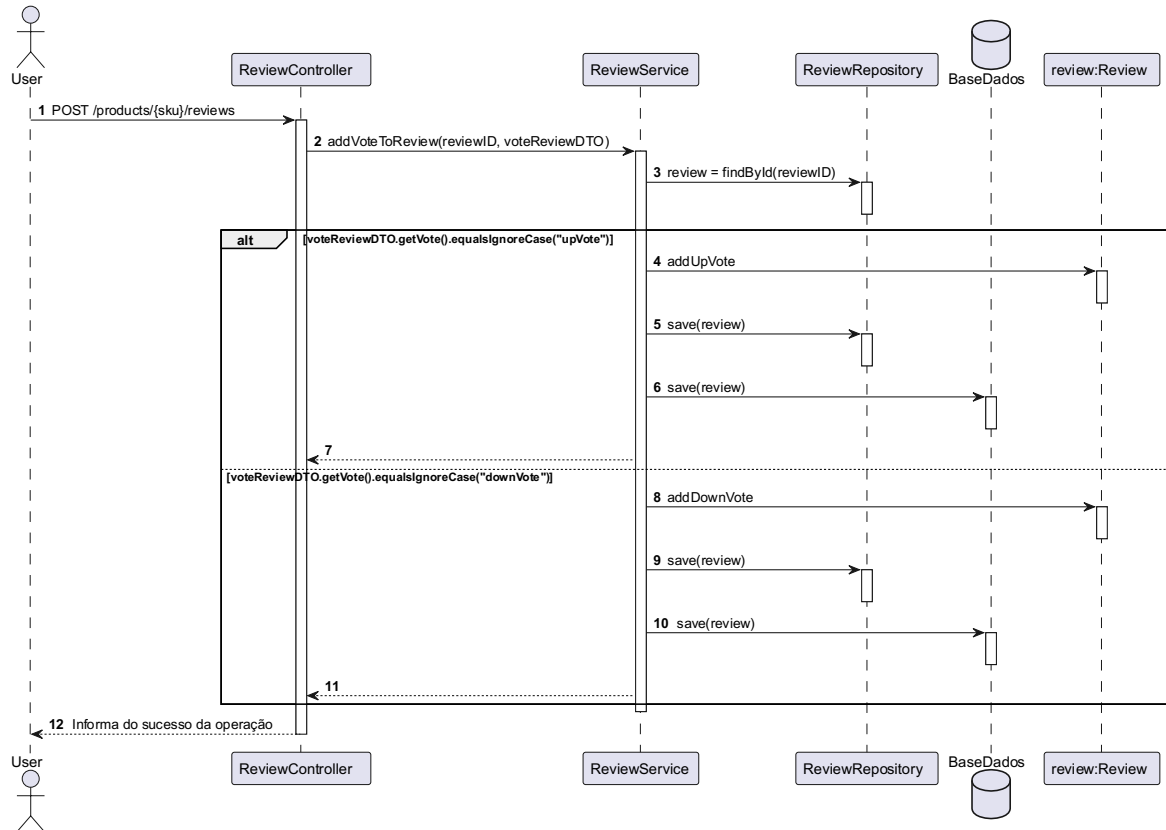


Figura 11 – Diagrama Sequencia Adicionar Voto

8. Login

Para algumas operações é necessária a autenticação de uma Role específica, para tal acontecer é necessário efetuar o login seguindo os passos representados no diagrama

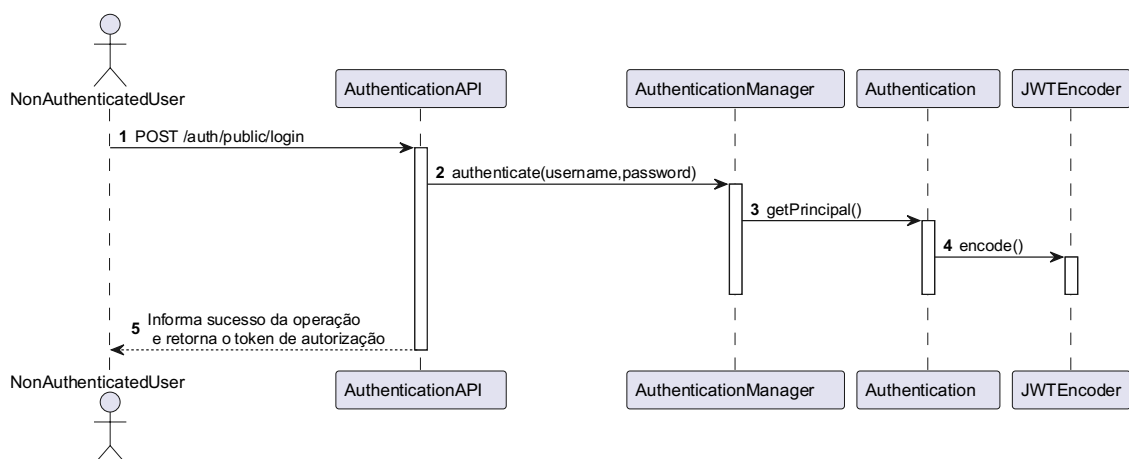


Figura 12 - Diagrama Sequencia Login

4.2 Novos Use Cases

- Criar Produto Sem Sku

Este Use Case permite ao utilizador criar um Produto sem precisar de fornecer um SKU. O produto é uma entidade que é necessária para a criação das restantes entidades como Review e Voto, sendo a base do projeto.

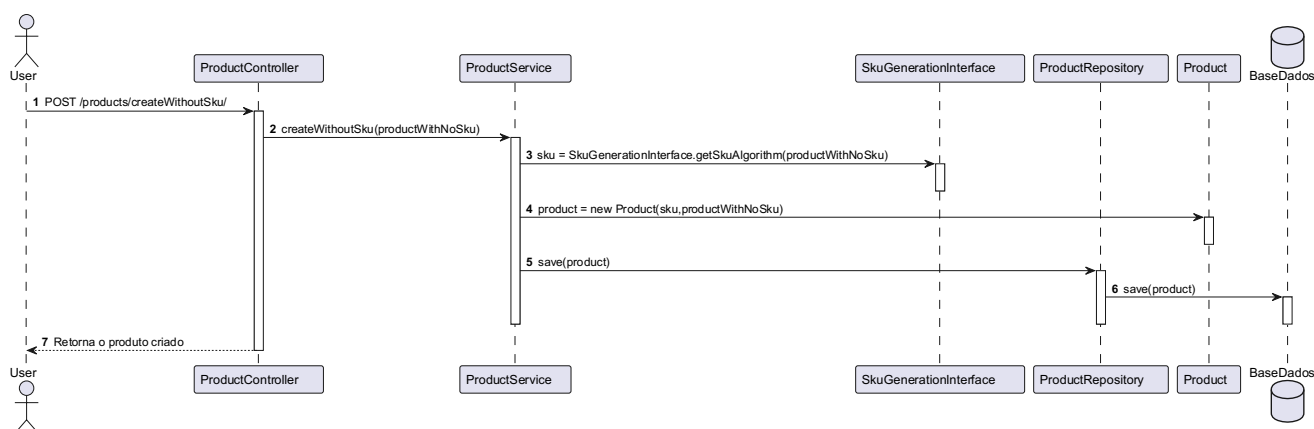


Figura 13 - Diagrama Nível 2(Criação Produto Sem SKU)

Na criação do SKU, implementamos a interface SkuGeneration para garantir a adesão ao princípio **SOLID**, especificamente o **Interface Segregation Principle**. Este princípio defende que as interfaces devem ser segregadas de forma a evitar a criação de interfaces muito grandes ou complexas.

No nosso cenário, a implementação da interface SkuGeneration permite a geração do SKU de três maneiras distintas, de acordo com a configuração definida no arquivo application.properties(como podem ver na Figura 14, que é escolhido o segundo algoritmo). Isso resulta em uma solução mais flexível e extensível, onde cada algoritmo de geração de SKU pode ser isoladamente modificado e mantido, seguindo as diretrizes do SOLID.

```
sku.generation=SkuGenerationAlgTwoImpl
```

Figura 14 - application.properties

- Obter Reviews Recomendadas(3º Algoritmo)

Este Use Case permite ao utilizador receber as suas Reviews recomendadas. O utilizador acede a uma seleção gerada de Reviews de acordo com as preferências de votos e interações semelhantes com outros utilizadores na plataforma.

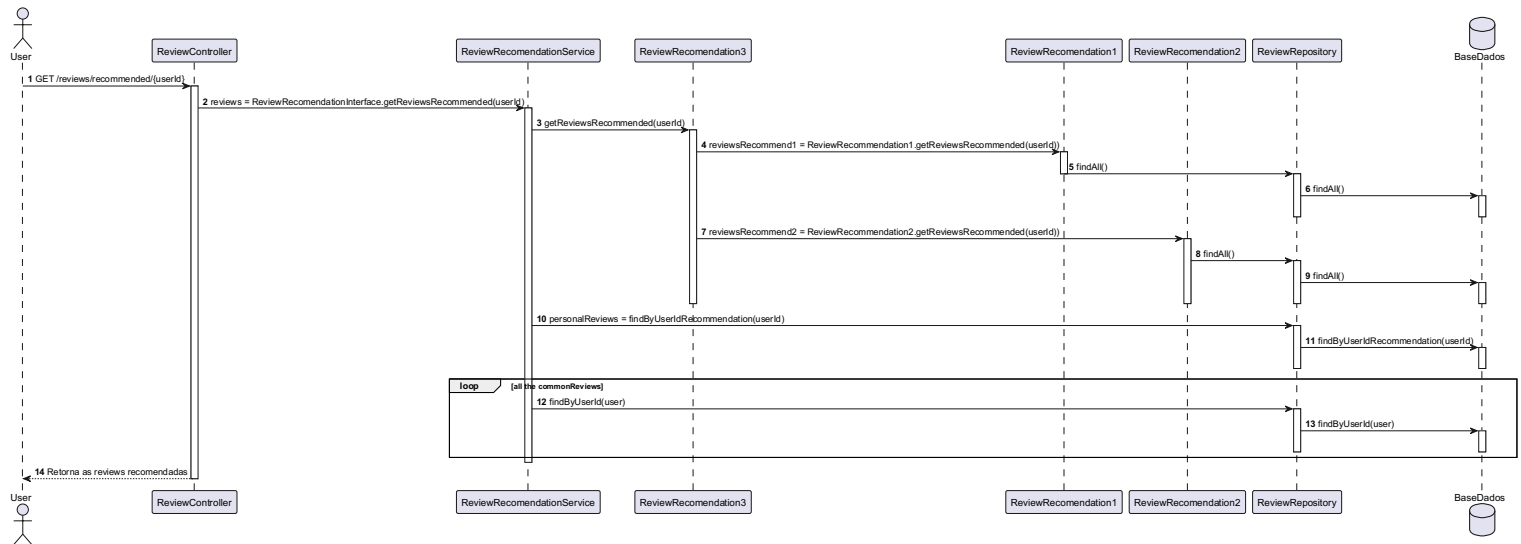


Figura 15 - Diagrama Nível 2(Algoritmo 3 de obter Reviews Recomendadas)

No processo de criação dos algoritmos de recomendação, aplicamos o mesmo princípio que no use case de criar produto sem SKU, aderindo ao **Interface Segregation Principle** do **SOLID**. Isso significa que dividimos as interfaces em partes menores e independentes, tornando os algoritmos de recomendação flexíveis e extensíveis, permitindo que cada algoritmo seja desenvolvido e mantido de forma isolada. Desta forma, garantimos uma estrutura sólida e modular para a geração de recomendações.

Neste use case é possível perceber que o ReviewRecommendation3 chama o ReviewRecommendation1 e o ReviewRecommendation2. Estas chamadas foram feitas através da **Dependency Injection** que oferece uma série de **vantagens** significativas para a arquitetura do sistema, tais como:

Desacoplamento de componentes: Ajuda a evitar acoplamento rígido entre os componentes do sistema. Isso significa que o ReviewRecommendation3 não precisa de se preocupar com a criação ou instância dos ReviewRecommendation1 e ReviewRecommendation2, tornando-o independente desses componentes. Isso simplifica o código e facilita a manutenção, pois as alterações em ReviewRecommendation1 e ReviewRecommendation2 não afetam diretamente o ReviewRecommendation3.

Reutilização de Componentes: Permite reutilizar componentes de forma eficiente. No nosso caso, ReviewRecommendation1 e ReviewRecommendation2 podem ser injetados em múltiplos locais, tornando a lógica de recomendação mais flexível e modular.

5. Implementação das novas funcionalidades

Para a implementação das novas funcionalidades, geração de sku e recomendação de reviews, foi utilizada a mesma abordagem. Foi criada uma interface com o método público comum e de seguida foram criadas as diferentes implementações dessa interface para permitir satisfazer os diferentes requisitos das diferentes formas de gerar o sku e recomendar reviews.

Para seleccionar em runtime a implementação a utilizar foi utilizada uma property no ficheiro `application.properties` para cada um dos casos.

```
sku.generation=SkuGenerationAlgTwoImpl  
review.recommendation=ReviewRecommendation1
```

Figura 16 - `application.properties`

É apenas necessário definir o nome da implementação que se pretende utilizar.

Ao nível do controller, onde a implementação é chamada, foi definido, com base nas propriedades, a implementação a utilizar.

```
@Autowired  
public void setReviewRecommendation(@Value("ReviewRecommendation1Mongo") String bean, ApplicationContext applicationContext){  
    reviewRecommendation = (ReviewRecommendation) applicationContext.getBean(bean);  
}
```

Figura 17 - `ReviewController`

Com esta abordagem de injeção de dependência, em que as implementações concretas das classes são definidas em tempo de execução com base em configurações externas (ficheiro `application.properties`). Esta abordagem é benéfica em comparação com a instanciação direta de todas as implementações:

- **Desacoplamento:** A classe que utiliza as implementações não precisa de se preocupar com a criação das instâncias. Em vez disso, a injeção é feita pelo Spring para fornecer a instância correta. Isto torna o código mais flexível e fácil de manter.
- **Configuração externa:** Ao definir em tempo de execução, pode-se configurar mais facilmente qual a implementação a ser usada sem mudar o código.
- **Testabilidade:** Facilita a criação de testes unitários visto que se pode injetar facilmente implementações Mock para testar isoladamente partes de código.
- **Reutilização:** Permite reutilizar as implementações em diferentes partes da aplicação sem criar instâncias separadas, economizando recursos e promover a consistência na aplicação.
- **Extensibilidade:** Se for necessário adicionar novas implementações no futuro basta configurá-las no ficheiro de properties e não é necessário modificar o código.
- **Separação de responsabilidades:** A injeção de dependência ajuda a manter a separação de responsabilidades visto que cada classe só se concentra na sua

tarefa e a criação e configuração de implementações não é da responsabilidade das classes que a utilizam.

Quanto à implementação dos novos modelos de persistência, foi utilizada uma abordagem semelhante. Foram criadas implementações dos serviços, repositórios e também dos modelos para os modelos de persistência que necessitam. Nos Serviços é injetado a implementação do repositório definida no ficheiro de properties.

Foi feita a implementação de vários repositórios com o objetivo de:

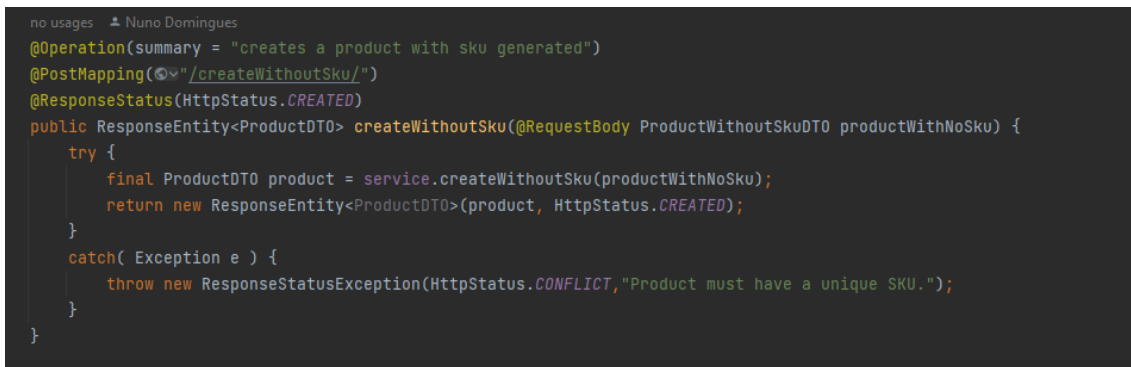
- **Separar responsabilidades:** Ao criar um Repositório para cada funcionalidade específica (cada modelo de dados) permite que cada Repositório seja responsável por um conjunto específico de ações, o que torna o código mais organizado e fácil de entender.
- **Facilidade de Manutenção:** Ter repositórios específicos para tarefas específicas facilita a manutenção e a correção de bugs. Permite saber exatamente onde procurar quando algo não funciona como esperado.
- **Testabilidade:** repositórios específicos são mais fáceis de testar, visto que permite criar testes unitários para cada repositório individualmente. Isto facilita a identificação e correção de problemas.
- **Escalabilidade:** À medida que a aplicação cresce, ter repositórios específicos permite escalar partes individuais da aplicação sem afetar o restante.
- **Melhor Compreensão do Código:** Um código que usa vários repositórios é geralmente mais legível e mais claro, pois reflete a estrutura lógica da aplicação de maneira mais precisa.
- **Customização:** Cada repositório pode ser personalizado para atender às necessidades específicas da funcionalidade que esta oferece, como é o caso da persistência da lista de votos no Redis, para qual necessitou de uma implementação diferente das demais. Isto dá mais controlo sobre o comportamento de cada parte da aplicação.

6. Princípios Utilizados

- **Inversion of Control (IoC)**

No nosso projeto, utilizamos a anotação `@Autowired` do Spring, visto que é uma forma de Inversion of Control. Ela permite que a framework (ou container IoC) gira o ciclo de vida dos seus objetos, promovendo a separação de preocupações e facilitando a gestão de dependências.

- **Single Responsibility Principle (SRP)**



```
no usages  Nuno Domingues
@Operation(summary = "creates a product with sku generated")
@PostMapping("/createWithoutSku/")
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<ProductDTO> createWithoutSku(@RequestBody ProductWithoutSkuDTO productWithNoSku) {
    try {
        final ProductDTO product = service.createWithoutSku(productWithNoSku);
        return new ResponseEntity<ProductDTO>(product, HttpStatus.CREATED);
    }
    catch( Exception e ) {
        throw new ResponseStatusException(HttpStatus.CONFLICT, "Product must have a unique SKU.");
    }
}
```

Figura 18 - Método Product Controller

O método `createWithoutSku` é responsável por tratar o pedido HTTP e devolver a resposta HTTP. Está a aderir ao SRP ao ter uma única responsabilidade, que é a criação de um produto(`ProductDTO`).

- **Open-Closed Principle (OCP)**

O código está aberto para extensão e fechado para modificação. É possível adicionar novos algoritmos de geração de SKU (por exemplo, `SkuGenerationAlgFourImpl`) sem modificar o código existente. Isto segue o OCP, uma vez que é possível alargar o comportamento do sistema sem alterar o código existente.

Além disso, esta mesma abordagem estende-se à escolha da recomendação da Review. É possível adicionar novas implementações para a seleção de Reviews recomendadas sem afetar o código já existente, oferecendo flexibilidade na definição de critérios e lógica para recomendações.

- **Encapsulation**

A classe `SkuGenerationAlgThreeImpl` e a `ReviewRecommendation3` encapsula a lógica para combinar algoritmos de geração de SKU. Oculta os detalhes do funcionamento dos algoritmos, permitindo-lhe alterar ou alargar a lógica sem afetar o resto do código.

7. Persistência

No desenvolvimento desta solução, foram utilizadas diferentes abordagens de persistência de dados, cada uma correspondente a modelos de informação distintos, tais como os modelos relacional e de documentos, entre outros. Para implementar essas abordagens, optámos pelos seguintes sistemas de gestão de bases de dados:

- Neo4J: Utilizado para o modelo de dados em árvore.
- MongoDB: Empregue para o modelo de dados em documentos.
- H2: Utilizado para o modelo de dados relacional.
- Redis: Implementado para o modelo de mensagens.

Além disso, para configurar adequadamente cada uma dessas bases de dados, criámos ficheiros Docker Compose individuais, que especificam não apenas as imagens e portas associadas, mas também, no caso do Neo4J e do MongoDB, os detalhes de autenticação, como nomes de utilizador e palavras-passe. Esta abordagem diversificada de persistência de dados permitiu-nos adaptar a solução às necessidades específicas de cada tipo de informação e, ao mesmo tempo, garantir uma integração eficaz e segura com os respetivos sistemas de gestão de bases de dados.

```
version: '3.9'

services:
  mongodb:
    image: mongo:5.0
    ports:
      - 27017:27017
    volumes:
      - ~/apps/mongo:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=citizix
      - MONGO_INITDB_ROOT_PASSWORD=S3cret
```

Figura 19 - Exemplo docker compose

Dado que um requisito fundamental consistia na capacidade de escolher diferentes alternativas no momento de execução, com base em configurações específicas, desenvolvemos perfis individuais para cada tipo de persistência. Esses perfis contêm as informações essenciais necessárias para cada sistema, incluindo URI, nome de utilizador, password, porta, entre outros. Para cada perfil, procedemos à incorporação de um sufixo no arquivo 'application.properties' correspondente ao nome de cada base de dados. Isso simplifica ainda mais o processo de configuração e garante a clareza na associação das configurações específicas a cada sistema de gestão de bases de dados.

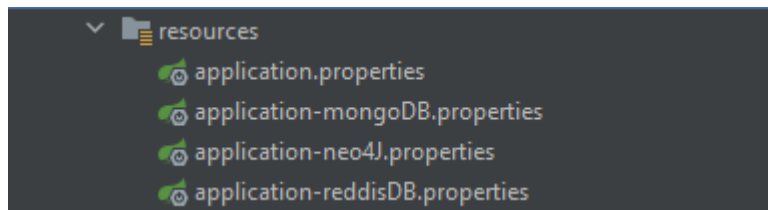


Figura 20 – application.properties associado a cada perfil

```
spring.data.mongodb.database=produtosDB
spring.data.mongodb.port=27017
spring.data.mongodb.host=localhost
spring.data.mongodb.username=produtoUser
spring.data.mongodb.password=pass
```

Figura 21 - Exemplo de application.properties

Para garantir a utilização exclusiva de um único método de persistência em cada instância, adotamos uma abordagem estruturada. Criamos uma interface para o repositório, a qual é utilizada nos serviços de acordo com o perfil selecionado. Nesse sentido, incluímos em cada perfil a definição da implementação do repositório correspondente, o que resulta numa associação direta entre o serviço desenvolvido para um perfil específico e a configuração em uso. Isso assegura uma administração eficaz e transparente da persistência de dados.

A injeção do tipo de implementação do repositório é realizada através do seguinte método:

```
@Autowired
public void setProductRepo(@Value("${product.repo}") String bean, ApplicationContext applicationContext) {
    repository = (ProductRepository) applicationContext.getBean(bean);
}
```

Figura 22 - Exemplo de injeção do repositório consoante o perfil

8. Alternativas

8.1 Geração de Sku e Recomendação de Reviews

Como referido anteriormente as implementações que se pretendem utilizar são especificadas no ficheiro application.properties. Em alternativa a esta solução poderiam ser utilizados perfis de Spring, que permite utilizar diferentes beans e configurações para diferentes cenários. É possível definir um perfil em tempo de execução

```
@Configuration
@Profile("review1")
public class Review1Config {
    new *
    @Bean
    public ReviewRecommendation reviewRecommendation() {
        return new ReviewRecommendation1();
    }
}

new *
@Configuration
@Profile("review2")
public class Review2Config {
    new *
    @Bean
    public ReviewRecommendation reviewRecommendation() {
        return new ReviewRecommendation2();
    }
}
```

Figura 23

No entanto optou-se por utilizar o @Value visto que:

- Permite seleccionar a implementação ao nível de um bean, enquanto os perfis são normalmente utilizados para agrupar configurações num scope mais amplo. Permitindo assim definir, utilizando o mesmo perfil, implementações diferentes para a geração do sku e recomendação de reviews sem ter de criar múltiplos perfis.
- Com a utilização de perfis era necessário definir o perfil ativo sempre que a aplicação é executada.

No geral permite uma solução com mais granularidade e simplicidade de configuração.

8.2 Diferentes modelos de dados

Para esta implementação era possível utilizar perfis ou a mesma solução utilizada nos outros dois casos anteriores. Ao utilizar apenas perfis impedia a especificação individual para cada domain, e ao utilizar apenas valores no ficheiro de propriedades, dificultava a especificação de um grupo de propriedades necessárias para o uso de um modelo de dados específicos.

Como referido anteriormente a configuração das implementações foi feita a nível do serviço, fazendo diferentes implementações para os repositórios. Uma outra alternativa seria fazer a configuração a nível do controller, fazendo diferentes implementações para os serviços, repositórios e modelo. Esta implementação trazia algumas vantagens como a separação de responsabilidades e agrupamento de lógica, no entanto iria trazer várias desvantagens como:

- Aumento de código duplicado, iria ser necessário reescrever todos os serviços para os diferentes modelos de dados
- A configuração seria mais complexa tendo de especificar todos os serviços a ser utilizados
- Replicação tanto de serviços como repositórios
- Seria necessário utilizar várias classes de modelo

9. Attribute-Driven Design

Attribute-Driven Design (ADD) é uma abordagem de design de software que se concentra na identificação e priorização dos atributos de qualidade do sistema. Esses atributos de qualidade podem incluir coisas como desempenho, segurança, escalabilidade, usabilidade e confiabilidade.

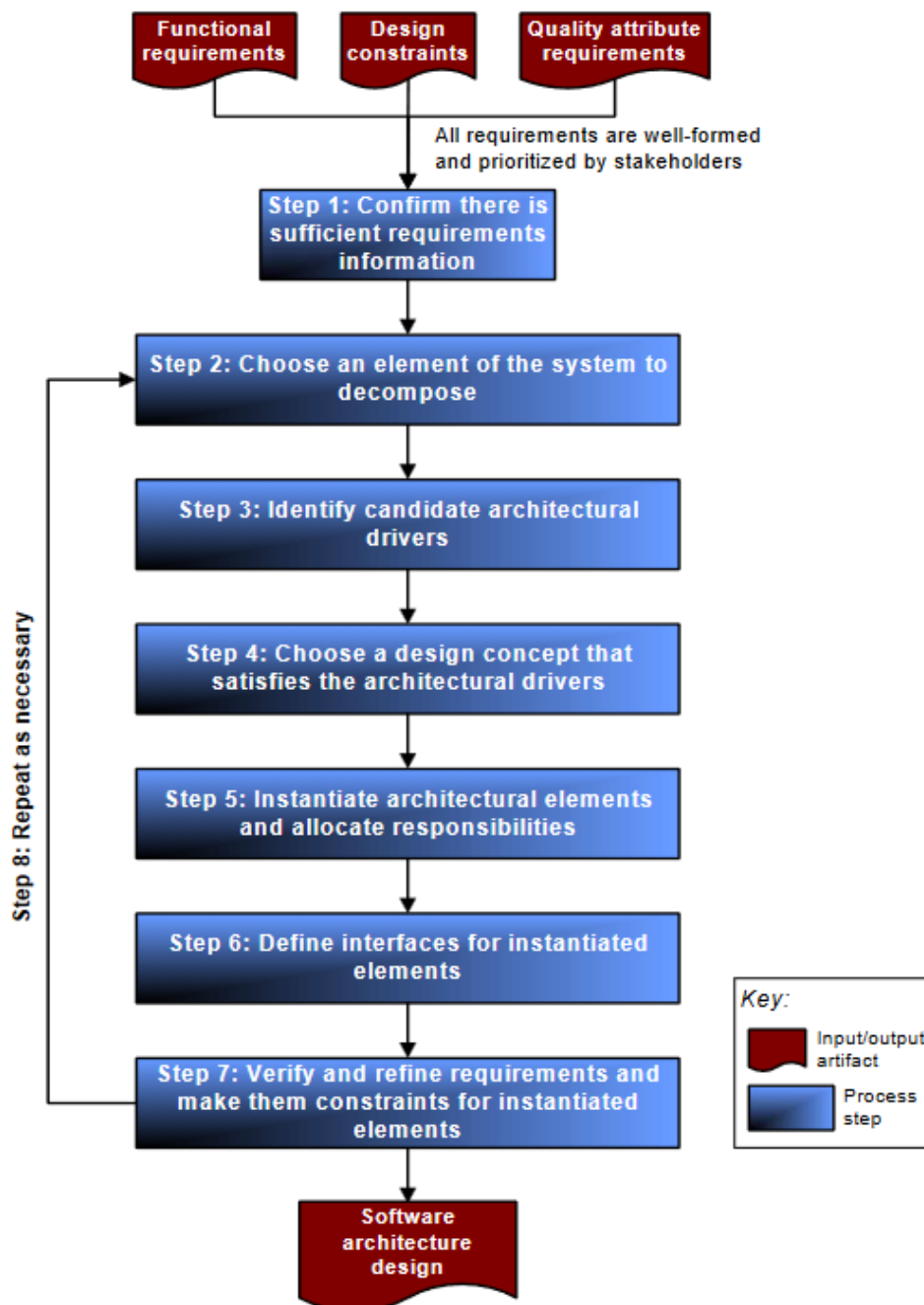


Figura 24 – ADD Diagrama

9.1 Iteração 1

9.1.1 ADD Step 1 – Review inputs

Na primeira iteração do processo de ADD (Attribute-Driven Design), começaremos por identificar todos os architectural drivers. Estes architectural drivers desempenham um papel fundamental na definição do sistema de software. Ao longo de cada iteração do processo de

desenvolvimento, estes elementos serão sujeitos a uma revisão contínua e avaliação para determinar se são necessárias quaisquer modificações. O objetivo desta revisão contínua é assegurar que a arquitetura de software se mantenha alinhada com os objetivos e requisitos do projeto.

Os architectural drivers servem como os princípios orientadores que definem o "o quê" e o "porquê" dos nossos esforços de desenvolvimento de software. Eles fornecem a base para descrever a essência do trabalho e a justificação subjacente que o fundamenta. Estes drivers incluem:

- Design Purpose
- Primary Functional Requirements (Use Cases)
- Quality Attributes Scenarios
- Architectural Concerns
- Constraints

Design Purpose

Aprimorar a experiência do utilizador e a eficiência operacional, integrando funcionalidades de recomendação de avaliações, geração de SKUs para produtos e implementação de novos modelos de persistência de dados.

Primary Functional Requirements (Use Cases)

Após a análise dos requisitos do projeto, é o momento de descrever os casos de uso identificados. Na tabela seguinte, encontram-se os casos de uso identificados:

Caso de usos	Description
UC1: Geração de SKU em Diferentes Formatos	Este caso de uso envolve a geração de códigos de SKU em diferentes formatos, de acordo com especificações variáveis.
UC2: Persistência de Dados em Modelos de Dados Diferentes	Este caso de uso aborda a capacidade de persistir dados em vários modelos de dados, como modelos relacionais ou de documentos, utilizando diferentes Sistemas de Gestão de Bases de Dados (SGBD).
UC3: Recomendação de Reviews	Este caso de uso envolve a recomendação de avaliações de produtos ou serviços com base em critérios

Quality Attributes Scenarios

Na tabela abaixo, são apresentados os atributos de qualidade definidos com base em cenários e os casos de uso associados. Além disso, são fornecidos os níveis de importância e dificuldade:

Quality Attribute	Scenario	Use Case	Importance Level	Difficulty Level
QA-1 Modifiability	O sistema deve permitir optar por diferentes implementações através das configurações em run-time.	ALL	HIGH	HIGH
QA-2 Maintainability	O sistema deve ter um baixo acoplamento entre os componentes e ser de fácil atualização.	ALL	HIGH	HIGH

Architectural Concerns

Um architectural concern é um elemento que deve ser levado em conta no processo de concepção arquitetural. Geralmente, corresponde a uma necessidade ou exigência de um stakeholder e, frequentemente, leva à criação de um novo cenário de atributos de qualidade.

Architectural Concern	Description
CRN-1	Estabelecimento de uma nova estrutura geral do sistema.
CRN-2	Alocar tarefas aos membros da equipa de desenvolvimento.

Constraints

Restrições arquiteturais são limitações no processo de desenvolvimento que afetam o processo de design arquitetural. Essas limitações podem ser de natureza técnica ou relacionadas a negócios.

Architectural Constraint	Description
CON-1	As tecnologias Spring Boot e Java são obrigatórias.
CON-2	A abordagem DDD é obrigatória.
CON-3	O sistema deve permitir escolher o tipo de persistência através de configurações.
CON-4	O sistema deve permitir escolher a implementação da recomendação através de configurações.
CON-5	O sistema deve permitir escolher a implementação da geração do sku através de configurações.
CON-6	Utilização do princípio Interface Segregation

9.1.2 ADD Step 2 – Establish the Iteration Goal

O objetivo desta iteração é definir a estrutura dos novos use cases que vão ser implementados no sistema.

A arquitetura de referência e os padrões de implementação são os conceitos de concepção escolhidos.

9.1.3 ADD Step 3 – Choose What to Refine

Este passo serve para definir o que vai ser necessário refinar durante a interação.

Devido ao facto disto ser a primeira interação, não existem elementos para refinar.

9.1.4 ADD Step 5 – Instantiate Architectural Elements, Allocate Responsibilities and Define Interfaces

A etapa 5 apresenta as vistas que instanciam os elementos arquiteturais decididos na etapa 4. Na figura seguinte é apresentado um diagrama que corresponde às decisões tomadas e ilustram as vistas de software.

Utilizando a Vista Lógica, a figura seguinte ilustra a estrutura global do sistema.

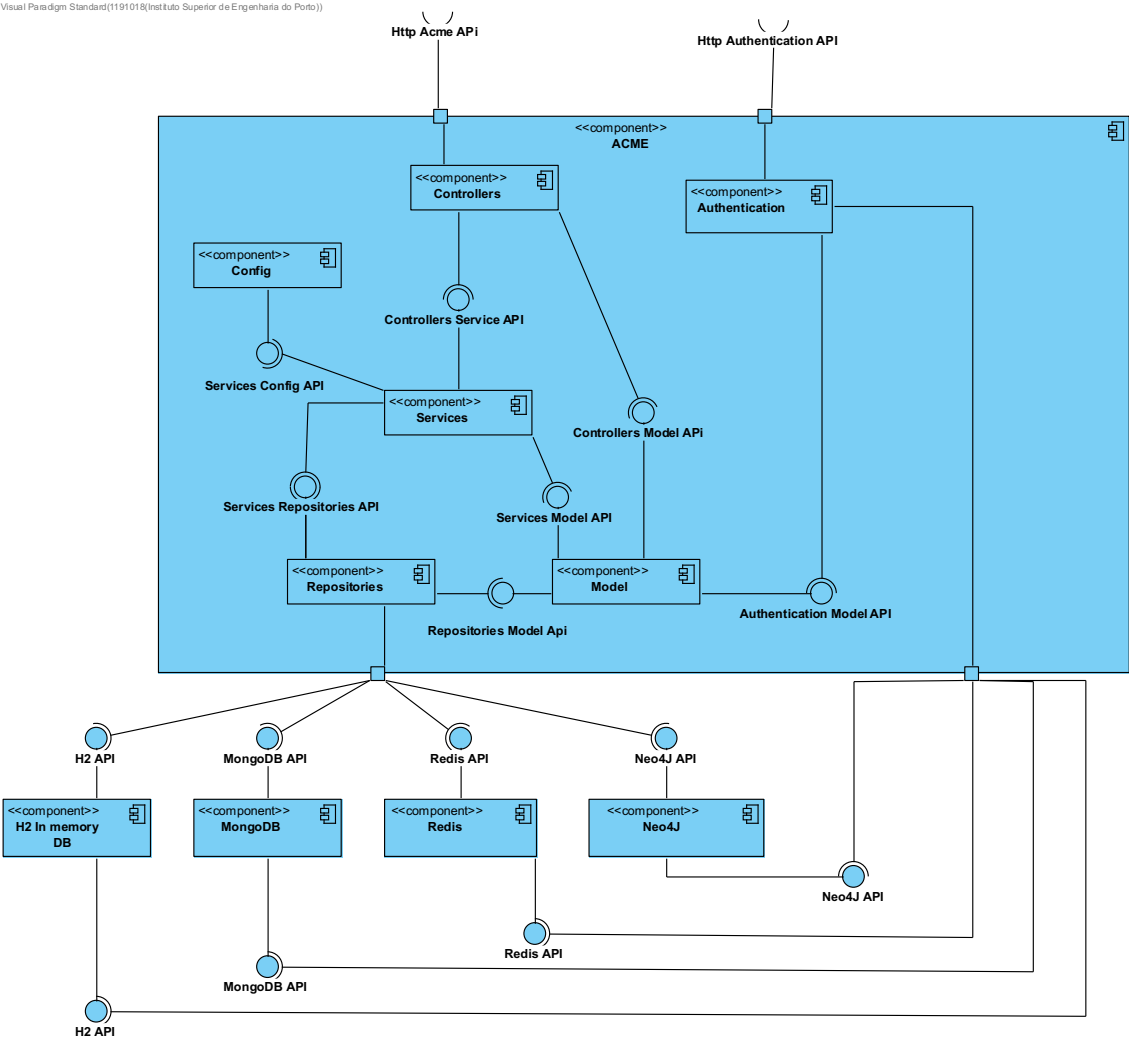


Figura 251 – Vista Lógica Nível 2

9.1.5 ADD Step 7 – Analyse Current Design, and Review Iteration Goal Achievement of Design Purpose

Através da tabela abaixo, é possível compreender como os Architectural Drivers foram elaborados durante a primeira iteração do ADD.

Não Elaborado	Elaborado
UC1	CRN-1
UC2	CRN-2
UC3	
QA-1	
QA-2	
CON-1	
CON-2	
CON-3	
CON-4	
CON-5	
CON-6	

9.2 Iteração 2

9.2.1 ADD Step 1- Review Inputs

A equipa analisou e decidiu que não era necessário efetuar quaisquer alterações.

9.2.2 ADD Step 2- Establish the Iteration Goal

O objetivo desta segunda interação é elaborar as funcionalidades analisadas na primeira interação.

9.2.3 ADD Step 3- Choose what to refine

- UC1: Geração de SKU em Diferentes Formatos
- UC2: Persistência de Dados em Modelos de Dados Diferentes
- UC3: Recomendação de Reviews

9.2.4 ADD Step 4- Choose design Concepts that satisfy the selected drivers

- **Configuração em Tempo de Execução**

A configuração em tempo de execução permite adaptar dinamicamente o comportamento do sistema com base nas necessidades específicas dos nossos drivers. Isto significa que poderemos ajustar parâmetros e configurações em tempo real, otimizando a experiência do utilizador e a eficiência operacional.

- **Uso de interfaces para diferentes implementações**

Permite definir contratos claros que várias implementações devem seguir. Isto permite flexibilidade na escolha das implementações com base nas necessidades específicas do sistema e dos drivers de design. A capacidade de configurar diferentes implementações em tempo de execução oferece versatilidade e adaptabilidade ao sistema.

- **Integração de Diferentes Modelos de Dados**

Considerando que os dados são essenciais numa aplicação de e-commerce, a integração de diferentes modelos de dados deve ser cuidadosamente planeada. Isto envolve a harmonização de estruturas de dados diversas, como bancos de dados relacionais e NoSQL, para garantir que as informações relevantes estejam disponíveis de maneira eficiente.

Foram selecionadas as seguintes tecnologias para os seguintes modelos:

- Relacional: H2
- Document Model: MongoDB
- Tree Model : Neo4J
- Message Model : Redis

9.2.5 ADD Step 5- Instantiate Architectural Elements, Allocate Responsibilities, and Define Interfaces

Para a geração do Sku deve ser possível escolher por ficheiro de configuração as diferentes implementações de algoritmo de geração de sku:

- {Algarismo, letra, algarismo, letra, algarismo, letra, algarismo, letra, algarismo, carater especial}
- 10 carateres da representação hexadecimal to hashcode da designação do produto
- Composição dos outros dois algoritmos, 6 carateres do primeiros e 5 do segundo

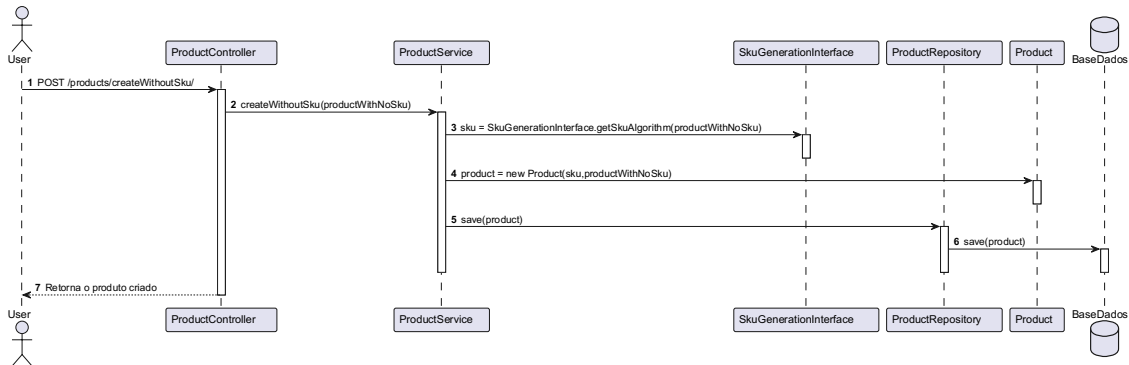


Figura 26- Diagrama Sequência POST(criar produto sem sku)

O algoritmo de recomendação de reviews, deve seguir uma abordagem semelhante e deve ser possível escolher em tempo de execução qual implementação de recomendação irá ser utilizada:

- Listar todas as reviews por quantidade de votos, em que o mínimo de votos positivos deve ser 4 e 65% devem ser votos positivos.
- As reviews de utilizadores que votam 50% como o utilizador. Se 2 ou mais votarem assim, as avaliações deles são sugeridas.
- Combinação dos 2 algoritmos anteriores, mas apenas as reviews que foram escritas por utilizadores que têm reviews semelhantes às do utilizador.

Deve ser possível armazenar a informação em diferentes modelos de dados:

- Relacional
- Document Model
- Tree Model
- Message Model

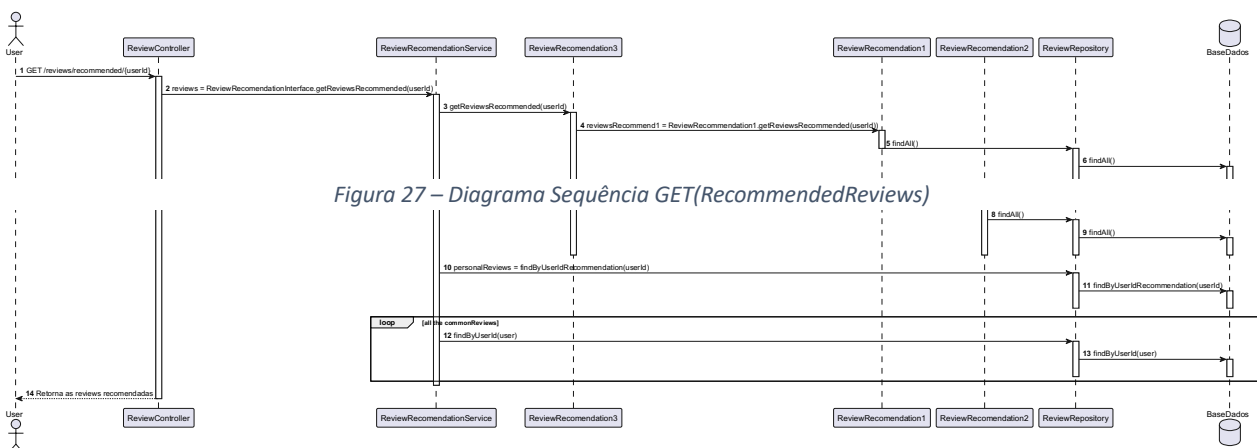


Figura 27 – Diagrama Sequência GET(RecommendedReviews)

Deve também ser possível seleccionar o diferente modelo de dados em tempo de execução.

Testes

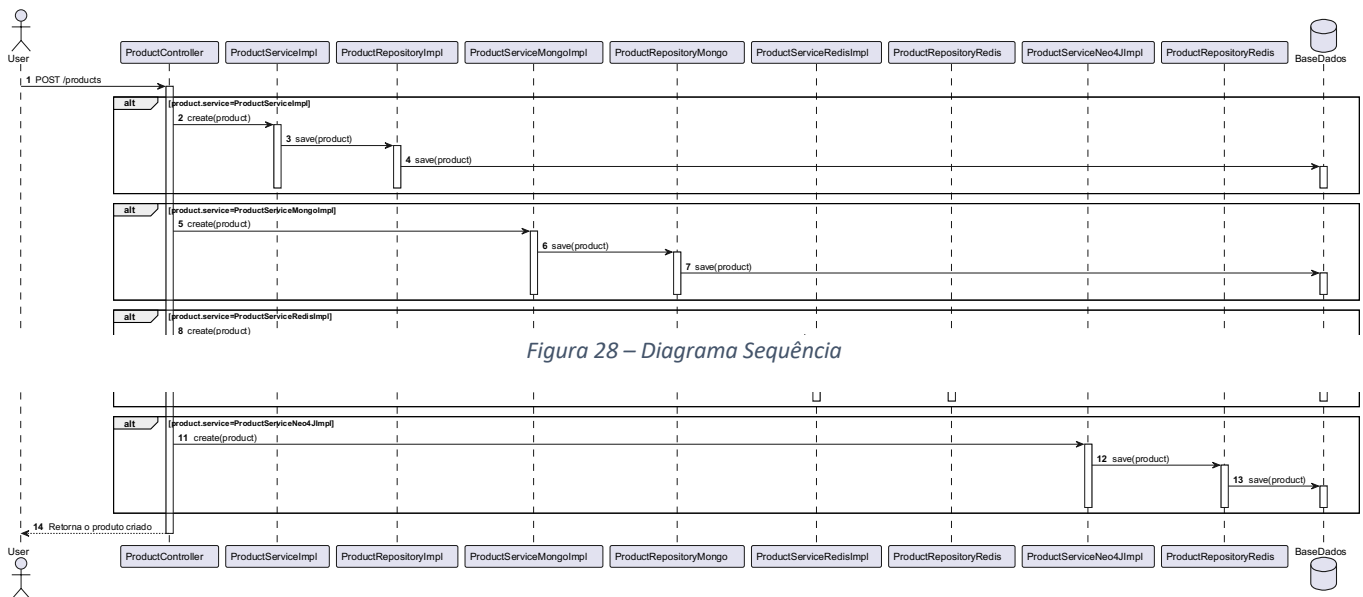


Figura 28 – Diagrama Sequência

Design Decisions	Rational
Testes unitários	Previne ocorrência de erros. Quando os testes unitários são implementados corretamente permitem identificar falhas no processo inicial do código, que podem ser mais difíceis de identificar numa fase de testes mais avançadas.
Testes de integração	Ao integrar diferentes módulos, por vezes desenvolvidos por diferentes desenvolvedores, pode originar problemas. Estes testes permitem verificar que os diferentes módulos funcionam corretamente como um todo.

Testes unitários (exemplo)

```
@Test
void testGetProductBySku() {
    String sku = "SKU123";
    ProductMongo productMongo = new ProductMongo(sku, designation: "Test Product", description: "Description");
    when(repository.findBySku(sku)).thenReturn(Optional.of(productMongo));

    Optional<Product> result = productServiceMongo.getProductBySku(sku);
    assertTrue(result.isPresent());
}
```

Figura 29 – Teste Get Product By Sku

Testes de integração (exemplo)

```

"request": {
  "method": "POST",
  "header": [],
  "body": {
    "mode": "raw",
    "raw": "{\n  \"sku\": \"{{sku}}\", \n  \"designation\": \"{{designation}}\", \n  \"description\": \"{{description}}\"\n}",
    "options": {
      "raw": {
        "language": "json"
      }
    }
  },
  "url": {
    "raw": "{{HOST}}/products",
    "host": [
      "{{HOST}}"
    ],
    "path": [
      "products"
    ]
  }
},
"response": []
},

```

Figura 30 – Teste Integração Parte 1

```

1. {
  "listen": "test",
  "script": {
    "exec": [
      "pm.test(\"Product created\", function () {",
      "  pm.response.to.have.status(201);",
      "});",
      "",
      "pm.test(\"A new product was returned\", function () {",
      "  ",
      "  const product = pm.response.json();",
      "  ",
      "  const sku = pm.environment.get(\"sku\");",
      "  const designation = pm.environment.get(\"designation\");",
      "  ",
      "  pm.expect(product.sku).to.equal(sku);",
      "  pm.expect(product.designation).to.equal(designation);",
      "});",
      "",
      ""
    ],
    "type": "text/javascript"
  }
}

```

Figura 31 – Teste Integração Parte 2

9.2.6

Step 5 -

Current Design, and Review Iteration Goal + Achievement of Design Purpose

ADD
Analyze

Não Elaborado	Elaborado
	CRN-1
	CRN-2
	UC1
	UC2
	UC3
	QA-1
	QA-2

	CON-1
	CON-2
	CON-3
	CON-4
	CON-5