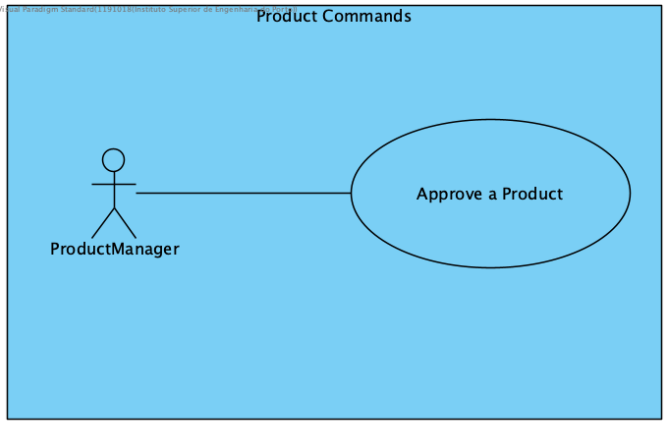
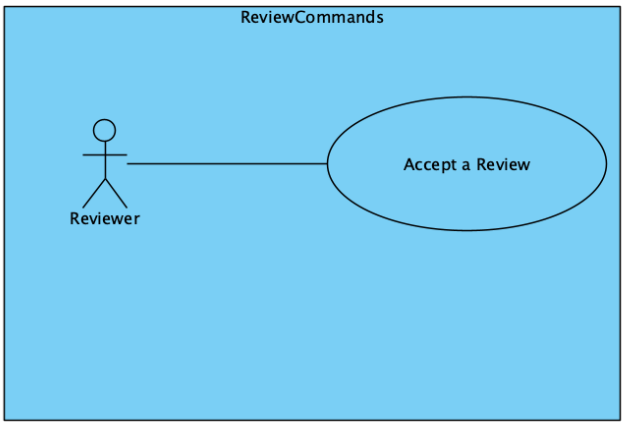
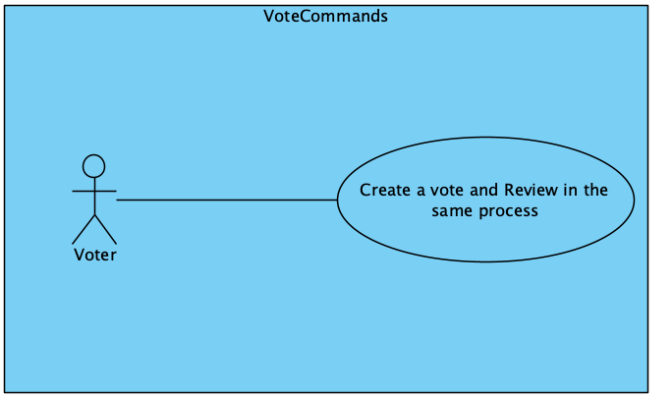
1.Use Cases:


Product Commands
Approve a Product
ProductManager

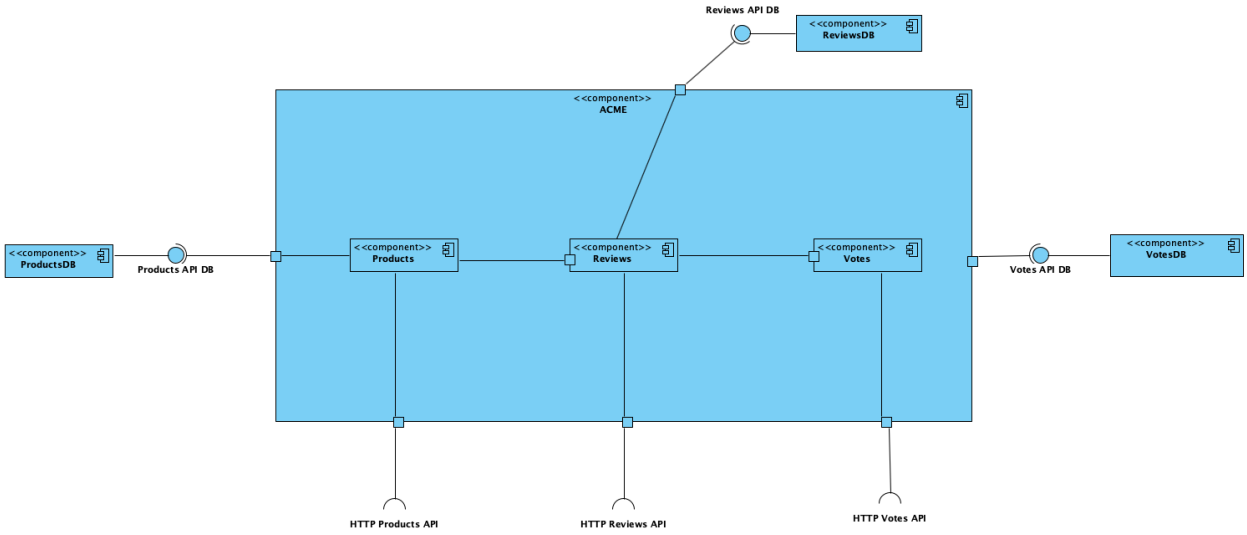

ReviewCommands
Accept a Review
Reviewer


VoteCommands
Create a vote and Review in the same process
Voter

2.Vista lógica

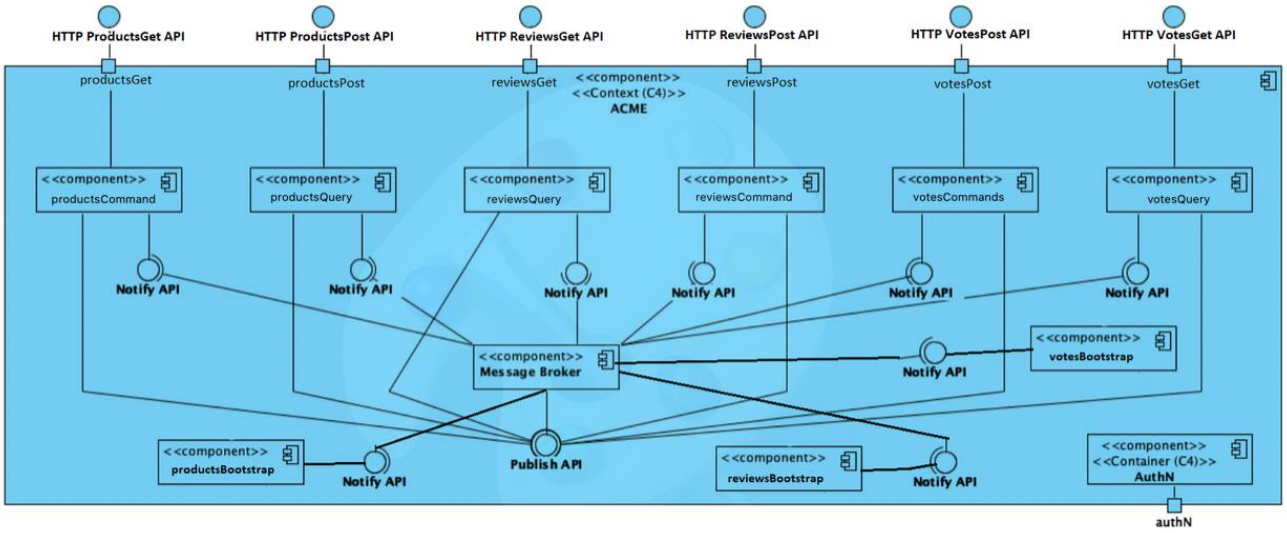
Nivel 2

Neste diagrama estão demonstrados os principais constituintes da aplicação. As API de HTTP que permitem fazer pedidos de HTTP aos constituintes do ACME. Estão presentes os diferentes componentes dos diferentes domínios que consomem dados da API HTTP e das bases de dados. Nos componentes de base de dados (DB) é armazenada toda a informação.

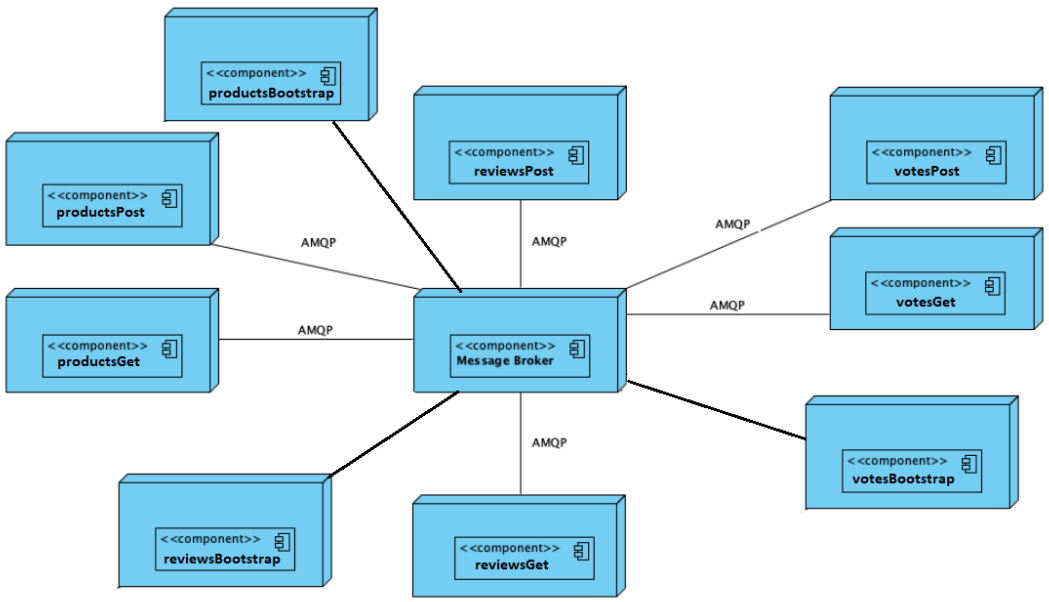


Nivel 3

Neste diagrama de nível 3, está descrito mais detalhadamente o serviço. Foi aplicado o padrão Command and Query Responsibility Segregation (CQRS) que separa as operações de leitura e atualização para uma base de dados e também foi aplicado o armazenamento de domain events. Para tal foram criados microserviços para os Commands, Queries e Events.



3.Vista física

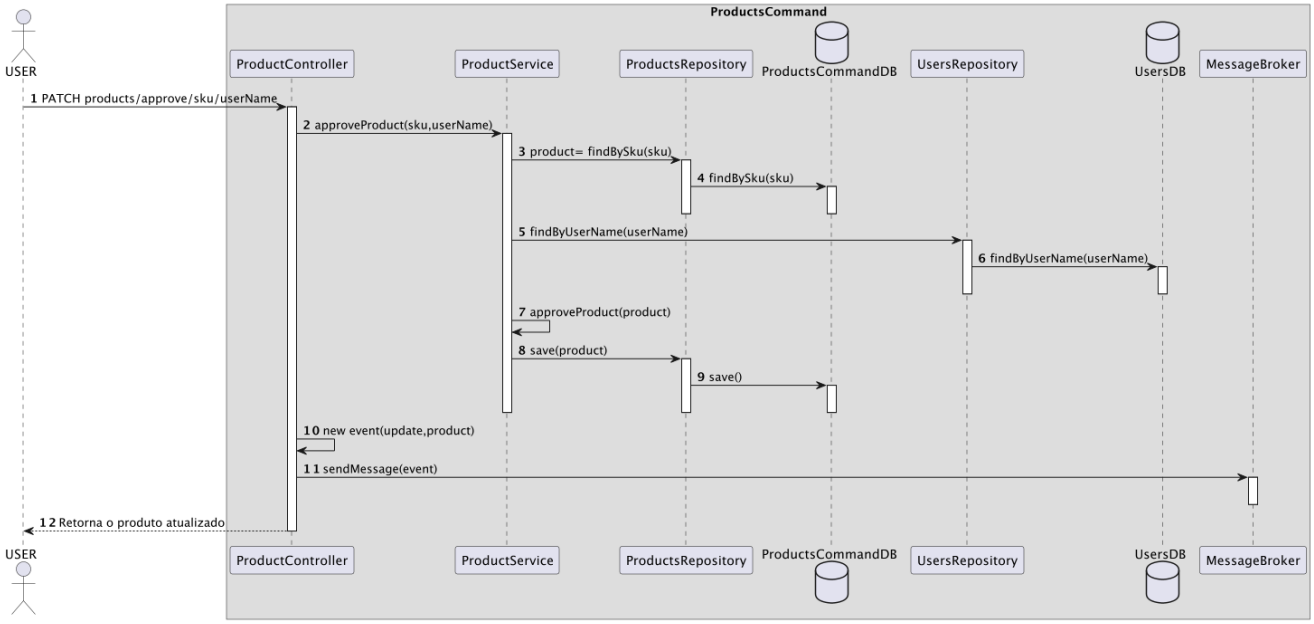


4. Diagramas de sequência

4.1 Use Cases

1. Aprovar Produto

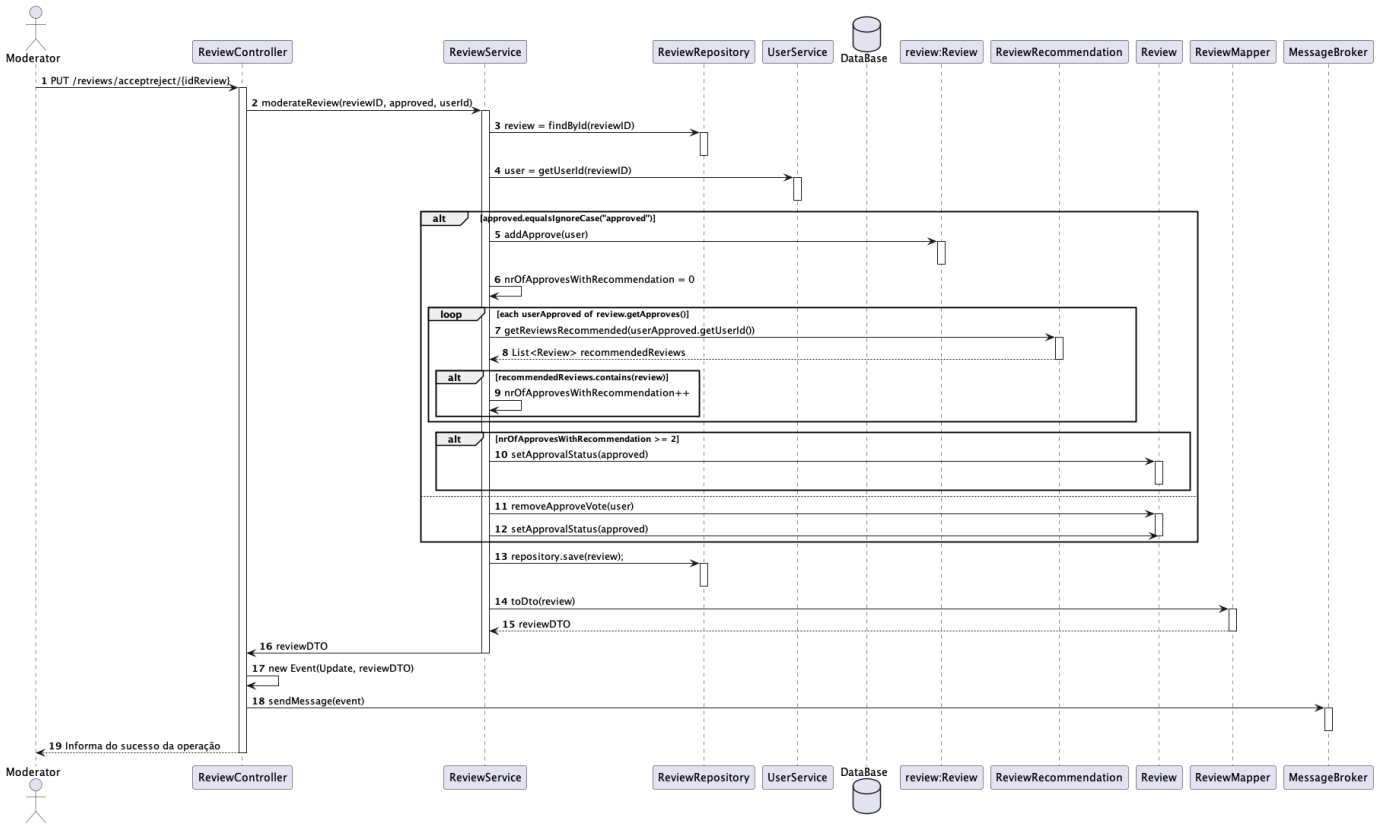
Como product manager posso aprovar um produto. Para tal deve fazer um pedido HTTP PATCH para o microserviço ProductsCommand passando o sku do produto e o seu username.



Se o produto for aprovado por dois product managers, este vai ser publicado.

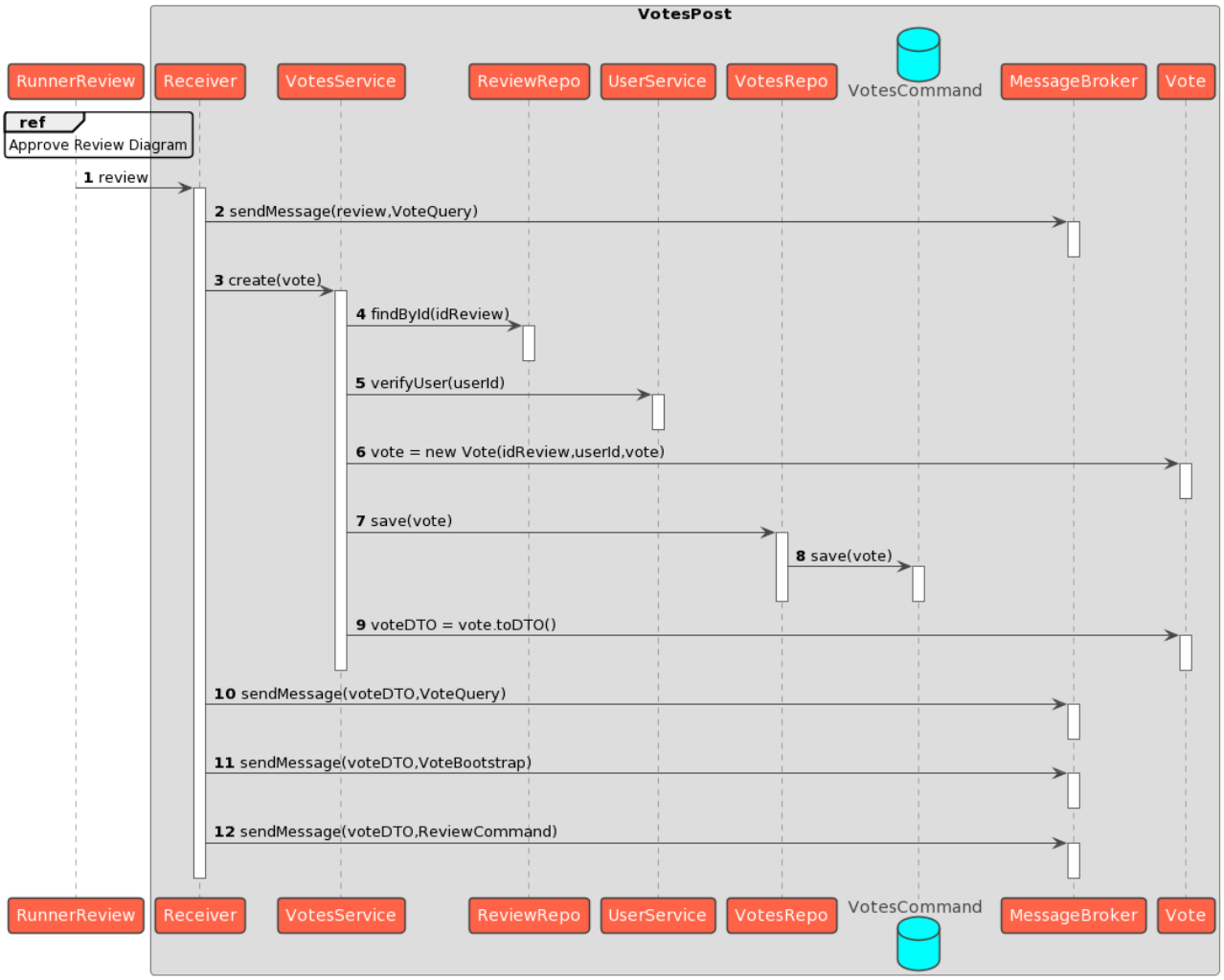
2. Aprovar review

Como reviewer quero que a minha review seja aprovada. Para tal, é necessário que dois moderadores aprovem e sejam recomendados por essa mesma review. Para realizar a aprovação, deve fazer um pedido HTTP PUT para o microserviço ReviewCommand passando o id da Review, o Utilizador e indicar se quer aprovar, rejeitar ou colocar em pendente.



3. Criar Vote

Após a aprovação de uma Review, é criado um voto no mesmo processo. Deste modo, o VoteCommand recebe a review criada e cria um voto com o eleitor e o id da review recebida.



Princípios Utilizados

Strangler Fig

O padrão de software "Strangler Fig" é uma estratégia de modernização de sistemas legados. Ele foi introduzido por Martin Fowler como uma abordagem para migrar gradualmente um sistema monolítico para uma arquitetura mais moderna e modular, sem interromper os serviços existentes.

A ideia por trás do "Strangler Fig" é semelhante à planta na natureza que envolve gradualmente outra árvore. No contexto de software, isto implica desenvolver novas funcionalidades ou serviços independentes do sistema legado existente. Estes novos componentes são integrados no sistema legado, coexistindo e interagindo com os serviços antigos.

À medida que mais funcionalidades são migradas e implementadas nos novos componentes, os serviços antigos podem ser desativados ou substituídos. Este processo continua até que o sistema legado seja completamente substituído pelos novos componentes, sem interrupções significativas.

Principais benefícios do padrão "Strangler Fig":

1. **Migração Gradual:** Permite a migração passo a passo, reduzindo riscos e interrupções.
2. **Entrega Contínua:** As equipas podem continuar a entregar valor ao utilizador final durante o processo de modernização.
3. **Mitigação de Riscos:** A abordagem gradual permite identificar e corrigir problemas à medida que surgem.
4. **Adoção de Tecnologias Modernas:** Facilita a incorporação de novas tecnologias sem uma reescrita completa do sistema.

O padrão "Strangler Fig" é especialmente útil quando lidamos com sistemas legados grandes e complexos, onde a substituição completa pode ser arriscada ou inviável. Esta abordagem proporciona uma transição suave para arquiteturas mais modernas e flexíveis.

CQRS

O Command-Query Responsibility Segregation (CQRS) é um padrão de arquitetura de software que separa as operações de leitura (queries) das operações de escrita (commands) num sistema. Este padrão foi introduzido para lidar com a complexidade associada às operações de leitura e escrita em sistemas distribuídos e escaláveis.

A ideia central por trás do CQRS é reconhecer que as operações de leitura e escrita têm requisitos diferentes em termos de modelagem de dados e otimização de desempenho. Portanto, ao separar as responsabilidades para essas operações, é possível otimizar cada uma delas independentemente.

Principais conceitos do CQRS:

1. **Commands (Comandos):**
    - Representam operações que alteram o estado do sistema.
    - Exemplos incluem criar, atualizar ou excluir dados.
    - Os comandos são manipulados por manipuladores de comandos, que são responsáveis por atualizar o estado do sistema.
2. **Queries (Consultas):**
    - Representam operações que retornam dados sem alterar o estado do sistema.
    - Exemplos incluem operações de leitura de dados.
    - As queries são manipuladas por manipuladores de queries, que são responsáveis por recuperar dados para apresentação.
3. **Modelos de Leitura e Escrita Separados:**

- O CQRS sugere que os modelos de dados para leitura e escrita sejam separados.
- Isto permite otimizar cada modelo para as suas operações específicas, facilitando o ajuste fino de desempenho.

4. **Event Sourcing (Origem de Eventos):**
   - Uma prática comum no contexto do CQRS é o uso do Event Sourcing.
   - Em vez de armazenar apenas o estado atual dos dados, o sistema mantém um registro de eventos que levaram a esse estado.
   - Os eventos são usados para reconstruir o estado do sistema a qualquer momento e também podem ser usados para auditoria e análise.
5. **Escalabilidade:**
   - A separação de comandos e queries permite a escalabilidade independente de cada parte do sistema.
   - Sistemas que enfrentam uma carga intensiva de leitura podem ser otimizados para consultas, enquanto sistemas que recebem muitos comandos podem ser otimizados para gravação.

O CQRS não é uma solução para todos os cenários e introduz complexidade adicional no design de sistemas. Deve ser considerado em casos onde a simplificação do modelo de dados e a otimização de desempenho são prioridades, especialmente em sistemas complexos ou distribuídos. A adoção do CQRS pode proporcionar maior flexibilidade e desempenho, mas também requer uma compreensão sólida dos requisitos do sistema e das implicações de design associadas.

Database-per-Service

O conceito de "Database-per-Service" é uma abordagem arquitetônica em que cada serviço numa arquitetura de microservices tem a sua própria base de dados dedicada. Essa estratégia é uma variação da arquitetura de microservices, na qual a independência e a autonomia dos serviços são levadas a um nível mais granular, estendendo-se às bases de dados associados a cada serviço específico.

Principais características do "Database-per-Service":

1. **Separação de Dados:**
   - Cada serviço possui a sua própria base de dados, o que implica que os dados relacionados a um serviço específico são mantidos separadamente dos dados de outros serviços.
   - Essa separação promove a independência entre os serviços, evitando acoplamento de dados.
2. **Autonomia de Desenvolvimento:**
   - A equipa de desenvolvimento responsável por um serviço tem controlo total sobre a sua base de dados.

    - Isso permite que as equipas escolham o tipo de base de dados que melhor atende às necessidades do serviço sem afetar outros serviços.
3. **Escalabilidade Independente:**
    - Cada base de dados pode ser escalada independentemente com base nos requisitos específicos de carga e desempenho do serviço associado.
    - Isso facilita a adaptação da infraestrutura para atender às necessidades variáveis de cada serviço.
4. **Manutenção e Evolução Facilitadas:**
    - Atualizações de esquema, migrações de dados e manutenção geral da base de dados podem ser realizadas de forma isolada para cada serviço, minimizando o impacto em outros componentes do sistema.
5. **Resiliência e Tolerância a Falhas:**
    - A falha numa base de dados de um serviço não afeta diretamente os outros serviços, proporcionando maior resiliência e isolamento de falhas.
6. **Consistência e Coerência:**
    - Embora a separação de dados seja uma vantagem, também pode requerer a implementação cuidadosa de mecanismos para garantir a consistência e a coerência entre os dados distribuídos nos diferentes bancos de dados.
7. **Custo e Overhead:**
    - Apesar das vantagens, ter uma base de dados por serviço pode aumentar os custos operacionais e a complexidade da administração de várias instâncias de base de dados.

É importante notar que a escolha de adotar uma abordagem de "Database-per-Service" deve ser cuidadosamente considerada com base nos requisitos específicos do sistema. Em alguns casos, a complexidade associada a ter várias bases de dados pode não ser justificada, e uma abordagem diferente, como compartilhar bases de dados entre serviços, pode ser mais apropriada. Cada estratégia tem seus prós e contras, e a decisão deve ser alinhada com os objetivos e as necessidades específicas do projeto.

Polyglot Persistence

Polyglot Persistence é uma abordagem em arquitetura de software que envolve o uso de diferentes tecnologias de armazenamento de dados para atender a requisitos específicos num sistema. Cada componente pode utilizar um tipo de base de dados que melhor se adequa às suas necessidades, proporcionando flexibilidade, adaptação aos requisitos do domínio e evolução gradual do sistema. Isto pode incluir o uso de bases de dados relacionais para dados transacionais e NoSQL para dados não estruturados, por exemplo. Apesar dos benefícios, é importante gerir desafios como consistência e integração ao adotar o polyglot persistence.

Messaging

"Messaging" em arquitetura de software refere-se à prática de usar sistemas de mensagens para facilitar a comunicação e a troca de informações entre diferentes partes de um sistema distribuído. Esta abordagem é fundamental em arquiteturas orientadas a eventos

e em sistemas que procuram desacoplar componentes para melhorar a escalabilidade, a flexibilidade e a confiabilidade.

Principais conceitos relacionados a "Messaging":

1. **Comunicação Assíncrona:**
    - O "Messaging" permite a comunicação assíncrona entre componentes de um sistema.
    - Em vez de componentes interagirem diretamente, eles enviam mensagens para filas ou tópicos, permitindo que outros componentes processem essas mensagens quando estiverem prontos.
2. **Filas de Mensagens:**
    - As filas de mensagens são estruturas de armazenamento temporário para mensagens.
    - Um componente envia uma mensagem para uma fila, e outro componente consome essa mensagem quando estiver pronto para processá-la.
3. **Tópicos de Mensagens:**
    - Os tópicos de mensagens permitem a publicação e a assinatura de mensagens por vários consumidores.
    - Uma mensagem enviada para um tópico é entregue a todos os consumidores interessados.
4. **Desacoplamento:**
    - O uso de mensagens permite o desacoplamento entre os diversos componentes de um sistema.
    - Componentes podem evoluir independentemente sem dependerem diretamente uns dos outros.
5. **Escalabilidade:**
    - Sistemas de mensagens facilitam a construção de sistemas escaláveis, pois permitem que diferentes partes do sistema evoluam e escalem independentemente.
6. **Persistência:**
    - Muitos sistemas de mensagens oferecem suporte à persistência, garantindo que as mensagens não sejam perdidas mesmo em caso de falha de algum componente.
7. **Padrões de Mensagens:**
    - A definição clara de padrões de mensagens facilita a interoperabilidade entre diferentes componentes e sistemas.
    - Padrões como JSON, XML ou protocolos específicos podem ser usados para a serialização das mensagens.
8. **Broker de Mensagens:**
    - Muitas implementações de sistemas de mensagens utilizam um "message broker" (corretor de mensagens) para facilitar a troca de mensagens entre os diferentes componentes.

"Messaging" é amplamente utilizado em arquiteturas de microservices, sistemas distribuídos e em ambientes onde a escalabilidade, a resiliência e o desacoplamento são prioridades. Essa abordagem ajuda a criar sistemas mais flexíveis, robustos e capazes de lidar com mudanças e expansões de forma eficaz.

Domain Events

"Domain Events" são eventos significativos que ocorrem no domínio de uma aplicação, representando mudanças importantes no estado. Estes servem para notificar partes do sistema sobre essas mudanças, possibilitando a comunicação, desacoplamento entre componentes e integração eficiente em sistemas distribuídos. Exemplos incluem a criação de uma entidade ou a conclusão de uma transação. Esses eventos são essenciais para modelar processos de negócios, registar histórico de alterações e facilitar a evolução flexível e independente do sistema.

Saga

O padrão Saga fornece gestão de transações com uma sequência de *transações locais*. Uma transação local é o esforço de trabalho atómico realizado por um participante da saga. Cada transação local atualiza a base de dados e publica uma mensagem ou evento para acionar a próxima transação local na saga. Se uma transação local falhar, a saga executa uma série de *transações compensatórias* que anulam as alterações efetuadas pelas transações locais anteriores.

Nos padrões da Saga:

- *As transações compensatórias* são transações que podem potencialmente ser invertidas ao processar outra transação com o efeito oposto.
- Uma *transação dinâmica* é o ponto go/no-go numa saga. Se a transação dinâmica for consolidada, a saga será executada até à conclusão. Uma transação dinâmica pode ser uma transação que não é compensavel nem replicável, ou pode ser a última transação compensatória ou a primeira transação replicável na saga.
- *As transações retráveis* são transações que seguem a transação dinâmica e têm a garantia de sucesso.

Containerização e Docker

A containerização é uma tecnologia que permite empacotar e executar aplicações juntamente com todas as suas dependências e configurações num ambiente isolado chamado container. Cada container é uma unidade leve e independente que pode ser executada consistentemente em diferentes ambientes, desde o desenvolvimento local até a produção em larga escala. O Docker é uma das plataformas de containerização mais populares.

Principais Conceitos:

1. **Container:**
    - Um container é uma unidade padronizada de software que inclui o código da aplicação, as suas bibliotecas e dependências, bem como configurações específicas do ambiente. Ele é isolado do sistema operacional hospedeiro e de outros containers.
2. **Docker:**
    - O Docker é uma plataforma de código aberto que facilita a criação, distribuição e execução de containers. Ele fornece ferramentas e uma API consistente para gerenciar containers.
3. **Imagem:**
    - Uma imagem Docker é um pacote que contém todas as informações necessárias para criar um container. Ela inclui o sistema operacional, as bibliotecas, o código da aplicação e as configurações.
4. **Dockerfile:**
    - Um Dockerfile é um arquivo de configuração usado para definir as instruções necessárias para construir uma imagem Docker. Ele especifica a base da imagem, as dependências e os comandos para configurar o ambiente.
5. **Orquestração de Container:**
    - Ferramentas como Docker Compose e Kubernetes são usadas para orquestrar a implantação, escala e gerir containers em ambientes de produção.

Vantagens da Containerização:

1. **Isolamento:**
    - Os containers fornecem isolamento eficiente, garantindo que cada aplicativo e suas dependências funcionem de maneira independente e não interfiram uns com os outros.
2. **Portabilidade:**
    - Os containers são portáteis, o que significa que podem ser executados de maneira consistente em diferentes ambientes, desde o desenvolvimento até a produção.
3. **Escalabilidade:**
    - A containerização facilita a escalabilidade horizontal, permitindo a rápida replicação e implantação de containers conforme necessário.
4. **Eficiência de Recursos:**
    - Os containers compartilham o kernel do sistema operacional hospedeiro, o que os torna mais leves e eficientes em termos de recursos em comparação com máquinas virtuais.
5. **Entrega Contínua:**
    - A containerização é fundamental para práticas modernas de entrega contínua, permitindo a rápida implantação e atualização de aplicativos.

CICD

CI/CD é a abreviação de *Continuous Integration/Continuous Delivery,* traduzindo para o português: integração e entrega contínuas. Trata-se de uma prática de desenvolvimento de software que visa tornar a integração de código mais eficiente por meio de builds e testes automatizados. Com a abordagem CI/CD é possível entregar aplicações com mais frequência aos clientes. Para tanto, regras de automação são aplicadas nas etapas de desenvolvimento de apps.

Os principais conceitos atribuídos ao método são: integração, entrega e implantação contínuas. Com a prática CI/CD é possível solucionar os problemas que a integração de novos códigos pode causar às equipas de operações e desenvolvimento.

Especificamente, CI/CD aplica monitorização e automação contínuos a todo o ciclo de vida das aplicações, incluindo as etapas de teste e integração, além da entrega e implantação. Juntas, essas práticas relacionadas são muitas vezes chamadas de "pipeline de CI/CD". Elas são compatíveis com o trabalho conjunto das equipes de operação e desenvolvimento que usam métodos ágeis, com uma abordagem de DevOps ou de engenharia de confiabilidade de sites (SRE).

Implementação de novas funcionalidades

Containers

Para a criação das imagens que posteriormente vão ser executadas em containers foram utilizados dockerfiles.

```dockerfile
FROM eclipse-temurin:17-jdk-jammy

WORKDIR /app

COPY .mvn/ .mvn
COPY mvnw pom.xml ./
RUN ./mvnw dependency:resolve

COPY src ./src

# Build the application
RUN ./mvnw package -DskipTests

# Expose the port on which the Spring Boot application will run (if applicable)
EXPOSE 8080

CMD ["java", "-jar", "-Dspring.profiles.active=reddisDB", "target/productCommand-0.0.1-SNAPSHOT.jar"]
```

Este Dockerfile cria uma imagem Docker para uma aplicação Java usando o Eclipse Temurin (anteriormente conhecido como AdoptOpenJDK) na versão 17. Aqui estão as principais etapas:

1. **FROM eclipse-temurin:17-jdk-jammy:**
   - Especifica a imagem base a ser usada, que é o Eclipse Temurin (JDK 17) com a tag "jammy". Isso serve como a base para construir a imagem da aplicação Java.
2. **WORKDIR /app:**
   - Define o diretório de trabalho dentro do container como "/app". Todas as instruções subsequentes serão executadas a partir deste diretório.
3. **COPY .mvn/ .mvn:**
   - Copia o diretório ".mvn" (contendo configurações Maven) do host para o diretório ".mvn" no container.
4. **COPY mvnw pom.xml ./RUN ./mvnw dependency:resolve:**
   - Copia o arquivo "mvnw" (Maven Wrapper) e os arquivos "pom.xml" do host para o diretório raiz do container.
   - Executa o comando "mvnw dependency:resolve" para resolver as dependências da aplicação. O Maven Wrapper permite a execução do Maven sem a necessidade de uma instalação prévia do Maven no host.
5. **COPY src ./src:**
   - Copia o diretório "src" do host para o diretório "src" no container.
6. **RUN ./mvnw package -DskipTests:**
   - Executa o comando "mvnw package -DskipTests" para compilar e empacotar a aplicação. O parâmetro "-DskipTests" evita a execução dos testes durante a construção.
7. **EXPOSE 8080:**

- Expõe a porta 8080, indicando que a aplicação dentro do container estará acessível nessa porta.

8. **CMD ["java", "-jar", "-Dspring.profiles.active=reddisDB", "target/productCommand-0.0.1-SNAPSHOT.jar"]:**
   - Define o comando padrão a ser executado quando o container for iniciado. Neste caso, inicia a aplicação Java com o comando "java -jar" usando o arquivo JAR gerado durante a compilação. O perfil "reddisDB" é ativado usando a opção "-Dspring.profiles.active".

CI/CD

Este arquivo YAML representa uma pipeline de integração contínua usando o Bitbucket Pipelines. Aqui estão os principais elementos da pipeline:

```yaml
image: maven:3.6.3

pipelines:
  default:
      - step:
          name: Build and Test
          caches:
          - maven
          script:
          - cd P2/ProductCommand
          - mvn -B verify --file pom.xml
      - step:
          name: Build and Push Docker Image
          services:
          - docker
          script:
            - cd P2/ProductCommand
            - docker build -t 1191018/arqsoft_1190914_1191018_1191042:latest .
            - docker login --username 1191018 --password arqsoft2023
            - docker push 1191018/arqsoft_1190914_1191018_1191042:latest
```

1. **Imagem Base:**
   - image: maven:3.6.3: Define a imagem base a ser usada para a execução da pipeline. Neste caso, é uma imagem Maven na versão 3.6.3.
2. **Pipelines:**
   - pipelines: Define as etapas (steps) da pipeline. A pipeline padrão (default) inclui duas etapas.
3. **Etapa 1: Build and Test:**
   - name: Build and Test: Nomeia a primeira etapa da pipeline como "Build and Test".
   - caches: - maven: Especifica que o cache do Maven será usado para armazenar dependências entre builds, melhorando a eficiência.

- script: - cd P2/ProductCommand - mvn -B verify --file pom.xml: O script realiza a navegação até ao diretório do projeto e executa o comando Maven para compilar e verificar o projeto. O parâmetro -B significa "modo silencioso".

4. **Etapa 2: Build and Push Docker Image:**
   - name: Build and Push Docker Image: Nomeia a segunda etapa da pipeline.
   - services: - docker: Inicia o serviço Docker para permitir a execução de comandos Docker na pipeline.
   - script: - cd P2/ProductCommand - docker build -t 1191018/arqsoft_1190914_1191018_1191042:latest . - docker login --username 1191018 --password arqsoft2023 - docker push 1191018/arqsoft_1190914_1191018_1191042:latest: O script navega até o diretório do projeto, constrói uma imagem Docker com a tag "latest", faz login no Docker Hub usando as credenciais fornecidas e, finalmente, envia a imagem Docker para o Docker Hub.

Esta pipeline é projetada para compilar, testar e construir uma imagem Docker para um projeto Maven, além de enviar essa imagem para o Docker Hub.

Separação em microserviços

Para a separação em microserviços foram utilizados vários padrões já referidos anteriormente, tendo a aplicação inicial sido separada em 9 microserviços:

- ProductCommand
- ProductQuery
- ProductEvent
- ReviewCommand
- ReviewQuery
- ReviewEvent
- VoteCommand
- VoteQuery
- VoteCommand

Para a comunicação entre os vários micro-serviços foi utilizado o Message Broker RabbitMq. Sempre que é feita alguma operação de Command ou Query é enviada uma mensagem para o message broker. Por exemplo, quando um produto é criado no Microserviço ProductCommand, é enviada uma mensagem para a queue que está a ser consumida pelo microserviço ReviewCommand, este vai consumir a mensagem e armazenar a informação que necessita do seu lado. No mesmo momento foi também enviada uma mensagem para a queue que está a ser consumida pelo microserviço ProductEvent, que vai armazenar o evento de criação de um produto.

Para permitir enviar para mais que uma queue foram utilizadas FanoutExchanges que permitem encaminhar mensagens para várias queues que estejam “bounded” à Exchange, permitindo assim enviar a mesma mensagem para várias queues.

Alternativas

Arquitetura

Utilização de Arquitetura Orientada a Serviços. Arquitetura Orientada a Serviços (SOA) é uma abordagem de design de software que estrutura um sistema em torno de serviços de alto nível que são independentes e interoperáveis. Num sistema SOA, cada serviço é uma unidade de negócio independente que pode ser desenvolvida, implantada e escalada separadamente. Os serviços comunicam entre si por meio de protocolos de mensagens, como REST ou SOAP.

Neste caso, a aplicação seria dividida em vários serviços, cada um correspondendo a um dos microserviços que você mencionou (ProductCommand, ProductQuery, ProductEvent, ReviewCommand, ReviewQuery, ReviewEvent, VoteCommand, VoteQuery, VoteCommand). Cada serviço seria responsável por uma parte específica da lógica de negócios e poderia ser desenvolvido, implantado e escalado de forma independente.

A comunicação entre os serviços seria realizada através de protocolos de mensagens, como REST ou SOAP. Por exemplo, quando um produto é criado no Microserviço ProductCommand, uma mensagem seria enviada para o microserviço ReviewCommand através de uma API REST ou SOAP. O microserviço ReviewCommand então consumiria essa mensagem, armazenaria a informação que precisava do seu lado e responderia com uma confirmação.

No entanto esta implementação traz algumas desvantagens:

- A SOA pode ser complexa de implementar, pois requer uma gestão cuidadosa da comunicação entre os serviços. Cada serviço precisa ser capaz de se comunicar com outros serviços através de protocolos de mensagens, como REST
- Performance: A performance pode ser uma preocupação numa arquitetura SOA, pois, cada chamada de serviço pode envolver uma latência de rede adicional. Com um Message Broker, as mensagens são processadas de forma assíncrona, o que pode resultar numa maior eficiência.

Message Broker

Seria possível utilizar outro Message Broker como o Kafka para comunicar entre os microserviços, foi utilizado rabbit devido à sua fácil configuração e utilização e ao facto de seguir um modelo de mensagens padrão com suporte para queues, que é o indicado para este use case, sendo que não é necessário armazenar as mensagens depois destas serem consumidas pelos diferentes microserviços.

FanoutExchange

Em alternativa ao uso de fanoutExchanges as mensagens poderiam ter sido duplicadas e enviadas separadamente para cada queue. No entanto ao seguir esta abordagem iriamos ter algumas desvantagens:

- Maior tráfego de rede
- Processamento Redundante
- Potencial Inconsistência
- Dificuldade na introdução de novas queues e escalabilidade

CI/CD

Para a implementação de CICD foram utilizadas as pipelines do Bitbucket. No entanto poderia ter sido utilizado outro serviço como por exemplo o Jenkins. Como se trata de uma implementação simples de CICD utilizar as pipelines do Bitbucket trouxe algumas vantagens:

- **Integração direta com o Bitbucket**: Não é necessário configurar integrações externas nem adicionar nova infraestrutura
- **Configuração como Código (YAML)**
- **Facilidade de uso**
- **Elasticidade e Dimensionamento Automático**

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.

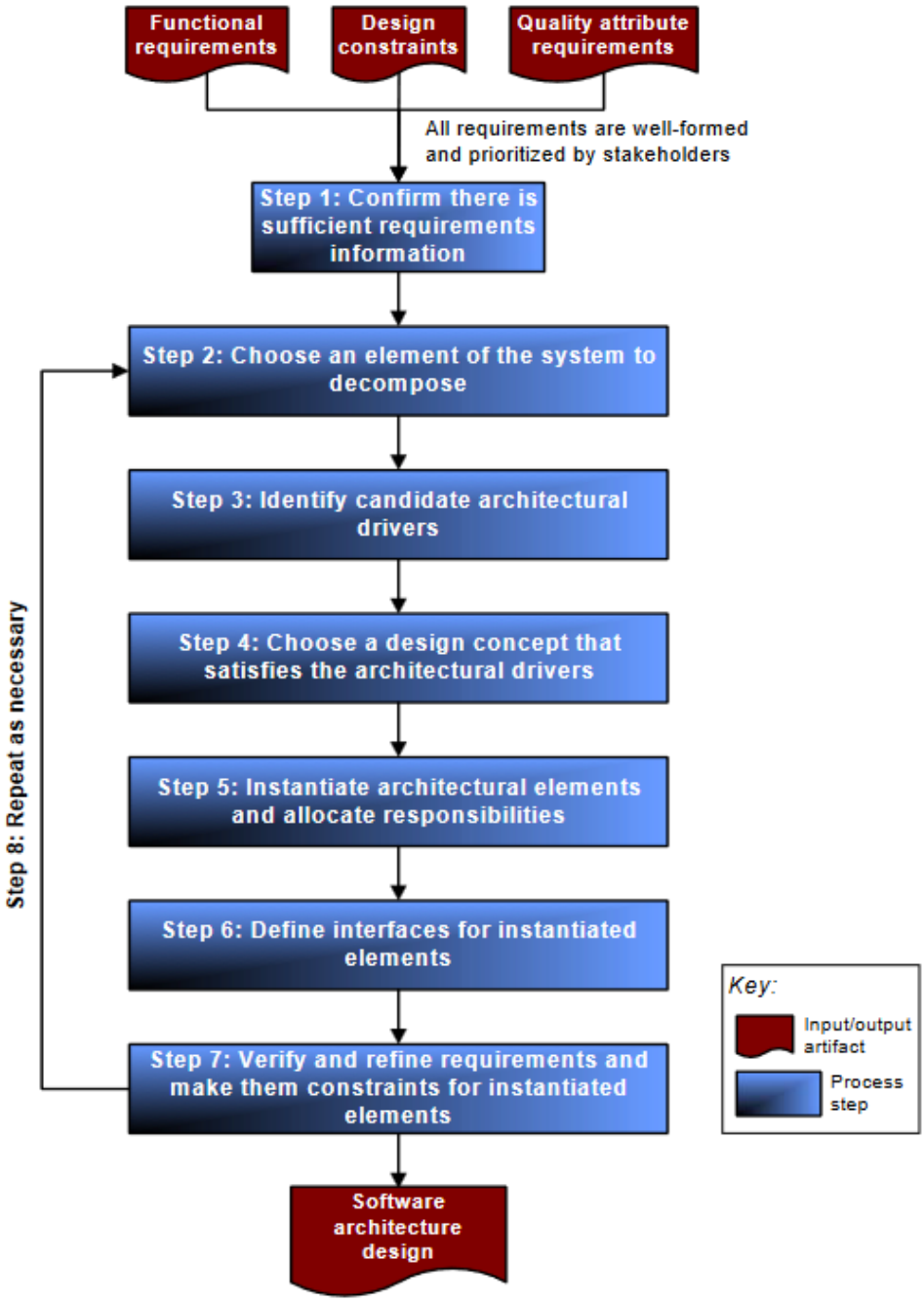


Figura 1 – ADD Diagrama

9.1 Iteração 3

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 architetural 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:

Casos de Uso	Description
UC1:Criar Produto Aprovado	Como product Manager, quero publicar um produto (ou seja, disponível para clientes, reviewer,voter) só depois de dois product managers aceitarem.
UC2: Criar Review Aprovada	Como reviewer quero que a minha review seja aprovada. Para tal, é necessário que dois moderadores aprovem e sejam recomendados por essa mesma review.
UC3: Vote criado juntamente com uma Review	Como voter, quero criar um Vote e uma Review no mesmo processo

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 estar separado em Micro Serviços
CON-6	Utilização do princípio Interface Segregation
CON-7	Utilização de Message Brokers para a comunicação entre os Micro serviços
CON-8	Adoção de CI/CD

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 conceção escolhidos.

9.1.3 ADD Step 3 – Choose What to Refine

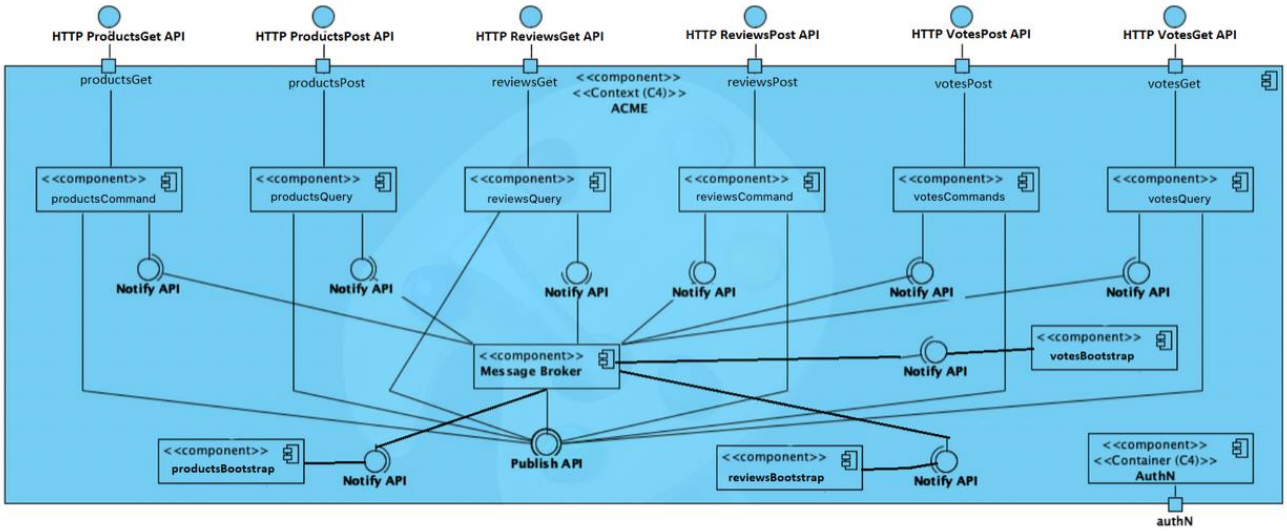
Este passo serve para definir o que vai ser necessário refinar durante a interação.

Com o intuito de otimizar a interação mencionada anteriormente, decidimos aprimorar o projeto no que diz respeito às comunicações dentro da aplicação. Anteriormente, cada base de dados estava associada a um serviço específico e a uma entidade, o que resultava na

necessidade de criar, no mínimo, quatro serviços para quatro bases de dados distintas. Dessa forma, considerando quatro entidades, acabávamos por ter 16 serviços. Assim sendo, pretendemos aperfeiçoar esta abordagem na próxima interação, de modo a garantir a existência de apenas um serviço para cada entidade.

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.

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 4

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 terceira iteração.

9.2.3 ADD Step 3- Choose what to refine

- UC1: Criar Produto Aprovado
- UC2: Criar Review Aprovada
- UC3: Vote criado juntamente com uma Review

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.

- **Utilização de Micro Serviços**

Os microserviços representam uma abordagem arquitetónica que diverge do modelo monolítico tradicional no desenvolvimento de software. Em vez de construir uma única aplicação coesa, os microserviços dividem o sistema em serviços independentes, cada qual responsável por uma função específica.

A principal vantagem dos microserviços reside na flexibilidade que oferecem. Ao fragmentar o sistema em unidades separadas, os desenvolvedores podem trabalhar em serviços individualmente, o que simplifica o processo de desenvolvimento, testes e implementação. Essa modularidade também permite que diferentes serviços sejam desenvolvidos usando tecnologias específicas mais adequadas para suas tarefas, proporcionando uma adaptação eficaz às necessidades particulares de cada componente.

- **Utilização de Brokers**

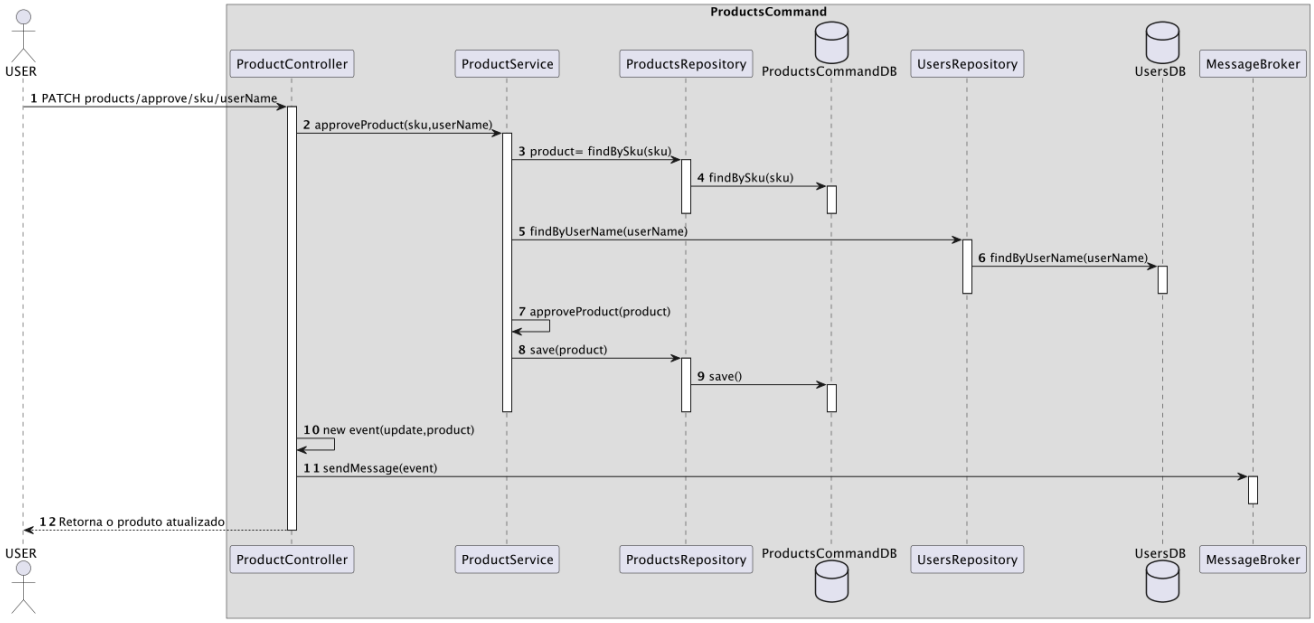
Ao adotar o RabbitMQ, os desenvolvedores podem aproveitar uma variedade de benefícios. Primeiramente, o RabbitMQ oferece uma forma assíncrona de comunicação, onde os serviços podem enviar e receber mensagens de forma eficiente e sem depender da disponibilidade imediata do destinatário. Isso contribui para a resiliência do sistema, pois os serviços podem operar de forma independente, mesmo quando outros estão temporariamente indisponíveis.

Outro ponto positivo do RabbitMQ é a capacidade de lidar com volumes elevados de mensagens de maneira escalável. Ele suporta filas de mensagens, permitindo que os serviços processem as mensagens à medida que estão prontos, evitando sobrecargas repentinas e melhorando a estabilidade do sistema.

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

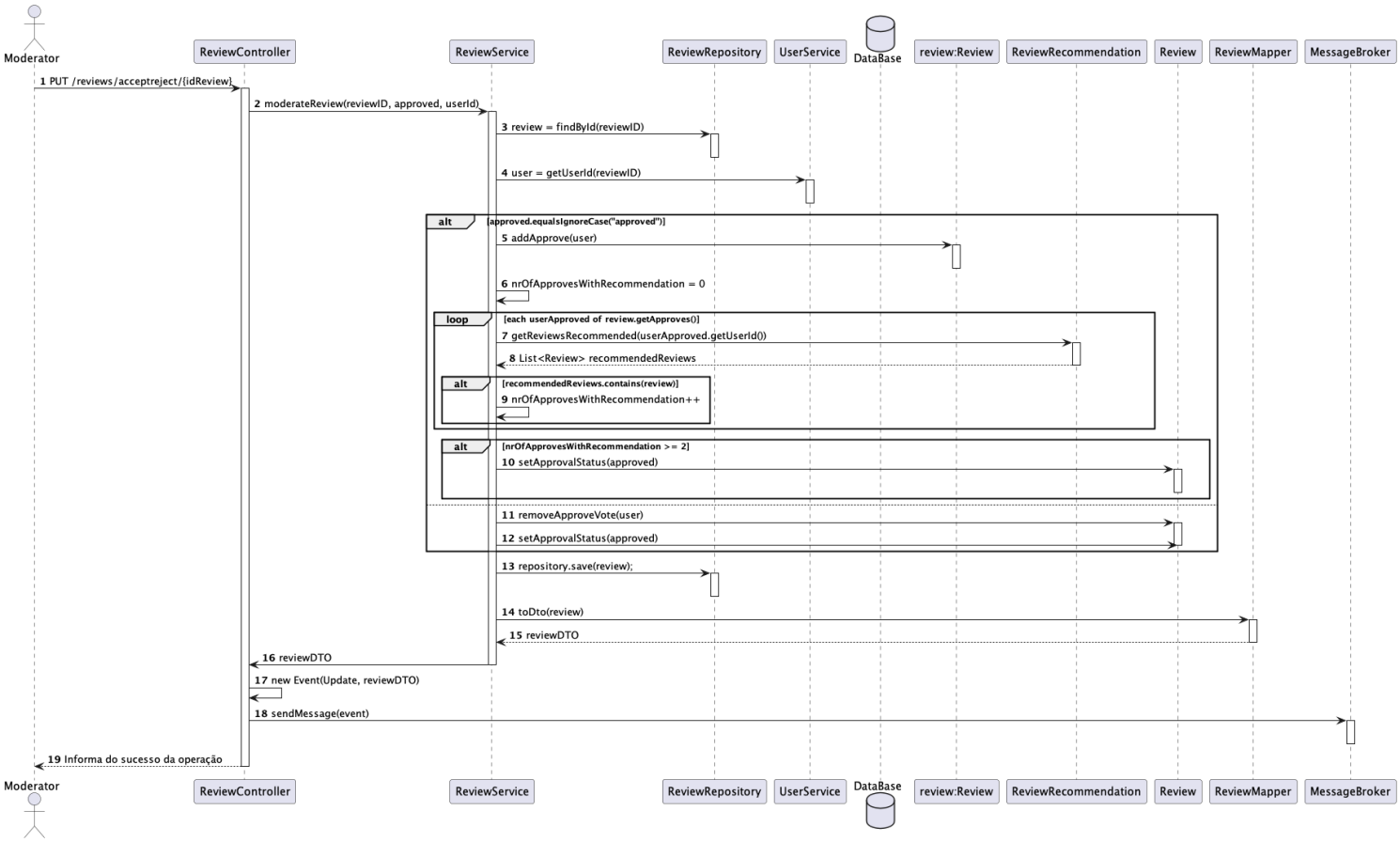
1. **Aprovar Produto**

Como product manager posso aprovar um produto. Para tal deve fazer um pedido HTTP PATCH para o microserviço ProductsCommand passando o sku do produto e o seu username. Se o produto for aprovado por dois product managers, este vai ser publicado.



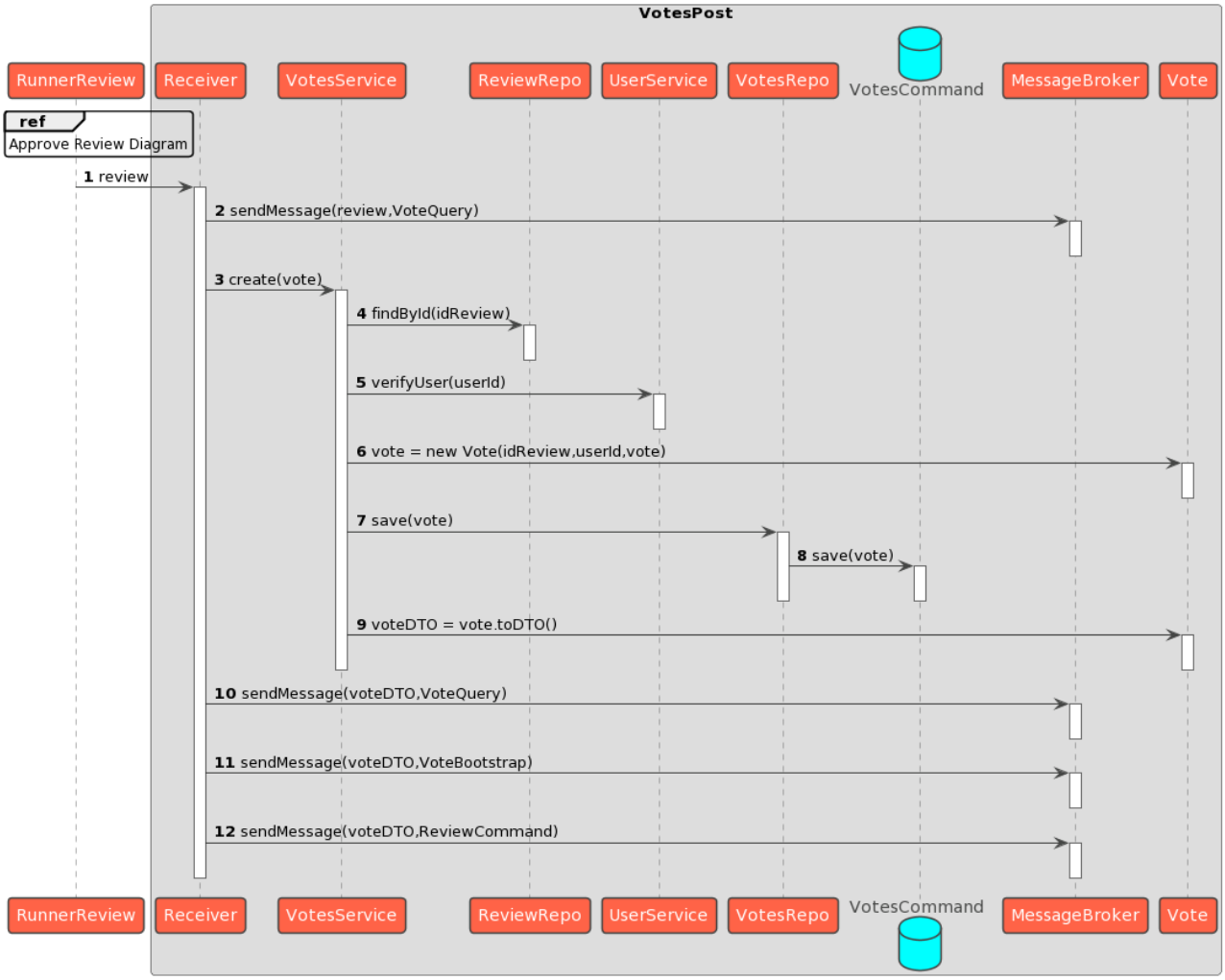
2. Aprovar review

Como reviewer quero que a minha review seja aprovada. Para tal, é necessário que dois moderadores aprovem e sejam recomendados por essa mesma review. Para realizar a aprovação, deve fazer um pedido HTTP PUT para o microserviço ReviewCommand passando o id da Review, o Utilizador e indicar se quer aprovar, rejeitar ou colocar em pendente.



3. Criar Vote

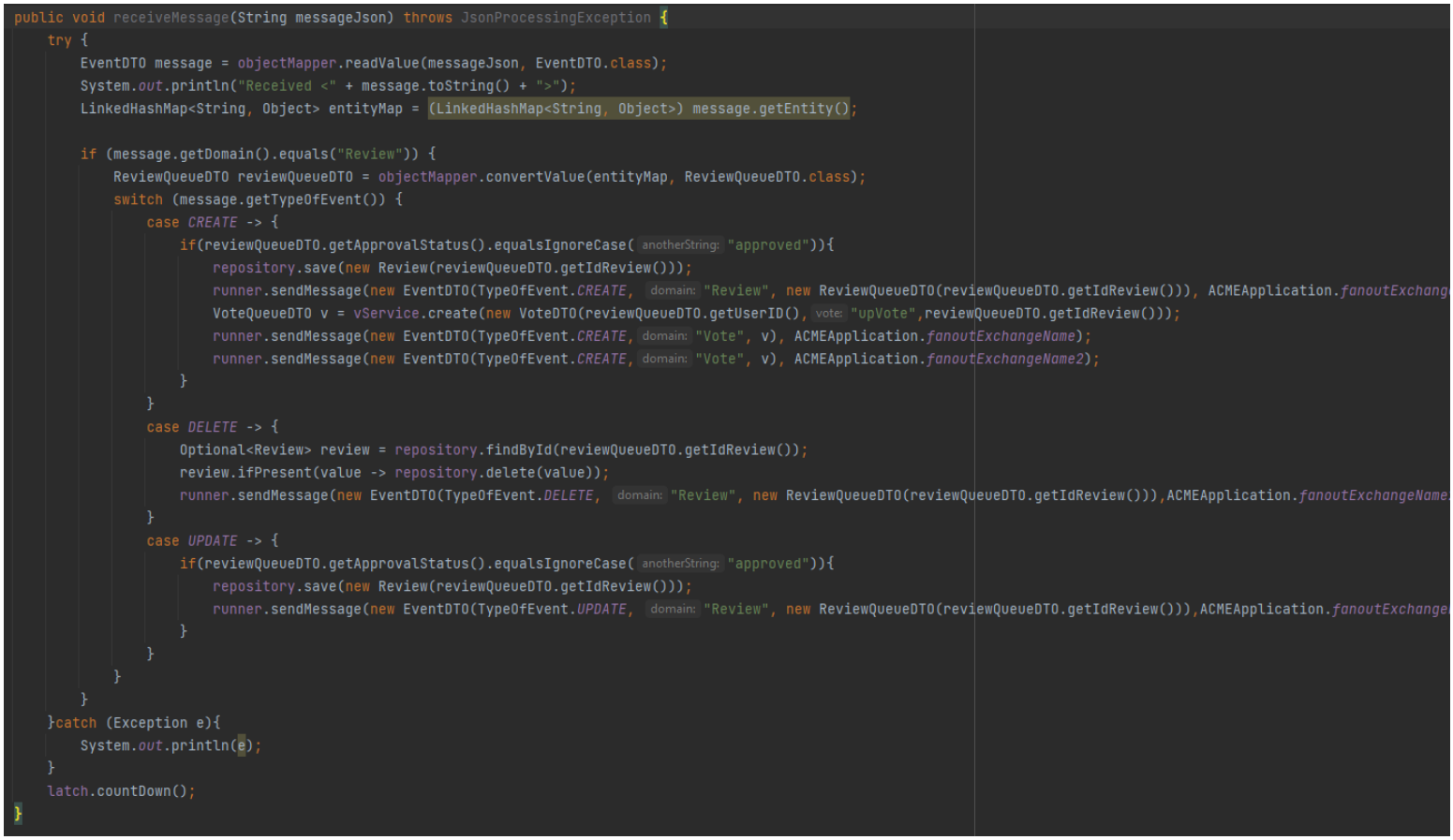
Após a aprovação de uma Review, é criado um voto no mesmo processo. Deste modo, o VoteCommand recebe a review criada e cria um voto com o eleitor e o id da review recebida.



Para garantir o funcionamento harmonioso dos três microserviços, é imperativo que a implementação do RabbitMQ esteja realizada de maneira eficaz. O RabbitMQ desempenha um papel essencial, sen do a espinha dorsal que facilita a comunicação entre esses serviços. Sua integração adequada permite a troca eficiente de mensagens entre os microserviços, promovendo a coesão do sistema como um todo.

Ao garantir uma implementação sólida do RabbitMQ, criamos uma infraestrutura de mensagens resiliente que suporta a comunicação assíncrona entre os serviços, contribuindo para a flexibilidade e escalabilidade da arquitetura de microserviços. A eficácia do RabbitMQ não apenas garante a transmissão eficiente de informações entre os microserviços, mas também fortalece a capacidade do sistema de lidar com cargas variáveis e demandas dinâmicas.

Em resumo, a implementação bem-sucedida do RabbitMQ é um pré-requisito essencial para assegurar a operação sem falhas dos microserviços, promovendo a coesão, eficiência e resiliência do sistema como um todo. Sua função central na comunicação entre os serviços é fundamental para o bom desempenho e integração eficaz dos microserviços.



```java
public void receiveMessage(String messageJson) throws JsonProcessingException {
    try {
        EventDTO message = objectMapper.readValue(messageJson, EventDTO.class);
        System.out.println("Received <" + message.toString() + ">");
        LinkedHashMap<String, Object> entityMap = (LinkedHashMap<String, Object>) message.getEntity();

        if (message.getDomain().equals("Review")) {
            ReviewQueueDTO reviewQueueDTO = objectMapper.convertValue(entityMap, ReviewQueueDTO.class);
            switch (message.getTypeOfEvent()) {
                case CREATE -> {
                    if(reviewQueueDTO.getApprovalStatus().equalsIgnoreCase( anotherString: "approved")){
                        repository.save(new Review(reviewQueueDTO.getIdReview()));
                        runner.sendMessage(new EventDTO(TypeOfEvent.CREATE, domain: "Review", new ReviewQueueDTO(reviewQueueDTO.getIdReview())), ACMEApplication.fanoutExchang
                        VoteQueueDTO v = vService.create(new VoteDTO(reviewQueueDTO.getUserID(), vote: "upVote",reviewQueueDTO.getIdReview()));
                        runner.sendMessage(new EventDTO(TypeOfEvent.CREATE, domain: "Vote", v), ACMEApplication.fanoutExchangeName);
                        runner.sendMessage(new EventDTO(TypeOfEvent.CREATE, domain: "Vote", v), ACMEApplication.fanoutExchangeName2);
                    }
                }
                case DELETE -> {
                    Optional<Review> review = repository.findById(reviewQueueDTO.getIdReview());
                    review.ifPresent(value -> repository.delete(value));
                    runner.sendMessage(new EventDTO(TypeOfEvent.DELETE, domain: "Review", new ReviewQueueDTO(reviewQueueDTO.getIdReview())),ACMEApplication.fanoutExchangeName
                }
                case UPDATE -> {
                    if(reviewQueueDTO.getApprovalStatus().equalsIgnoreCase( anotherString: "approved")){
                        repository.save(new Review(reviewQueueDTO.getIdReview()));
                        runner.sendMessage(new EventDTO(TypeOfEvent.UPDATE, domain: "Review", new ReviewQueueDTO(reviewQueueDTO.getIdReview())),ACMEApplication.fanoutExchange
                    }
                }
            }
        }
    }catch (Exception e){
        System.out.println(e);
    }
    latch.countDown();
}
```

A imagem superior representa um método essencial na classe Receiver do VoteCommand. Este método foca exclusivamente na receção de Reviews. Quando uma nova Review é criada e o seu estado de aprovação é marcado como "approved", o sistema procede ao armazenamento na base de dados. Em seguida, automaticamente, é gerada a criação de um voto. Após a conclusão deste processo, uma mensagem contendo o voto é enviada para tanto o VoteQuery quanto para o ReviewCommand. Este procedimento visa assegurar que ambos os micro serviços armazenem o voto nas respetivas bases de dados, mantendo a integridade e consistência das informações.

A lógica implementada neste método garante uma abordagem eficaz e organizada. Ao responder à aprovação de uma Review com a execução de tarefas subsequentes, como o armazenamento na base de dados e a criação de um voto, o sistema mantém a coerência entre os diversos componentes. A comunicação fluida com o VoteQuery e o ReviewCommand, mediante o envio da mensagem com o voto, evidencia uma abordagem orientada à colaboração entre os microserviços, promovendo uma operação integrada e eficiente do sistema como um todo.

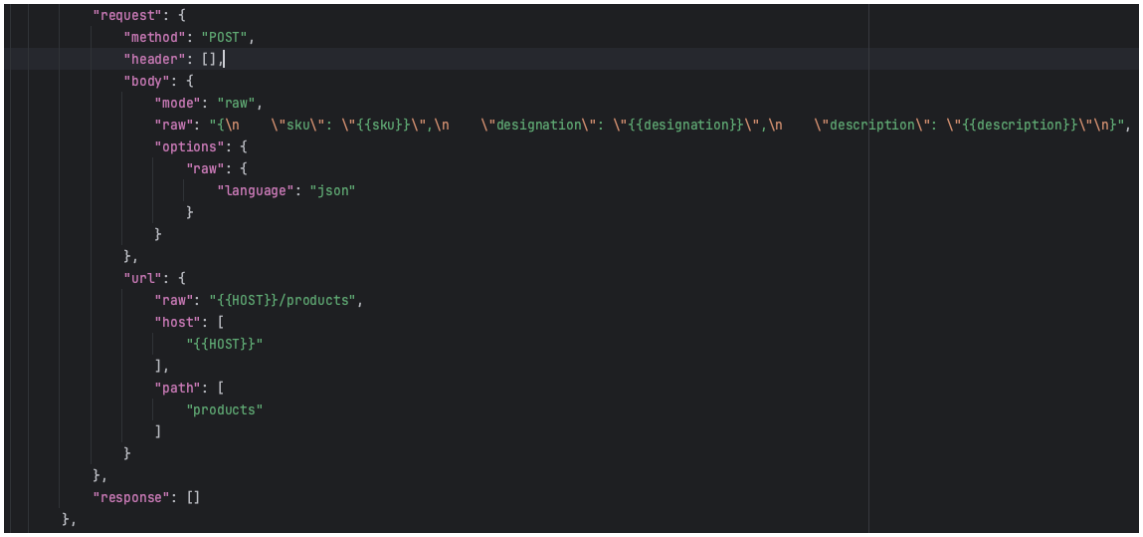
Testes

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 de carga	Os testes de carga desempenham um papel crucial na avaliação do desempenho de sistemas e aplicações. Sua principal finalidade é verificar como o software reage sob condições de demanda extrema, permitindo identificar limites de capacidade, avaliar estabilidade, otimizar desempenho e prevenir falhas.

Testes unitários (exemplo)

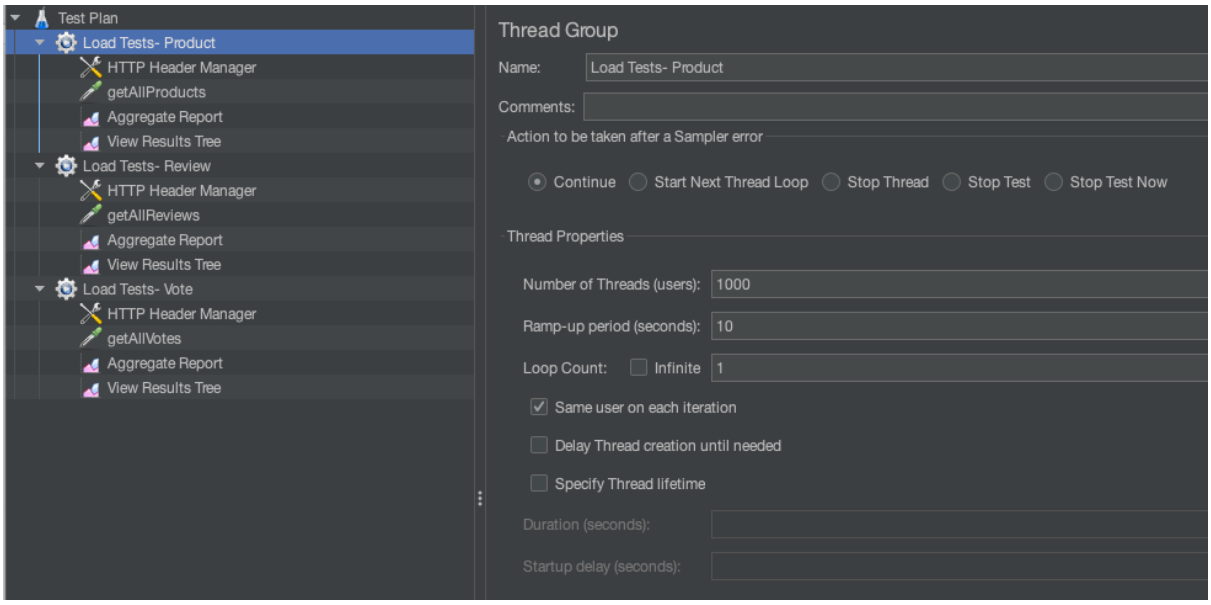
```java
@Test
void createVote() {
    Long userId = 50L;
    String voteText = "downVote";
    Long reviewId = 150L;

    VoteDTO createVoteDTO = new VoteDTO(userId, voteText, reviewId);

    User user = new User(userId);

    when(reviewRepository.findById(reviewId)).thenReturn(Optional.of(new Review()));
    when(userService.getUserId(userId)).thenReturn(Optional.of(user));
    when(voteRepository.findVote(any())).thenReturn( t: false);

    Vote vote = new Vote( voteId: 1L, userId: 5L, vote: "upVote", reviewId: 10L);
    when(voteRepository.save(any())).thenReturn(vote);

    VoteQueueDTO result = voteService.create(createVoteDTO);

    verify(reviewRepository).findById(reviewId);
    verify(voteRepository).findVote(any());
    verify(voteRepository).save(any());
```

Testes de End2End (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": []
},
```

Testes de carga (exemplo)



Teste de Integração

```
5 usages
private VoteServiceImpl voteService;

1 usage
@Autowired
private UserServiceImpl userService;

2 usages
@Autowired
private VoteRepositoryMongo voteRepositoryImp;

2 usages
@Autowired
private ReviewRepositoryMongo reviewRepositoryImp;


no usages
@BeforeEach
void setUp(){
    voteService = new VoteServiceImpl();
    voteService.setVoteRepo(voteRepositoryImp);
    voteService.setVoteReviewRepo(reviewRepositoryImp);
    voteService.setVoteUserService(userService);
}

no usages
@Test
public void testCreateVote() {
    Long userId = 1L;
    String vote = "upVote";
    Long reviewId = 10L;
    Review review = new Review( idReview: 10L);
    reviewRepositoryImp.save(review);
    VoteQueueDTO v = voteService.create(new VoteDTO(userId, vote,reviewId));

    Assertions.assertEquals(v.getVote(),vote);
    Assertions.assertEquals(v.getUserId(),userId);
    Assertions.assertEquals(v.getReviewId(),reviewId);

    voteRepositoryImp.deleteById(v.getVoteId());
}
```

9.2.6 ADD Step 5 - Analyze Current Design, and Review Iteration Goal + Achievement of Design Purpose

Não Elaborado	Elaborado
	CRN-1
	CRN-2
	UC1
	UC2
	UC3
	QA-1
	QA-2
	CON-1
	CON-2
	CON-3
	CON-4
	CON-5
	CON-6
	CON-7
	CON-8