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

Compromissos da implementação de arquiteturas de microsserviços utilizando o modelo de atores

Wander Douglas Andrade de Souza

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman
Cossupervisor: João Francisco Lino Daniel
Cossupervisor: Renato Cordeiro Ferreira
Cossupervisora: Thatiane de Oliveira Rosa

São Paulo
15 de Fevereiro de 2022

Resumo

Wander Douglas Andrade de Souza. **Compromissos da implementação de arquiteturas de microsserviços utilizando o modelo de atores**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Cada vez mais usuários procuram por aplicações com alta responsividade e disponibilidade. Além disso, as equipes de desenvolvimento têm se tornado cada vez maiores, o que exige uma nova maneira de organizá-las. Para atender a essas demandas, tem sido recorrente adotar o estilo arquitetural de microsserviços. Uma das principais características desse estilo é o baixo acoplamento entre os serviços, o que permite maior independência e escalabilidade do sistema. De modo a aproveitar o máximo dessas características, uma das abordagens recomendadas é a reativa. Nessa abordagem, a comunicação entre serviços é assíncrona, ou seja, um serviço que necessita se comunicar com outros emite uma mensagem, e a responsabilidade de reagir ou não a ela pertence a cada serviço. Um dos arcabouços mais utilizados para implementar arquiteturas reativas é o Akka, uma plataforma que implementa o modelo de atores, um modelo conceitual para lidar com concorrência que regra sobre como os componentes de um sistema devem se comportar e interagir entre si. Apesar dos benefícios descritos anteriormente, a escolha de uma plataforma reativa para implementar sistemas reativos necessita ser cautelosa, pois esse paradigma traz uma mudança na forma de pensar a solução para sistemas de grande escala. Considerando esse contexto, o objetivo deste trabalho é analisar as vantagens e desvantagens na implementação de sistemas reativos utilizando programação reativa. Para cumprir com objetivo proposto, a estratégia adotada foi comparar duas implementações de padrões arquiteturais de microsserviços. Para isso, foram implementadas duas versões do padrão Saga, sendo uma versão baseada no modelo de atores e a outra não. Além disso, por meio de um questionário foram coletadas as percepções de desenvolvedores de software sobre a implementação de padrões de microsserviços utilizando modelo de atores e programação reativa, onde os participantes informaram o nível de conhecimento sobre os padrões investigados, analisaram e a compararam as implementações do padrão Saga com e sem o modelo de atores. Durante o desenvolvimento das duas implementações do padrão Saga, foi perceptível que o modelo de atores proveu boas estruturas para lidar com a persistência que adota o padrão Event Sourcing, presente na implementação escolhida do padrão arquitetural de microsserviços Saga. A facilidade de configurar comportamentos para otimizar a recuperação de estado em memória de entidades, o *snapshot*, também destacou a implementação utilizando modelo de atores quando comparada com a implementação sem o uso do modelo. Os resultados obtidos na pesquisa feita com desenvolvedores indicou que mesmo sem experiência prévia com os conceitos de modelo de atores e programação reativa, os participantes do estudo conseguiram compreender de forma satisfatória a implementação do padrão Saga com modelo de atores utilizando a plataforma Akka, demonstrando a alta expressividade da ferramenta, apesar da sua alta curva de aprendizado. Com os resultados obtidos durante a implementação dos padrões e na pesquisa feita com desenvolvedores, pode-se concluir que a ferramenta Akka se mostrou benéfica na implementação de aplicações que adotam o modelo de atores.

Palavras-chave: modelo de atores. programação reativa. microsserviços.

Abstract

Wander Douglas Andrade de Souza. **Trade-offs of implementing microservices architectures using the actor model**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

More and more users are looking for applications with high responsiveness and availability. In addition, development teams have become larger and larger, requiring a new way to organize them. To meet these demands, it has been recurrent to adopt the microservices architectural style. One of the main characteristics of this style is the low coupling between the services, which allows greater independence and scalability of the system. In order to make the most of these characteristics, one of the recommended approaches is the reactive approach. In this approach, the communication between services is asynchronous, that is, a service that needs to communicate with others sends a message, and the responsibility to react or not to this message belongs to each service. One of the most used frameworks for implementing reactive architectures is Akka, a platform that implements the actor model, a conceptual model for handling concurrency that rules about how the components of a system should behave and interact with each other. Despite the benefits described above, the choice of a reactive platform to implement reactive systems needs to be cautious, because this paradigm brings a change in the way of thinking about the solution for large-scale systems. Considering this context, the objective of this paper is to analyze the advantages and disadvantages in implementing reactive systems using reactive programming. To meet the proposed objective, the strategy adopted was to compare two implementations of microservices architectural patterns. For this, two versions of the Saga pattern were implemented, one version based on the actor model and the other not. Furthermore, a survey was used to collect software developers' perceptions about the implementation of microservices patterns using the actor model and reactive programming, where the participants informed their level of knowledge about the patterns investigated and analyzed and compared the implementations of the Saga pattern using and not using the actor model. During the development of the two implementations of the Saga pattern, it was noticeable that the actor model provided good frameworks for dealing with persistence that adopts the Event Sourcing pattern, present in the chosen implementation of the Saga microservices architectural pattern. The ease of configuring behaviors to optimize in-memory state retrieval of entities, *snapshot*, also highlighted the implementation using the actor model when compared to the implementation without the use of the model. The results obtained from the survey of developers indicated that even without prior experience with the concepts of actor model and reactive programming, the participants of the study were able to satisfactorily understand the implementation of the Saga pattern with actor model using the Akka platform, demonstrating the high expressiveness of the tool, despite its high learning curve. With the results obtained during the pattern implementation and in the survey done with developers, it can be concluded that the Akka tool proved beneficial in the implementation of applications that adopt the actors model.

Keywords: actor model. reactive programming. microservices.

Lista de Figuras

Sumário

Apêndices

Capítulo 1

Introdução

Cada vez mais usuários procuram por aplicações com alta responsividade e disponibilidade. Além disso, as equipes de desenvolvimento têm se tornado cada vez maiores, o que exige uma nova maneira de organizá-las [Bonér, 2016]. Para atender a essas demandas, tem sido recorrente adotar o estilo arquitetural de microsserviços. Uma das principais características desse estilo é o baixo acoplamento entre os serviços, o que permite maior independência e escalabilidade do sistema [Newman, 2015][Ford *et al.*, 2017].

De modo a aproveitar melhor essas características, uma das abordagens recomendadas é a reativa. Nessa abordagem, a comunicação entre serviços é assíncrona, ou seja, não há acoplamento temporal entre os serviços, permitindo uma maior flexibilidade no tempo de resposta das requisições ao serviço.

Um dos arcabouços utilizados para implementar arquiteturas reativas é o Akka, uma plataforma que implementa o modelo de atores. Este é um modelo conceitual para lidar com concorrência que apresenta regras gerais sobre como os componentes de um sistema devem se comportar e interagir entre si [Roestenburg *et al.*, 2015].

Apesar das características descritas anteriormente, a escolha de uma plataforma reativa para implementar sistemas reativos necessita ser cautelosa, pois esse paradigma traz uma mudança na forma de pensar a solução para sistemas de grande escala. Considerando esse contexto, o objetivo deste trabalho é identificar as vantagens e desvantagens na adoção do modelo de atores na implementação de arquiteturas reativas de microsserviços.

Este trabalho está estruturado da seguinte forma: O capítulo 2 detalha os conceitos técnicos importantes para a compreensão do trabalho. O capítulo 3 detalha a proposta, apresentando as etapas planejadas para alcançar o objetivo descrito. O capítulo 4 apresenta os resultados obtidos.

Capítulo 2

Fundamentação Teórica

2.1 Arquitetura de Software

Arquitetura de software é "o conjunto de estruturas necessárias para a compreensão de um sistema, que abrange elementos que o compõem, a relação entre eles e suas propriedades" [Bass *et al.*, 2013]. No contexto de arquitetura de software, um conceito muito importante é o de estilo arquitetural. Estilo arquitetural é o conjunto de características que são comuns entre a essência de diversas arquiteturas concretas semelhantes [Richardson, 2018]. Existe uma variada gama de estilos arquiteturais, este trabalho explora especificamente o estilo arquitetural baseado em microsserviços, abordado na próxima seção.

2.1.1 Estilo Arquitetural de Microsserviços

Atualmente, confiabilidade, escalabilidade e disponibilidade são algumas das características mais relevantes ao desenvolver sistemas e seus serviços. Além disso, com o aumento da complexidade dos sistemas, houve um aumento do número de equipes para atender às demandas técnicas dos sistemas. Esse acréscimo traz desafios de organizar grandes equipes de desenvolvimento de software de maneira a otimizar o tempo de entrega de novas funcionalidades e oferecer manutenção aos sistemas [Newman, 2015]. Para atender a tais requisitos, o estilo arquitetural de microsserviços tem sido adotado [Richardson, 2018]. Um dos fatores que favorece a adoção desse estilo arquitetural nesse contexto é que, conforme afirma [Fowler, Lewis, 2014], com microsserviços, o sistema é dividido em pequenas partes distribuídas e autônomas – os microsserviços – focadas em um *Bounded Context* [Evans, 2004].

2.1.2 Padrões Arquiteturais Em Microsserviços

Um padrão arquitetural é uma solução reutilizável para um problema que ocorre recorrentemente na arquitetura de *software* [Richardson, 2018]. Este trabalho foca nas soluções dos problemas envolvendo arquiteturas de microsserviços, tais como o CQRS e o Saga.

Saga

Em uma arquitetura distribuída, como a de microsserviços, que adota comunicação assíncrona, uma das estratégias adotadas para implementar transações de dados é criar transações locais em cada um dos microsserviços que, ao final, emitem uma mensagem notificando seu resultado. Com isso os demais serviços poderão reagir a esta mensagem com suas respectivas transações locais. Algumas das mensagens podem ser de falha, quando seria necessário fazer um "*roll back*" em transações, e, para isso, é importante implementar transações locais de compensação [Richardson, 2018].

A figura 2.1 ilustra um exemplo de uma Saga em um fluxo de um pedido em um *e-commerce*. Tal Saga é composta por diversas transações locais, tais como 1, 2, e 4, em que cada uma altera dados de um único microsserviço.

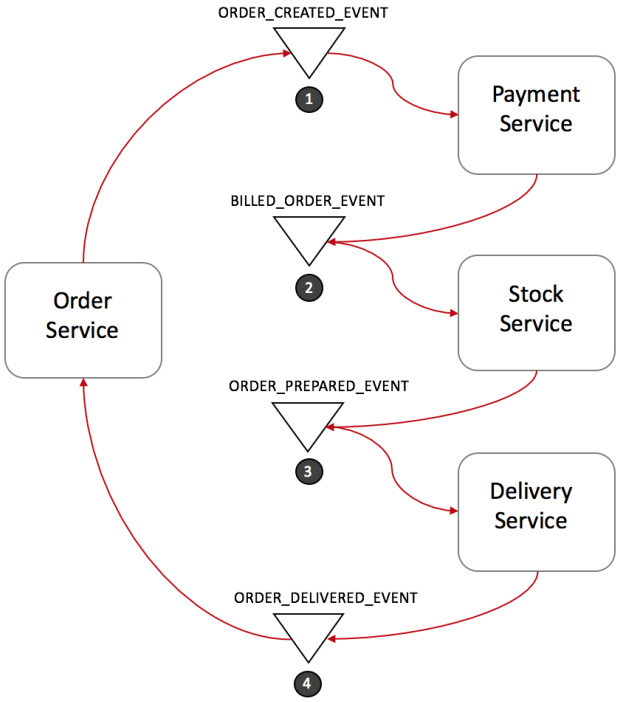


Figura 2.1: *Exemplo de uma Saga com diversas transações locais, cada uma alterando dados em um único microsserviço [Rosa, 2018]*

Event Sourcing

Em uma arquitetura de microsserviços orientada a eventos, um serviço, muitas vezes, necessita atualizar a sua base de dados e emitir mensagens ou eventos. Por exemplo, ao implementar o padrão Saga, um serviço atualiza a sua base de dados e emite um evento descrevendo a mudança ocorrida para o próximo serviço, envolvido na sequência de transações locais. Todavia, é interessante que esse conjunto de operações seja realizado de maneira atômica, a fim de evitar inconsistências de dados. As transações ACID (acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade) entre serviços também não são uma solução interessante, pois acarretam um acoplamento forte entre serviços e suas respectivas bases de dados.

O padrão Event Sourcing propõe um método de gerenciamento de dados, no qual o serviço armazena o estado de uma entidade da base dados como uma sequência ordenada de eventos, que descrevem as alterações ocorridas desde sua criação. Sempre que o estado de uma entidade precisa ser alterado, um evento descrevendo a mudança necessária é adicionado à lista de eventos da entidade[Richardson, 2018].

A figura 2.2 ilustra uma tabela de banco de dados em um sistema que adota o padrão Event Sourcing. Nela, há um registro do histórico dos eventos ocorridos com a entidade *Order* de *id* 101.

event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...	...	...	...	...

Figura 2.2: *Exemplo de uma tabela de um banco de dados de uma aplicação que adota o padrão Event Sourcing [Richardson, 2018]*

Command Query Responsibility Segregation - CQRS

Em uma aplicação monolítica tradicional, consultas complexas envolvendo diversas entidades (tabelas) podem ser realizadas de forma trivial, pois elas estão inseridas em um mesmo banco de dados. Em uma arquitetura distribuída, como a de microsserviços, cada serviço possui seu próprio banco de dados, tornando as consultas, que necessitam de dados de várias tabelas, mais difíceis.

Nesse contexto, o padrão CQRS propõe uma separação de responsabilidades por meio do desacoplamento dos modelos de escrita e leitura na base de dados dos serviços, permitindo otimizações que antes não eram possíveis, devido ao acoplamento dos modelos, tal como a criação de um modelo para otimizar consultas nas quais é necessário consultar diversos serviços.

A figura 2.3 ilustra uma aplicação que adota o padrão CQRS. Nela, por meio de uma interface gráfica (UI - *User Interface*), um cliente faz uma requisição para o serviço *Command* que altera a base de dados. Após isso, o serviço de *Query* é notificado da alteração e realiza as mudanças necessárias no banco de dados de leitura.

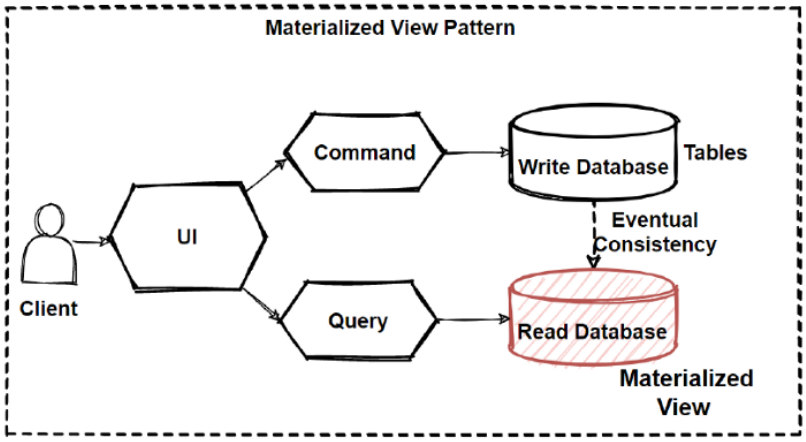


Figura 2.3: *Exemplo de aplicação do padrão CQRS [Özkaya, 2021]*

2.2 Reatividade

Esta seção define e diferencia os conceitos para sistemas reativos e programação reativa.

2.2.1 Sistemas Reativos

Adotar a orientação a eventos como meio de comunicação entre os serviços aumenta a escalabilidade e a taxa de transferência dos sistemas. Serviços que enviam mensagens não necessitam ficar ociosos esperando por uma resposta, e uma mesma mensagem pode ser consumida por diversos outros serviços, se publicados em um sistema de mensageria [Bonér, 2017].

Para Bonér, 2017, um sistema reativo é um estilo arquitetural que permite que múltiplas aplicações individuais se unam como uma unidade, reagindo ao redor, enquanto se mantêm cientes sobre cada um dos elementos internos. Para isso, devem atender às seguintes características definidas no Manifesto Reativo, proposto por Bonér, Farley, Kuhn e Thompson (Tradução oficial, 2014):

- "**Responsivos:** O sistema responde em um tempo razoável, se possível.[...] Sistemas responsivos visam prover tempos de resposta curtos e consistentes, estabelecendo margens de tolerância confiáveis que garantem uma qualidade de serviço consistente [...].
- **Resilientes:** O sistema continua responsivo em caso de falha [...]. Falhas são contidas dentro de cada componente, isolando-os uns dos outros, portanto, garantindo que partes do sistema podem falhar e se recuperar sem comprometer o sistema como um todo [...].
- **Elásticos:** O sistema continua responsivo mesmo sob variações de carga de demanda. Sistemas Reativos podem reagir a mudanças na taxa de solicitações por meio do aumento ou diminuição dos recursos alocados para lidar com essas entradas [...].

- **Orientados a Mensagens:** Sistemas Reativos baseiam-se em transferência de mensagens assíncronas para estabelecer fronteiras entre os componentes e garantir baixo acoplamento, isolamento, transparência na localização [...]. "

2.2.2 Programação Reativa

Programação reativa é um subconjunto da programação assíncrona e um paradigma de programação, em que a disponibilidade de novas informações orienta a lógica de computação interna dos componentes de uma aplicação. Esse paradigma propõe decompor um problema em etapas menores, em que cada uma pode ser executada de maneira assíncrona e sem bloqueios, e então serem combinadas para produzir um fluxo de trabalho. Com isso, tal paradigma auxilia a lidar com um dos maiores obstáculos para a escalabilidade de aplicações, que é a contenção [Bonér, 2016].

Uma das aplicações da programação reativa é a implementação do modelo de atores, um modelo conceitual para lidar com computação concorrente. A próxima seção explora os conceitos fundamentais de modelos de atores.

2.2.3 Modelo de Atores

O modelo de atores é um modelo conceitual criado para lidar com computação concorrente. Ele define algumas regras gerais sobre como os componentes de um sistema devem se comportar e interagir entre si [Vernon, 2015].

O principal elemento desse modelo é o “ator”, que é uma unidade primitiva de computação, que recebe uma mensagem com alguma informação e aplica uma lógica de negócio. Além disso, não compartilham estado entre si.

Logo, a comunicação entre atores se dá por troca de mensagens. Quando um ator recebe uma mensagem, ele pode fazer uma das seguintes ações: Criar mais atores; Enviar outras mensagens para outros atores; Determinar o que fazer com a próxima mensagem [Vernon, 2015].

Apesar de múltiplos atores poderem ser executados simultaneamente, cada ator só processa as mensagens direcionadas a ele de maneira sequencial. Essas mensagens, enviadas de maneira assíncrona, são armazenadas no *mailbox*.

A figura 2.4 ilustra o esquema geral de interação em um modelo de atores, em que mensagens assíncronas que são enviadas entre atores e que cada ator possui seu estado interno e uma *mailbox*, na qual as mensagens são armazenadas antes de serem processadas.

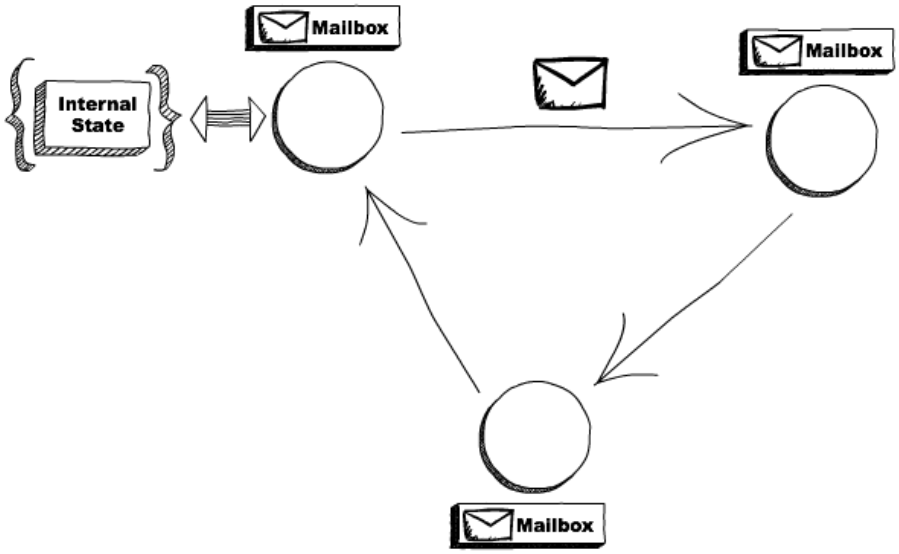


Figura 2.4: *Funcionamento interno de um modelo de atores [Storti, 2015]*

Capítulo 3

Metodologia

Para alcançar o objetivo proposto neste trabalho, ou seja, identificar vantagens e desvantagens da adoção do modelo de atores na implementação de arquiteturas reativas de microsserviços. A estratégia adotada foi comparar duas implementações de padrões arquiteturais de microsserviços. Para isso, foram implementadas duas versões dos padrões CQRS e Saga, sendo uma versão baseada no modelo de atores e a outra não. Para implementar a versão com o modelo de atores, foi adotada a plataforma Akka, que é baseada nesse modelo e permite o desenvolvimento de aplicações distribuídas, concorrentes, resilientes e orientadas a mensagens. Após implementar cada uma das versões (com e sem o modelo de atores) foram avaliados os aspectos: organização, coesão, e legibilidade. A Figura 3.1 ilustra as principais etapas da metodologia e o fluxo de execução da pesquisa.

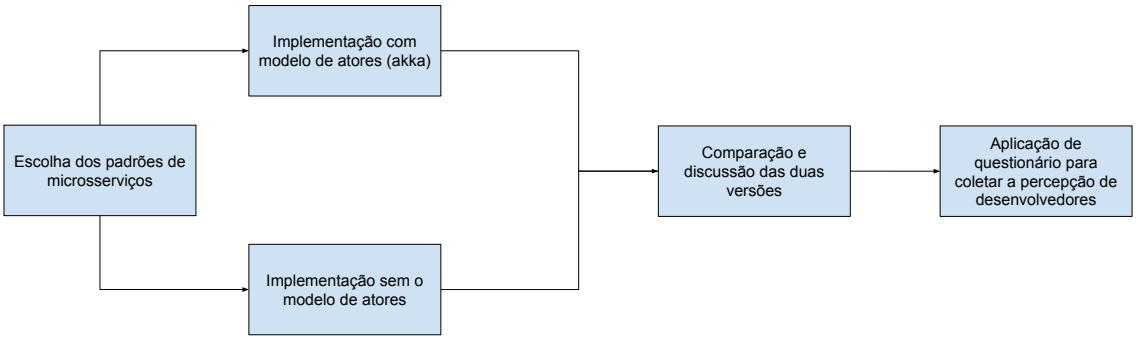


Figura 3.1: *Fluxo da metodologia*

A primeira escolha de padrão foi o CQRS, por utilizar comunicação assíncrona entre os serviços que o compõe. Para implementá-lo, selecionou-se um caso de uso de uma agenda telefônica simples, no qual há o cadastro de usuários na base de dados e eles podem ser consultados por meio de um *id* identificando o usuário de quem se deseja obter informações.

Após desenvolver as duas versões do padrão CQRS (com e sem modelo de atores), o resultado foi apresentado e discutido com um grupo de pesquisa do IME/USP, focado no estudo de microsserviços. A partir dos *feedbacks* recebidos, decidiu-se não dar continuidade

às análises relacionadas a esse padrão. Essa decisão se justifica pelo fato de que o caso de uso escolhido para implementação do CQRS ser pouco complexo, isto é, com poucas interações assíncronas entre os microsserviços envolvidos, o que poderia afetar a qualidade da comparação entre as duas versões, sobretudo, considerando a natureza assíncrona do modelo de atores na versão com Akka.

Na busca de um novo padrão arquitetural a ser utilizado como objeto de pesquisa, o escolhido foi o Saga, principalmente pelas interações assíncronas presentes nele e pelo desafio de realizar transações de dados no contexto de sistemas distribuídos. A implementação seguiu o caso de uso de um fluxo de aprovação de um pedido em um aplicativo de entrega de comida.

Após o desenvolvimento das versões da Saga do caso de uso escolhido, assim como no padrão CQRS, o resultado foi apresentado e discutido com o mesmo grupo de pesquisa. A partir disso, obteve-se feedbacks e foi possível comparar as duas versões da Saga (com e sem modelo de atores), levantando qualidades e defeitos de cada abordagem adotada.

Por fim, com o objetivo de coletar as percepções de desenvolvedores de *software* sobre a implementação de padrões de microsserviços utilizando modelo de atores e programação reativa, foi elaborado um questionário (Apêndice A), onde os participantes informaram o nível de conhecimento sobre os padrões investigados analisaram e a compararam as implementações do padrão Saga com e sem o modelo de atores.

Capítulo 4

Resultado e Discussões

Este capítulo apresenta e discute os resultados e análises dos experimentos realizados com os padrões CQRS e Saga e o modelo de atores implementado com Akka. Além disso, explora as percepções de 16 desenvolvedores, no que tange à legibilidade, organização e concisão do código de diferentes versões de implementação do padrão Saga.

4.1 Experimentos

Esta seção apresenta a implementação de experimentos feitos com os padrões CQRS e Saga. Para cada padrão, foram feitas duas implementações: uma utilizando modelo de atores e outra não utilizando modelo de atores. Cada versão adota um caso de uso específico, aonde cada um dos padrões são aplicados.

4.1.1 Implementação do Padrão CQRS

Para implementar o padrão CQRS, foi escolhido o caso de uso de uma agenda telefônica, em que cada usuário possui os atributos **id**, **nome** e **telefone**. Para tanto, dois serviços foram criados, denominados **UserWrite** e **UserRead**. O serviço **UserWrite** é responsável por armazenar eventos relacionados à criação de usuário e à alteração de dados dos mesmos. Já o serviço **UserRead** consome as alterações e armazena, de maneira otimizada, para leitura na sua base de dados.

Cada um dos serviços também possui uma base de dados própria. Além disso, há um *message broker* que recebe as mensagens publicadas e notifica os serviços interessados nelas. Todos esses elementos e as interações entre eles estão ilustrados na figura 4.1.

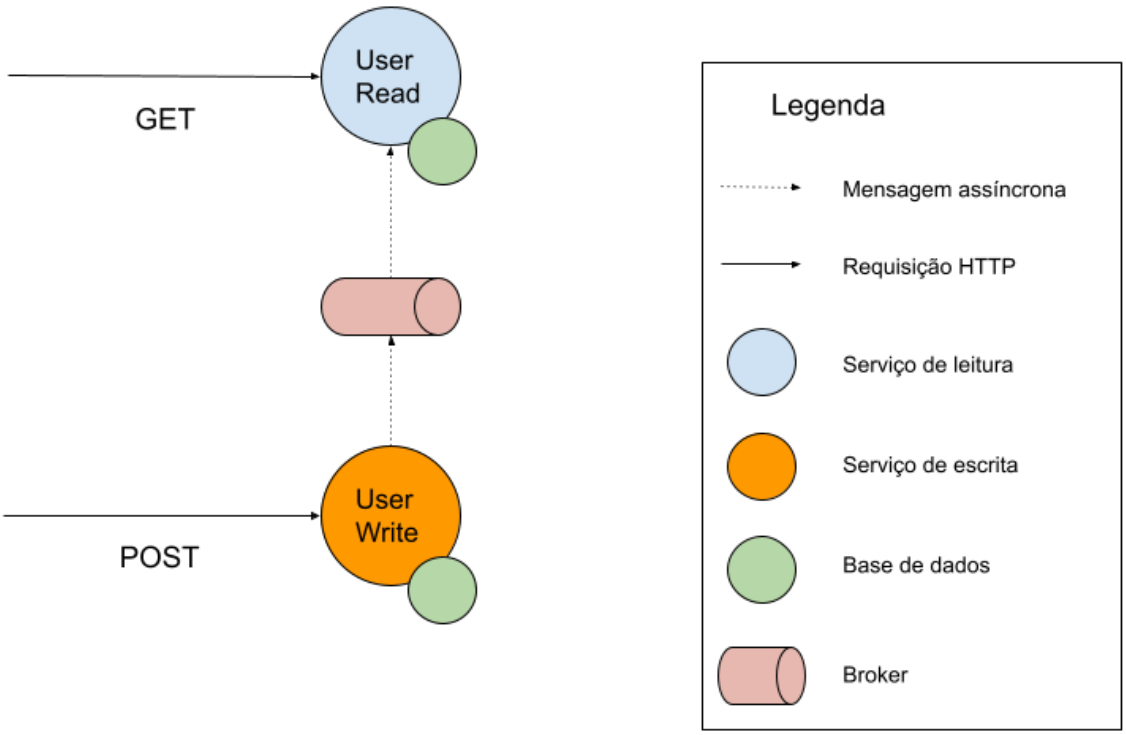


Figura 4.1: *Implementação do padrão CQRS*

Ambos os serviços são formados internamente por 4 componentes:

- **API**: Expõe *endpoints* para a realização de operações com os dados do serviço, tais como inserção, leitura e alteração de dados dos usuários;
- **Repository**: Um componente intermediário entre a API e a base dados, responsável por encapsular as operações com os dados da aplicação;
- **Consumer**: Responsável por consultar o *message broker* e consumir eventos de interesse do serviço;
- **Producer**: Responsável por publicar eventos ocorridos no serviço no *message broker*, para consumo de outros serviços interessados.

No fluxo implementado, quando um usuário é cadastrado na base de dados, por meio de uma requisição do tipo *POST* na *API* do serviço de escrita, um evento do tipo ***UserCreated*** contendo as informações do novo usuário é armazenado na base de dados do serviço de escrita. Após isso, esse evento é publicado em um *message broker*. O serviço de leitura é notificado sobre o novo evento na fila e o consome, processando-o e armazenando as informações do usuário de maneira otimizada para a leitura.

CQRS sem modelo de atores

Na versão do padrão CQRS sem o modelo de atores, os serviços foram implementados utilizando o **finch**, um *microframework* web para a linguagem de programação Scala.

O serviço **UserWrite** recebe, por meio de uma API REST, requisições de criação e alteração de informações de usuários, e persiste, seguindo o padrão **Event Sourcing**, em um banco de dados orientado a documentos.

Em seguida, os eventos são publicados no *message broker*. O **UserRead** consome os eventos do *broker*, emitidos pelo **UserWrite** e altera ou cria as entidades presentes no banco de dados de escrita.

As informações atualizadas podem ser consultadas realizando requisições HTTP, do tipo GET na API. É possível consultar todos os usuários presentes na base ou apenas um usuário específico, por meio do **id**. Para recuperar as informações de usuário na base de dados, o serviço de leitura realiza um processo de *replay* para reconstruir o estado do usuário ao mais atual disponível na base. Um trecho do código que realiza esse processo é mostrado no bloco de código 4.1

Programa 4.1 função readUser presente Repository do serviço UserRead.

```
1   def readUser(id: String): User = {
2     val userId = new ObjectId(id)
3     val userEvents = events.find(Filters.eq("entity_id", userId)).results
4     var user = User("", "")
5     userEvents.foreach(e =>
6       e.getString("event_type") match {
7         case "User Created" => user = User(e.get("event_data").get.asDocument.
              getString("name").getValue, e.get("event_data").get.asDocument.
              getString("tel").getValue)
8         case "User Name Updated" => user = User(e.get("event_data").get.
              asDocument.getString("name").getValue, user.tel)
9         case "User Tel Updated" => user = User(user.name, e.get("event_data").
              get.asDocument.getString("tel").getValue)
10        case "User Deleted" => user = null
11      }
12    )
13
14    user
15  }
```

CQRS com modelo de atores

A versão do padrão CQRS com o modelo de atores, implementa apenas um serviço, responsável tanto pela leitura quanto pela escrita. Para implementá-lo, foram utilizados os módulos **akka-http** para a construção dos *endpoints* da API, o **akka-cluster** para o gerenciamento do sistema de atores da aplicação, o **akka-serialization** para serialização tanto das requisições e das respostas dos *endpoints* quanto para as mensagens enviadas e recebidas por atores, e o **akka-persistence** para implementar o conjunto de atores responsável pela persistência e recuperação dos dados. A persistência nesse serviço adota o padrão Event Sourcing.

Contudo, esta implementação possui um componente extra, além dos 3 citados na implementação sem modelo de atores, que é o **Persistence**, que contém o ator que gerencia a persistência das informações dos usuários na aplicação.

Ao receber uma requisição em um dos *endpoints* da **API**, o **Repository** envia uma mensagem ao **Persistence** solicitando as informações necessárias para prosseguir com a operação. Caso um usuário, com as informações repassadas, ainda não exista na base de dados, um novo ator é criado para representar tal usuário na aplicação.

O bloco de código 4.2 ilustra os principais componentes do **Persistence**: O **commandHandler**, função que recebe as mensagens enviadas para o ator e armazena os eventos resultantes do processamento da mensagem, caso ocorram, e o **eventHandler**, que recebe esses eventos ocorridos no **commmandHandler** e atualiza o estado atual do ator em memória.

Programa 4.2 Trecho de cógido da classe UserRepository).

```
1   final case class State(id: String, name: String, tel: String) extends
        JsonSerializable {
2    def createUser(user: User) = copy(name = user.name, tel = user.tel)
3     def updateName(newName: String) = copy(name = newName)
4     def updateTel(newTel: String) = copy(tel = newTel)
5    def removeUser = copy(name = "", tel = "")
6   }
7
8    private def commandHandler(context: ActorContext[Command], state: State,
         command: Command): ReplyEffect[Event, State] = {
9     command match {
10      case GetUser(replyTo) =>
11       Effect
12         .reply(replyTo)(GetUserResponse(Option(User(state.name, state.tel))))
13      case CreateUser(user, replyTo) =>
14       Effect
15         .persist(UserCreated(user))
16         .thenReply(replyTo)(newUserState => newUserState.id)
17      case UpdateUserName(newName, replyTo) =>
18       Effect
19         .persist(UserNameUpdated(newName))
20         .thenReply(replyTo)(newUserState =>
21         GetUserResponse(Option(User(newUserState.name, newUserState.tel)))
22       )
23      case UpdateUserTel(newTel, replyTo) =>
24       Effect
25         .persist(UserTelUpdated(newTel))
26         .thenReply(replyTo)(newUserState =>
27         GetUserResponse(Option(User(newUserState.name, newUserState.tel)))
28       )
29      case DeleteUser(replyTo) =>
30       Effect.persist(UserDeleted).thenReply(replyTo)(newUserState =>
31         newUserState.id
32       )
33     }
34   }
35
36   private def eventHandler(state: State, event: Event): State = {
37    event match {
38      case UserCreated(user) =>
```

cont ⟶

```
    ⟶ cont
39        state.createUser(user)
40      case UserNameUpdated(name) =>
41        state.updateName(name)
42      case UserTelUpdated(tel) =>
43        state.updateTel(tel)
44      case UserDeleted =>
45        state.removeUser
46    }
47  }
```

Após o término da primeira versão desta implementação do CQRS, foi decidido não seguir adiante com a comparação entre as versões com e sem modelo de atores deste padrão, pois, como citado no capítulo 3, o caso de uso foi considerado simplório para os objetivos do trabalho. Seria necessário buscar outro caso de uso que contivesse um número maior de interações entre os serviços, para realçar as vantagens e desvantagens de cada implementação.

4.1.2 Implementação do Padrão Saga

A Saga escolhida para implementação neste projeto foi retirada de um caso de uso descrito por Richardson, 2018, em que uma empresa de entrega de comida possui um fluxo de aprovação de um pedido, ilustrado na figura 4.2. Nesse fluxo, estão envolvidos 4 microsserviços:

- **Order**: Responsável pelo gerenciamento de pedidos feitos na plataforma;
- **Consumer**: Encarregado de gerenciar os dados de usuários cadastrados na plataforma;
- **Kitchen**: Responsável por gerenciar os *tickets* criados nas cozinhas dos restaurantes cadastrados;
- **Accounting**: Responsável pelo gerenciamento e processamento das informações de pagamento dos pedidos.

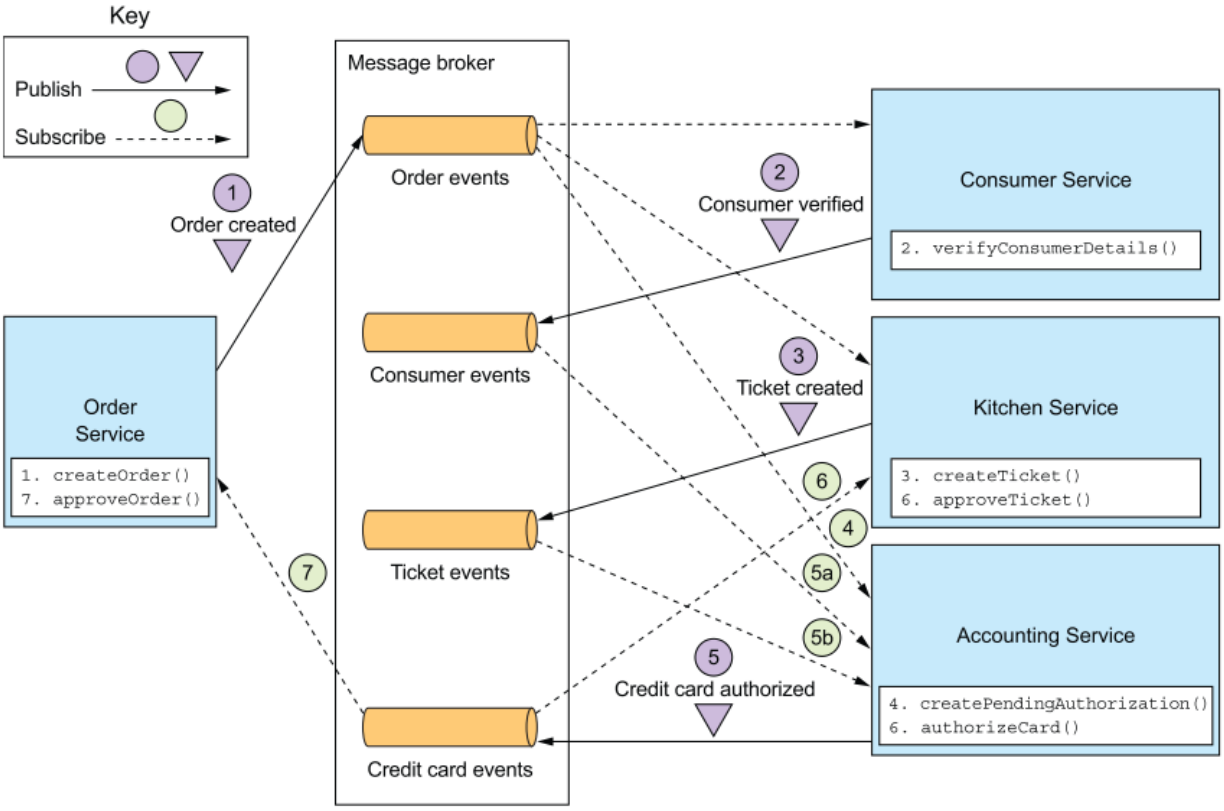


Figura 4.2: *Saga do fluxo de aprovação de um pedido em um sistema de entrega de comida [Richardson, 2018]*

No "caminho feliz", no qual todas as transações ocorrem da melhor maneira possível, isto é, não há falha em nenhuma das transações locais do fluxo, o seguinte cenário se estabelece:

1. O serviço Order cria um pedido com o estado APPROVAL_PENDING, e publica um evento dizendo que o pedido foi criado (OrderCreated);
2. O serviço Consumer consome o evento OrderCreated e verifica se o cliente atende aos requisitos para prosseguir com aquele pedido. Em caso positivo, o evento ConsumerVerifieid é publicado;
3. O serviço Kitchen consome o evento OrderCreated, valida o pedido, cria um ticket com estado CREATE_PENDING e publica um evento TicketCreated;
4. O Serviço Accounting consome o evento OrderCreated e cria uma autorização de cartão de crédito em estado PENDING;
5. O Serviço Accounting consome o evento TicketCreated e ConsumerVerified, efetua a cobrança no cartão de crédito do usuário e publica o evento CreditCardAuthorized;
6. O serviço Kitchen consome o evento CreditCardAuthorized, e muda o estado do Ticket para AWAITING_ACCEPTANCE;

7. O serviço Order consome o evento CreditCardAuthorized, e muda o estado do pedido para APPROVED e publica o evento OrderAproved.

A figura 4.3 ilustra os estados possíveis após cada transação realizada na Saga da figura 4.2, com as possíveis ações de compensação, caso alguma das transações locais falhe.

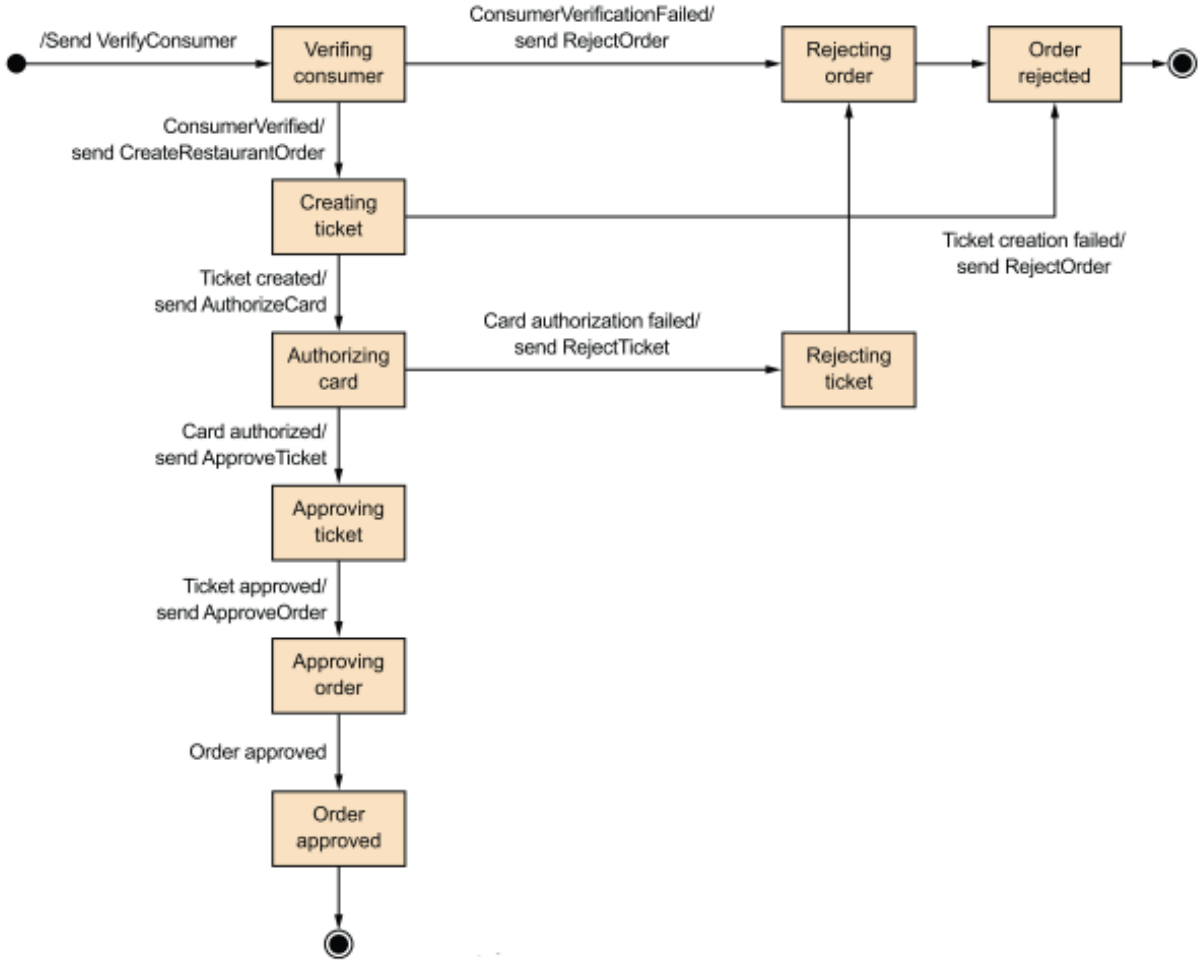


Figura 4.3: *Máquina de estados da Saga de criação de um pedido*

Todos os serviços são formados internamente por 5 componentes:

- **API**: Expõe endpoints para a realização de operações com os dados do serviço, tais como inserção, leitura e alteração de dados dos usuários;
- **Repository**: Um componente intermediário entre a API e a base dados, é responsável por encapsular as operações com os dados da aplicação;
- **Consumer**: Responsável por consultar o *message broker* e consumir eventos de interesse do serviço;
- **Producer**: Responsável por publicar eventos ocorridos no serviço no *message broker* para consumo de outros serviços interessados;

- **Persistence**: Contém o ator de persistência, responsável por armazenar e ler na base dados o estado da entidade.

Saga sem modelo de atores

A versão do padrão Saga sem o modelo de atores, os serviços foram implementados utilizando novamente o **finch**, adotado na implementação sem modelo de atores do padrão CQRS discutida na seção 4.1.1.

Ao chegar uma requisição no *endpoint* do serviço, a API solicita ao Repository realizar as alterações necessárias na base de dados. Ao final desse processo, é publicado um evento contendo as informações sobre a modificação realizada. Caso a requisição feita necessite do estado atual de uma entidade na base de dados, um replay dos eventos ocorridos a ela armazenados é realizado até obtê-lo.

O bloco de código 4.3 ilustra a classe OrderRepository, que implementa o componente Repository no serviço Order.

Programa 4.3 Classe OrderRepository.

```
1   class OrderRepository {
2     val mongoClient = MongoClient()
3
4     val orders = mongoClient.getDatabase("order").getCollection("order-sync")
5
6
7     def getOrderById(id: String): Option[Order] = {
8       val orderId = new ObjectId(id)
9       val orderEvents = orders.find(Filters.eq("entity_id", consumerId)).results
            ()
10      var order = Option.empty[Order]
11      orderEvents.foreach(event => {
12        val eventData = event.get("event_data")
13        event.get("event_type") match {
14          case "OrderCreated" => order = Some(Order(orderId, eventData.get("
                consumer_id"), "PENDING"))
15          case "OrderApproved" => order = Some(order.get.copy(status = "APPROVED")
                )
16          case "ConsumerDeleted" => order = Option.empty[Order]
17        }
18      })
19      order
20    }
21
22    def approveOrder(orderId: String) = {
23      val eventId = new ObjectId()
24      val eventDoc = Document(
25        "_id" -> eventId,
26        "event_type" -> "OrderApproved",
27        "entity_id" -> orderId,
28        "event_data" -> Document(
29          "status" -> "APPROVED"
```

cont ⟶

cont ⟶

```
30        )
31      )
32      orders.insertOne(eventDoc).printHeadResult()
33    }
34
35    def createOrder(consumerId: String): String = {
36      val eventId = new ObjectId()
37      val orderId = new ObjectId()
38      val orderDoc = Document(
39        "_id" -> eventId,
40        "event_type" -> "OrderCreated",
41        "entity_id" -> orderId,
42        "event_data" -> Document(
43          "consumer_id" -> consumerId,
44          "status" -> "PENDING"
45        )
46      )
47      orders.insertOne(orderDoc).printHeadResult()
48      writeToKafka("order-created", orderDoc.toJson())
49      orderId.toString
50    }
51  }
```

Saga com modelo de atores

A versão do padrão Saga com o modelo de atores, possui apenas um serviço, responsável tanto pela leitura quanto pela escrita. Para implementá-lo, foram utilizados os módulos **akka-http** para a construção dos *endpoints* da API, o **akka-cluster** para o gerenciamento do sistema de atores da aplicação, o **akka-serialization** para serialização tanto das requisições e das respostas dos *endpoints* quanto para as mensagens enviadas e recebidas por atores, e o **akka-persistence** implementar o conjunto de atores responsável pela persistência e recuperação dos dados. A persistência neste serviço adota o padrão Event Sourcing.

A API é o componente responsável por expor uma *API* REST permitindo operações básicas de acesso aos dados, como inserção e retorno. O Repository é o mediador da relação com a camada de persistência, tanto para as chamadas vindas da *API* quanto para realizar as alterações no estado interno do serviço resultantes dos eventos externos consumidos pelo **Consumer**. A comunicação com o **Persistence** se estabelece adotando o padrão de comunicação assíncrona **Ask**, no qual uma mensagem é enviada para um ator e o emissor aguarda a resposta.

O processamento de mensagens recebidas pelo ator de persistência se inicia no **command handler**. Ele é uma função que recebe como parâmetro o estado atual do ator, obtido por meio do *replay* de eventos ocorridos com o ator anteriormente, considerando um estado inicial definido na inicialização do ator. Caso esse processo de recuperação tenha ocorrido no mesmo ator, em um espaço curto de tempo, o *replay* de eventos não é realizado, pois o Akka armazena o estado durante um intervalo configurável de tempo. Essa característica é denominada ***passivation***. Caso o processamento da mensagem acarrete

alguma alteração no estado interno do serviço, um evento descrevendo essa mudança é persistido na base de dados, assim como descreve o padrão **Event Sourcing**.

No sucesso, em persistir esse evento, esse *handler* envia o evento para o **event handler**, uma função que analisa esse evento e altera o estado interno, conforme a regra de negócio implementada nele. O novo estado permanece em memória até que o ***passivation*** expire.

O trecho de código 4.4 ilustra os principais componentes do Persistence (do serviço Order): O **commandHandler**, função que recebe as mensagens enviadas para o ator e armazena os eventos resultantes do processamento da mensagem, caso ocorram, e o **eventHandler**, que recebe esses eventos ocorridos no commmandHandler e atualiza o estado atual do ator em memória.

Programa 4.4 Código do ator de persistência do serviço Order.

```
1   object OrderPersistence {
2
3    final case class State(id: String, consumerId: String, orderState: OrderState
         ) extends JsonSerializable {
4     def createOrder(consumerId: String): State = copy(consumerId = consumerId)
5     def approveOrder(): State = copy(orderState = OrderState.APPROVED)
6     def rejectOrder(): State = copy(orderState = OrderState.REJECTED)
7     def removeOrder(): State = copy(orderState = null)
8    }
9
10   object State {
11    def empty(orderId: String): State = State(orderId, "", OrderState.PENDING)
12   }
13
14   val EntityKey: EntityTypeKey[Command] =
15    EntityTypeKey[Command]("Order")
16
17   private def commandHandler(context: ActorContext[Command], state: State,
         command: Command): ReplyEffect[Event, State] = {
18    implicit val mat: Materializer = Materializer(context.system)
19    command match {
20     case GetOrder(replyTo) =>
21      Effect
22       .reply(replyTo)(GetOrderResponse(Option(Order(state.consumerId, state.
             orderState))))
23     case CreateOrder(order, replyTo) =>
24      Effect
25       .persist(OrderCreated(order))
26       .thenReply(replyTo)(newUserState => newUserState.id)
27     case ApproveOrder(replyTo) =>
28      Effect
29       .persist(OrderApproved)
30       .thenReply(replyTo)(newState => newState.id)
31     case DeleteOrder(replyTo) =>
32      Effect.persist(OrderDeleted).thenReply(replyTo)(newOrderState =>
33       newOrderState.id
34      )
```

cont ⟶

```
    ⟶ cont
35      }
36    }
37
38    private def eventHandler(context: ActorContext[_], state: State, event:
          Event): State = {
39      implicit val mat: Materializer = Materializer(context.system)
40      event match {
41        case OrderCreated(order) =>
42          val f = state.createOrder(order.consumerId)
43          OrderProducer.publish("order-created", OrderCreatedToKafka("
                OrderCreated", f.id, order.consumerId))
44          f
45        case OrderApproved => state.approveOrder()
46        case OrderDeleted =>
47          state.removeOrder()
48      }
49    }
50
51    def initSharding(system: ActorSystem[_]): Unit = {
52      val behaviorFactory: EntityContext[Command] => Behavior[Command] = {
53        entityContext =>
54          OrderPersistence(entityContext.entityId)
55      }
56      ClusterSharding(system).init(Entity(EntityKey)(behaviorFactory))
57    }
58
59
60    def apply(orderId: String): Behavior[Command] = {
61      Behaviors.setup { context =>
62        EventSourcedBehavior[Command, Event, State](
63          persistenceId = PersistenceId(EntityKey.name, orderId),
64          emptyState = State.empty(orderId),
65          commandHandler = (state, command) => commandHandler(context, state,
                command),
66          eventHandler = (state, event) => eventHandler(context, state, event))
67          .withTagger(_ => Set("orders"))
68      }
69
70    }
71
72  }
```

4.2 Discussão Analítica

Durante as implementações realizadas neste trabalho, alguns pontos são interessantes a serem considerados.

A configuração inicial dos serviços desenvolvidos sem utilizar Akka ocorreu de forma simples, principalmente devido ao uso do *finch* fornecer uma estrutura base de organização de código e rotas para a API REST, restando apenas configurar quais *message brokers* seriam escutados pelo serviço e a base de dados a ser utilizada. Enquanto na versão com modelo

de atores, além das configurações citadas anteriormente, foi necessário configurar o akka-cluster, módulo do akka responsável por gerenciar o sistema de atores coordenador da persistência.

A serialização é uma questão que também difere nas duas versões. Ao desenvolver uma API sem modelo de atores, a serialização, processo de conversão de objetos para vetores de *bytes*, é um ponto de atenção somente na comunicação externa, ou seja, na entrada de dados, onde são recebidos os parâmetros, e no envio da resposta para uma requisição. Adotando o Akka, a serialização ocorre também ao enviar alguma mensagem ao ator, pois como os atores possuem transparência de localização, toda a mensagem enviada a ele deve ser serializável. Programadores menos experientes com este conceito, principalmente usuários de *frameworks* que não explicitam esse tipo de operação, podem sentir dificuldades em um primeiro contato com o Akka.

Ao adotar o Event Sourcing, sempre que alguma lógica de negócio necessita do estado atual de uma entidade, se faz necessário aplicar todas as mudanças promovidas pelos eventos que ocorreram anteriormente com a entidade. Uma possível otimização para esse processo é armazenar o estado da entidade em uma base de dados separada de tempos em tempos, ou a partir de um determinado número de eventos processados. Essa otimização é conhecida como ***snapshot***. Essa otimização é facilmente configurável no Akka, bastando apenas adicionar no arquivo de configuração qual base de dados onde serão armazenados os *snapshots* e quais regras para que esse processo seja inciado.

4.3 Percepções de Desenvolvedores Sobre o Modelo de Atores, Programação Reativa e Padrões Arquiteturais de Microsserviços

Esta seção detalha um questionário elaborado com o objetivo de coletar as percepções de desenvolvedores de *software* sobre a implementação de padrões de microsserviços utilizando modelo de atores e programação reativa.

Nesse questionário, detalhado no Apêndice A, os participantes informaram o nível de conhecimento sobre os padrões investigados e analisaram e a compararam as implementações do padrão Saga com e sem o modelo de atores.

O questionário foi disponibilizado por meio de um link para a plataforma *Google Forms*, e permaneceu aberto para respostas durante uma semana. O grupo-alvo da divulgação foram universitários de áreas da computação e profissionais da área de desenvolvimento. No final do período, 16 desenvolvedores responderam o formulário.

As questões foram divididas em blocos, e as respostas obtidas serão apresentadas nas seguintes subseções: **Perfil dos participantes** e **Comparação de código**.

4.3.1 Perfil dos Participantes

Nesse bloco, os participantes foram questionados sobre o tempo de experiência na área de desenvolvimento de software, o nível de experiência com a linguagem de programação

Scala e como eles avaliavam o nível de conhecimento sobre conceitos utilizados neste trabalho, tais como programação reativa, modelo de atores, além do estilo arquitetural de microsserviços e os padrões arquiteturais Event Sourcing e Saga.

Sobre o nível de experiência, a maioria dos entrevistados possui menos de 5 anos de experiência na área de desenvolvimento, como mostra o gráfico da figura 4.4

Para identificar o nível de experiência dos participantes com a linguagem Scala, programação reativa, modelo de atores, microsserviços e os padrões Saga e Event sourcing, foi utilizada a seguinte escala, que variada de inexperiente a especialista.

- **Inexperiente** - Não conhece e nem tem experiência teórica ou prática.
- **Novato** - Conhece a teoria e os princípios e os aplica em contextos simples.
- **Iniciante avançado** - Tem experiência prática em cenários reais.
- **Competente** - Tem experiência em diferentes contextos e cenários reais.
- **Proficiente** - Pode tomar decisões de maneira consciente e com alto grau de assertividade.
- **Especialista** - Pode tomar decisões de forma assertiva e sem dificuldades significativas.

Sobre o nível de experiência com a linguagem de programação Scala, apenas um participante se autodeclarou como proficiente e nenhum como especialista, como demonstrado na figura 4.5.

Sobre o nível de conhecimento dos participantes em programação reativa, a maioria dos participantes se considerou novata, enquanto que no modelo de atores houve um número alto de inexperientes e novatos, como ilustra o gráfico da figura 4.6.

Sobre o nível de experiência com o estilo arquitetural de microsserviços, apenas um participante se autodeclarou como proficiente e nenhum como especialista, como demonstrado na figura 4.7

Sobre o nível de conhecimento dos participantes a respeito dos padrões arquiteturais Event Sourcing e Saga, a maioria se considerou inexperiente com o padrão Saga, enquanto que a maioria dos desenvolvedores se considerou novata a respeito do padrão Event Sourcing, como ilustra o gráfico da figura 4.8.

Após compreender o perfil dos participantes, no bloco de perguntas seguintes, buscou-se a percepção dos desenvolvedores ao analisar as versões de implementação do padrão Saga com e sem o modelo de atores. Os resultados coletados são apresentados na seção 4.3.2.

4.3.2 Percepção dos Desenvolvedores

Neste bloco de perguntas, foram exibidos aos desenvolvedores trechos de código responsável pelo gerenciamento dos dados armazenados por uma entidade, do microsserviço responsável pelos pedidos (Order), em duas versões: a primeira adota modelo de atores, por

meio do Akka, ilustrada no bloco de código 4.5. Já a segunda segue uma maneira síncrona tradicional, ilustrada no bloco de código 4.6. Ambos escritos na linguagem Scala.

Programa 4.5 Trecho de código do ator de persistência do serviço Order.

```
1   private def commandHandler(context: ActorContext[Command], state: State,
        command: Command): ReplyEffect[Event, State] = {
2     implicit val mat: Materializer = Materializer(context.system)
3     command match {
4       case GetOrder(replyTo) =>
5         Effect
6           .reply(replyTo)(GetOrderResponse(Option(Order(state.consumerId, state.
                orderState))))
7       case CreateOrder(order, replyTo) =>
8         Effect
9           .persist(OrderCreated(order))
10          .thenReply(replyTo)(newUserState => newUserState.id)
11      case ApproveOrder(replyTo) =>
12        Effect
13          .persist(OrderApproved)
14          .thenReply(replyTo)(newState => newState.id)
15      case DeleteOrder(replyTo) =>
16        Effect.persist(OrderDeleted).thenReply(replyTo)(newOrderState =>
17          newOrderState.id
18        )
19    }
20  }
21
22   private def eventHandler(context: ActorContext[_], state: State, event:
        Event): State = {
23    implicit val mat: Materializer = Materializer(context.system)
24    event match {
25      case OrderCreated(order) =>
26        val f = state.createOrder(order.consumerId)
27        OrderProducer.publish("order-created", OrderCreatedToKafka("
              OrderCreated", f.id, order.consumerId))
28        f
29      case OrderApproved => state.approveOrder()
30      case OrderDeleted =>
31        state.removeOrder()
32    }
33  }
```

Programa 4.6 Trecho de código da classe OrderRepository do serviço Order.

```
1   def getOrderById(id: String): Option[Order] = {
2     val orderId = new ObjectId(id)
3     val orderEvents = orders.find(Filters.eq("entity_id", consumerId)).results
          ()
4     var order = Option.empty[Order]
5     orderEvents.foreach(event => {
6       val eventData = event.get("event_data")
```

cont ⟶

⟶ *cont*

```
7       event.get("event_type") match {
8         case "OrderCreated" => order = Some(Order(orderId, eventData.get("
              consumer_id"), "PENDING"))
9         case "OrderApproved" => order = Some(order.get.copy(status = "APPROVED")
              )
10        case "ConsumerDeleted" => order = Option.empty[Order]
11      }
12    })
13    order
14  }
15
16  def approveOrder(orderId: String) = {
17    val eventId = new ObjectId()
18    val eventDoc = Document(
19      "_id" -> eventId,
20      "event_type" -> "OrderApproved",
21      "entity_id" -> orderId,
22      "event_data" -> Document(
23        "status" -> "APPROVED"
24      )
25    )
26    orders.insertOne(eventDoc).printHeadResult()
27  }
28
29  def createOrder(consumerId: String): String = {
30    val eventId = new ObjectId()
31    val orderId = new ObjectId()
32    val orderDoc = Document(
33      "_id" -> eventId,
34      "event_type" -> "OrderCreated",
35      "entity_id" -> orderId,
36      "event_data" -> Document(
37        "consumer_id" -> consumerId,
38        "status" -> "PENDING"
39      )
40    )
41    orders.insertOne(orderDoc).printHeadResult()
42    writeToKafka("order-created", orderDoc.toJson())
43    orderId.toString
44  }
```

Em seguida, foram coletadas as opiniões dos participantes para cada trecho de código em três aspectos: **legibilidade**, **organização** e **concisão**. A legibilidade se refere à facilidade de leitura e compreensão do código. A organização é entendida como a separação e modularização do código. Já a concisão é entendida como a expressividade do código. A escala utilizada em cada resposta possui níveis de 1 a 5, no qual o nível 1 (muito bom) indica que o trecho atende muito bem ao aspecto e o nível 5 (muito ruim) indica que o código definitivamente não atende ao aspecto.

Em relação à legibilidade, conforme ilustrado na figura 4.9, 9 dos 16 participantes indicaram que a legibilidade do código sem o modelo de atores é boa (7 desenvolvedores)

ou muito boa (2 desenvolvedores). Ao analisar a legibilidade do código com o modelo de atores, apenas 5 participantes sinalizaram que a legibilidade é boa ou muito boa, os 11 restantes consideraram a legibilidade moderada (7 desenvolvedores) ou ruim (4 desenvolvedores).

Em relação à organização, conforme ilustrado na figura 4.11, 11 dos 16 participantes indicaram que a organização do código sem o modelo de atores é boa (9 desenvolvedores) ou muito boa (2 desenvolvedores). Ao analisar a organização do código com o modelo de atores, 10 participantes sinalizaram que a organização é boa ou muito boa, 6 restantes consideraram a organização moderada (5 desenvolvedores) ou ruim (1 desenvolvedor).

Em relação à concisão, conforme ilustrado na figura 4.10 8 dos 16 participantes indicaram que a concisão do código sem o modelo de atores é boa (7 desenvolvedores) ou muito boa (2 desenvolvedores). Ao analisar a concisão do código com o modelo de atores, 11 participantes sinalizaram que a concisão é boa ou muito boa, os 5 restantes consideraram a legibilidade moderada (4 desenvolvedores) ou ruim (1 desenvolvedor).

Análise Qualitativa dos Resultados

Analisando os dados obtidos na pesquisa, percebe-se que uma distribuição no nível de conhecimento na linguagem em Scala. Como ambos os trechos de código analisados pelos participantes são escritos nessa linguagem, um recorte considerando o nível de conhecimento na linguagem é algo a ser estudado. Neste recorte, 8 participantes se consideraram experientes com a linguagem Scala enquanto o restante (8 pessoas) indicaram serem inexperientes na linguagem.

Uma hipótese que pode explicar esses resultados é o nível de experiência na área de desenvolvimento de software dos participantes, e, portanto, um maior contato com linguagens que possuem conceitos e sintaxe semelhantes à linguagem Scala, como, por exemplo, o encadeamento de métodos, presente nos dois códigos analisados, que diversas linguagens orientadas a objetos passaram a implementar em versões mais recentes, como Java, Kotlin, entre outras. Entretanto, é importante salientar que a distribuição das respostas são semelhantes, indicando que este o nível de conhecimento em Scala possui baixo impacto na análise.

Outro recorte interessante é analisar como o nível de conhecimento em modelo de atores influencia a percepção sobre a concisão da implementação com modelo de atores. Neste caso, dividimos os participantes em dois grupos: Aqueles que possuem alguma experiência prévia com modelo de atores e aqueles que não conhecem o modelo de atores, como ilustrado na figura 4.13

Uma possível explicação para que ambos os grupos considerem boa a concisão da implementação com modelo de atores é a boa expressividade que a ferramenta Akka fornece para implementar sistemas que adotam esse modelo, mesmo tendo uma certa curva de aprendizado.

Com isso, pode-se concluir que as duas implementações foram consideradas bem equivalentes em expressividade pelos entrevistados. Mesmo os 8 participantes inexperientes com Scala e os 9 inexperientes com o conceito de modelo de atores conseguiram compreender de

maneira satisfatória a implementação com modelo de atores feito utilizando a plataforma Akka, indicando uma alta concisão da ferramenta para códigos dessa natureza

4.3.3 Ameaças à Validade

Análises qualitativas de dados estão sujeitas a vieses. No contexto deste trabalho, o perfil dos entrevistados é um fator a ser considerado, pois a distribuição do nível de experiência entre os participantes não é homogêneo, apesar de desejável por ser mais próximo à realidade. Para reduzir um possível viés na análise de resultados, uma discussão junto a um grupo de pesquisadores da área de sistemas do IME-USP (Instituto de Matemática e Estatística da Universidade de São Paulo) foi realizada analisando os dados coletados.

O conjunto pequeno de respostas obtidos também é um fator que ameaça à validade dos resultados apresentados nesta seção. Por isso, é encorajado novas pesquisas sejam feitas no tema deste trabalho de modo a validar e expandir os resultados obtidos.

4.4 Discussão

Durante o desenvolvimento das duas implementações do padrão Saga, foi perceptível que o modelo de atores, utilizado por meio da plataforma Akka, apesar da dificuldade inicial de configuração, que pode ser uma barreira de entrada para iniciantes, proveu boas estruturas para lidar com a persistência que adota o padrão Event Sourcing.

Os resultados obtidos na pesquisa feita com desenvolvedores indicou que mesmo sem experiência prévia com os conceitos de modelo de atores e programação reativa, os participantes do estudo conseguiram compreender de forma satisfatória a implementação do padrão Saga com modelo de atores utilizando a plataforma Akka, demonstrando a alta expressividade da ferramenta, apesar da sua alta curva de aprendizado.

Com os resultados obtidos durante a implementação dos padrões e na pesquisa feita com desenvolvedores, pode-se concluir que a ferramenta Akka se mostrou benéfica na implementação de aplicações que adotam o modelo de atores

Figura 4.4: *Anos de experiência*

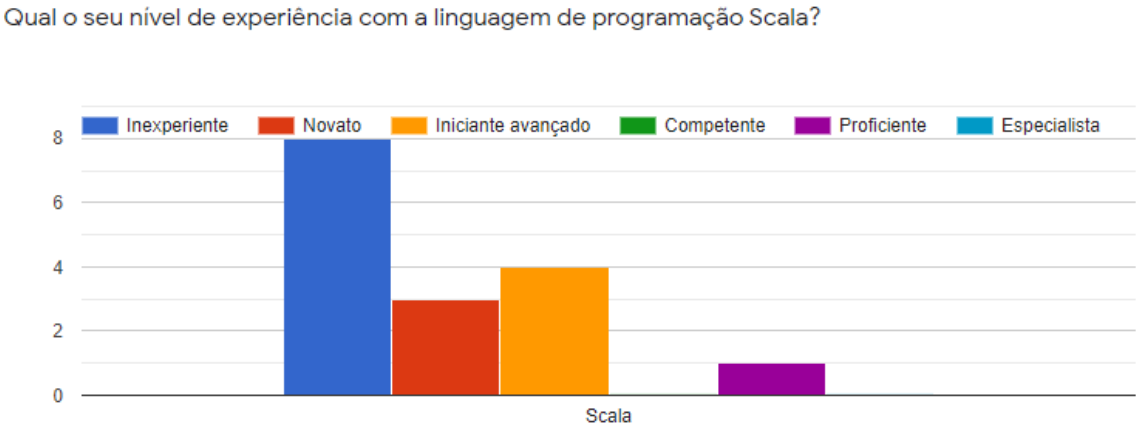


Figura 4.5: *Nível de experiência em Scala dos participantes*

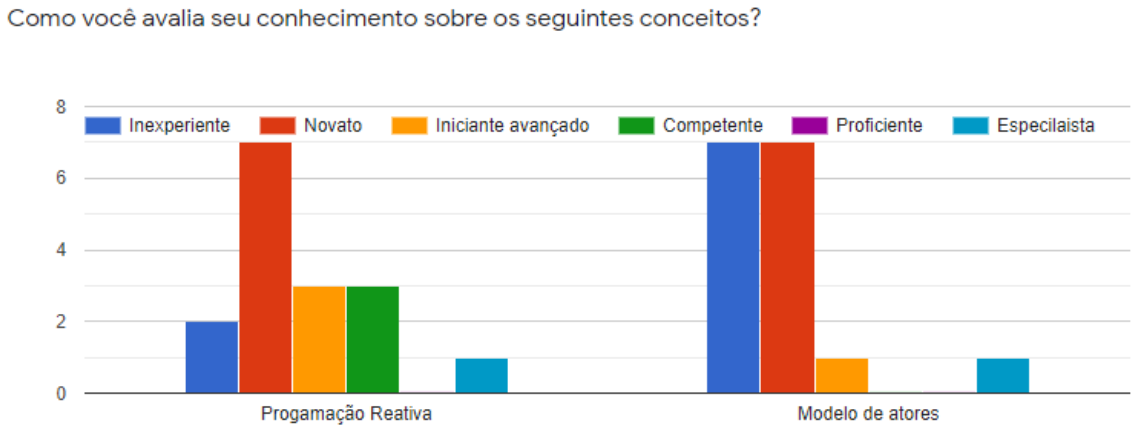


Figura 4.6: *Nível de conhecimento em programação reativa e modelo de atores dos participantes*

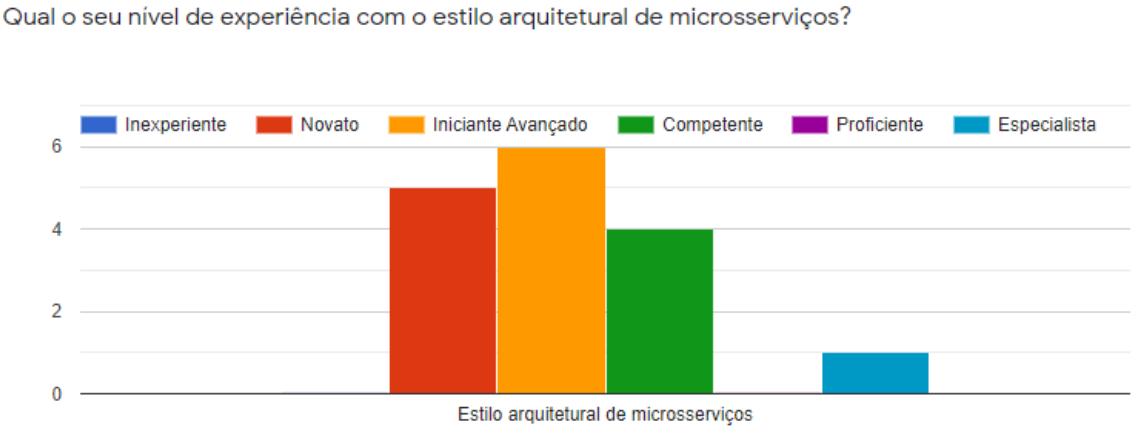


Figura 4.7: *Nível de conhecimento dos participantes sobre o estilo arquitetural de microsserviços*

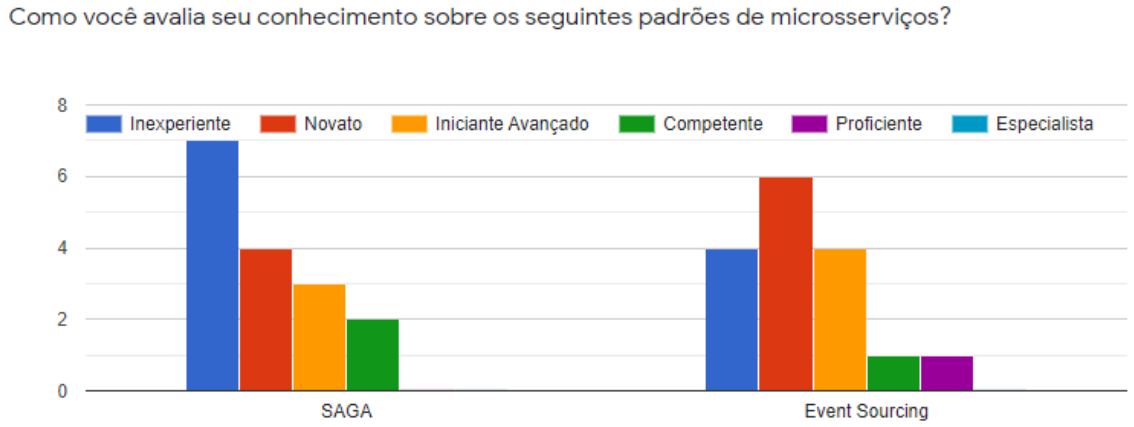
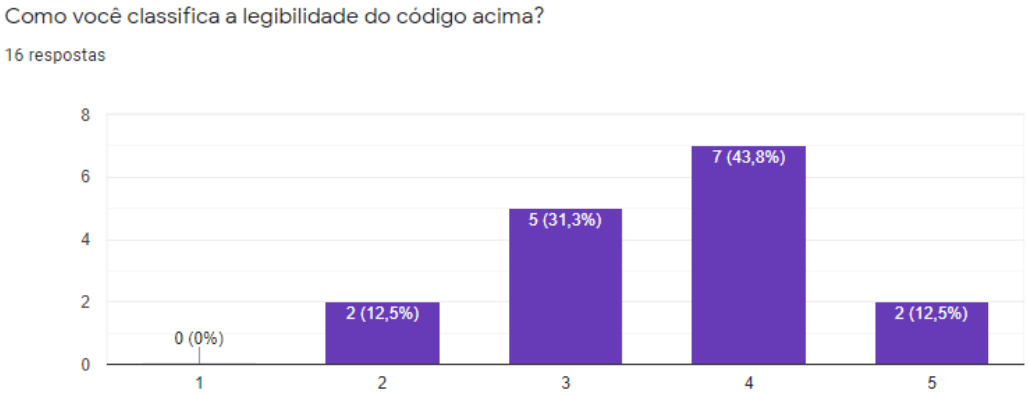
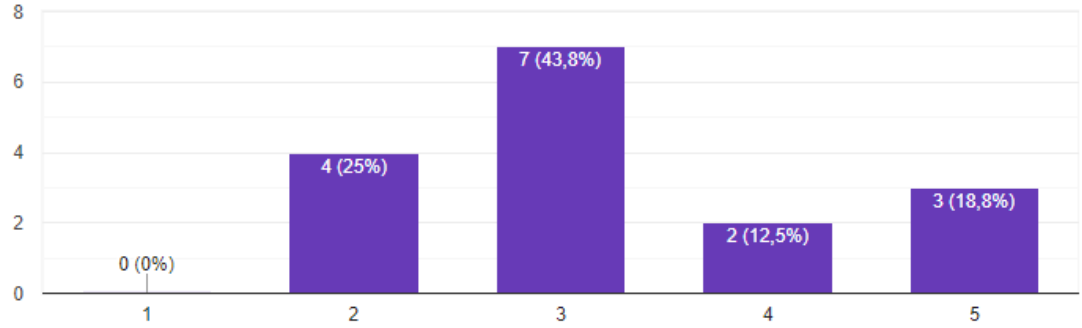


Figura 4.8: *Nível de conhecimento dos participantes sobre o padrão Saga, a esquerda, e Event Sourcing, a direita*

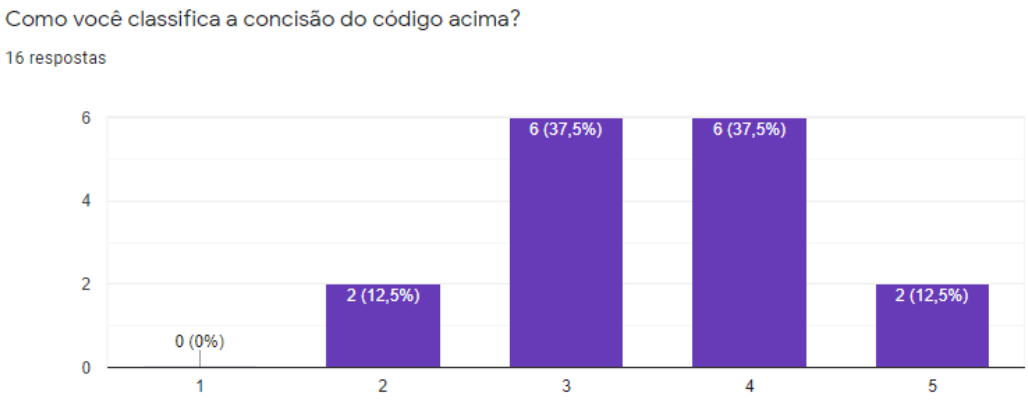


(a) *Implementação sem modelo de atores*

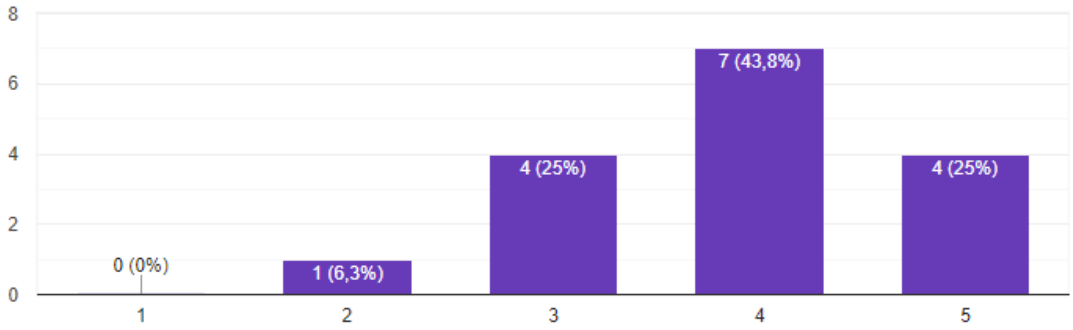


(b) *Implementação com modelo de atores*

Figura 4.9: *Como você classifica a legibilidade do código acima?*

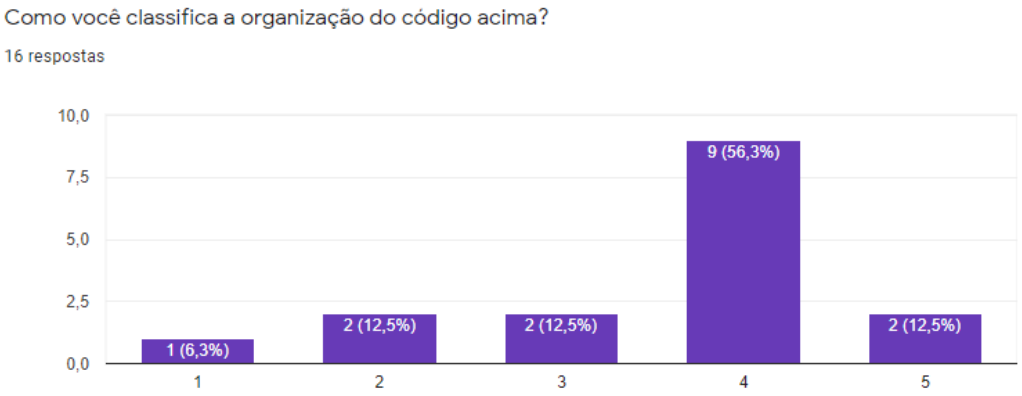


(a) *Implementação sem modelo de atores*

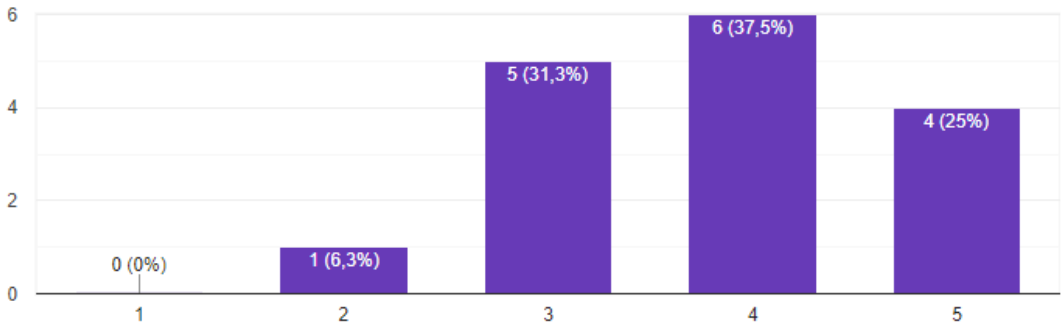


(b) *Implementação com modelo de atores*

Figura 4.10: *Como você classifica a concisão do código acima?*

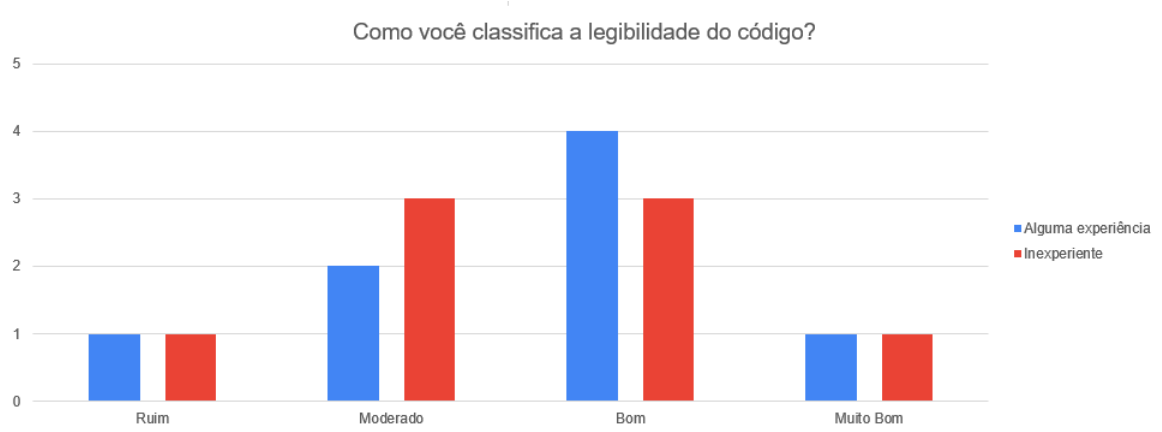


(a) *Implementação sem modelo de atores*

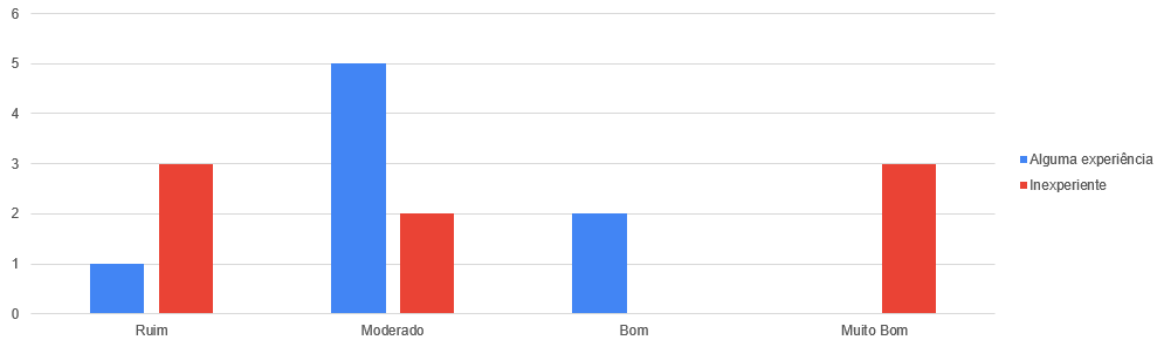


(b) *Implementação com modelo de atores*

Figura 4.11: *Como você classifica a organização do código acima?*



(a) *Implementação sem modelo de atores*



(b) *Implementação com modelo de atores*

Figura 4.12: *Como você classifica a legibilidade do código acima?*

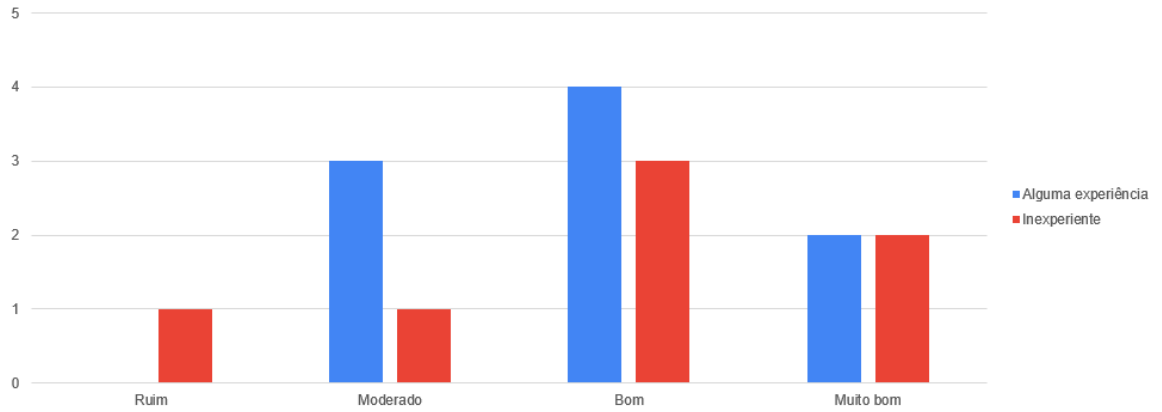


Figura 4.13: *Concisão da implementação com modelo de atores considerando nível de conhecimento dos participantes do modelo*

Capítulo 5

Considerações Finais

O objetivo deste trabalho é identificar as vantagens e desvantagens na adoção do modelo de atores na implementação de arquiteturas reativas de microsserviços. Para cumprir com objetivo proposto, a estratégia adotada foi comparar duas implementações de padrões arquiteturais de microsserviços. Para isso, foram implementadas duas versões do padrão Saga, sendo uma versão baseada no modelo de atores e a outra não. Além disso, por meio de um questionário foram coletadas as percepções de desenvolvedores de *software* sobre a implementação de padrões de microsserviços utilizando modelo de atores e programação reativa, onde os participantes informaram o nível de conhecimento sobre os padrões investigados analisaram e a compararam as implementações do padrão Saga com e sem o modelo de atores.

Durante o desenvolvimento das duas implementações do padrão Saga, foi perceptível que o modelo de atores, utilizado por meio da plataforma Akka, apesar da dificuldade inicial de configuração, que pode ser uma barreira de entrada para iniciantes, proveu boas estruturas para lidar com a persistência que adota o padrão Event Sourcing, presente na implementação escolhida do padrão arquitetural de microsserviços Saga. A facilidade de configurar comportamentos para otimizar a recuperação de estado em memória de entidades, o *snapshot*, também destaca a implementação utilizando modelo de atores quando comparada com a implementação sem o uso do modelo.

Os resultados obtidos na pesquisa feita com desenvolvedores indicou que mesmo sem experiência prévia com os conceitos de modelo de atores e programação reativa, os participantes do estudo conseguiram compreender de forma satisfatória a implementação do padrão Saga com modelo de atores utilizando a plataforma Akka, demonstrando a alta expressividade da ferramenta, apesar da sua alta curva de aprendizado.

Com os resultados obtidos durante a implementação dos padrões e na pesquisa feita com desenvolvedores, pode-se concluir que a ferramenta Akka se mostrou benéfica na implementação de aplicações que adotam o modelo de atores.

Dado o baixo número de respostas obtido na pesquisa feita com desenvolvedores, outras pesquisas no tema deste trabalho são incentivadas para validar e expandir os resultados alcançados. Além disso, experimentos envolvendo *benchmarks* para avaliar aspectos práticos de desempenho na execução das duas implementações também são

incentivados

Apêndice A

Questionário da coleta dados sobre a percepção dos desenvolvedores sobre a implementação de padrões de microsserviços utilizando programação reativa

Trade-offs da implementação de arquiteturas de microsserviços utilizando o modelo de atores

Este estudo está sendo conduzido por Wander Douglas Andrade de Souza (Universidade de São Paulo), Alfredo Goldman (Universidade de São Paulo), João Francisco Lino Daniel (Universidade de São Paulo), Renato Cordeiro Ferreira (Universidade de São Paulo), Thatiane de Oliveira Rosa (Universidade de São Paulo)

Este questionário visa coletar dados sobre a percepção dos desenvolvedores sobre a implementação de padrões de microsserviços utilizando programação reativa

Os resultados serão utilizados apenas em pesquisas acadêmicas, sem finalidade comercial.

O tempo estimado para completar a pesquisa é de 10 minutos.

Se você tiver alguma dúvida, sinta-se à vontade para entrar em contato conosco:
Wander Douglas Andrade de Souza <wander.souza@usp.br>
Alfredo Goldmam <gold@ime.usp.br>
João Daniel <joaofran@ime.usp.br>
Renato Ferreira <renatocf@ime.usp.br>
Thatiane Rosa <thatiane@usp.br>

*Obrigatório

Declaração de Confidencialidade

Podemos garantir que todos os dados coletados serão usados apenas para apoiar esta pesquisa científica. Todos os dados coletados são anônimos, além disso, apenas os pesquisadores deste estudo terão acesso aos dados brutos.

Ao concordar em colaborar com esta pesquisa, o participante permite que os pesquisadores utilizem os dados anônimos conforme descrito a seguir. O participante entende que:

1. A participação neste estudo não envolverá nenhum risco físico e risco emocional mínimo;

2. A participação neste estudo é totalmente voluntária, e você tem a liberdade de cancelar a pesquisa a qualquer momento, sem qualquer penalidade;

3. Você pode entrar em contato com os pesquisadores a qualquer momento, se tiver alguma dúvida sobre o estudo;

4. Os dados coletados são confidenciais e que quaisquer objetos de estudo serão anônimos se incluídos em publicações em periódicos, conferências ou posts de blogs;

5. Os pesquisadores manterão os dados coletados perpetuamente e poderão utilizá-los para futuros estudos científicos;

6. Esta pesquisa usa o pacote de aplicativos do Google Docs, portanto, a coleta e o uso de informações pelo Google estão sujeitos à Política de Privacidade do Google (https://www.google.co.uk/policies/privacy/).

Você concorda em participar desta pesquisa? * *

- ◯ Sim
- ◯ Não

Conhecimentos Prévios

Quanto tempo, em anos, de experiência você possui na área de desenvolvimento de software? *

Sua resposta

Nas perguntas a seguir, considere a seguinte definição:

Inexperiente - não conhece e nem tem experiência teórica ou prática
Novato - conhece a teoria e os princípios e os aplica em contextos simples
Iniciante avançado - tem experiência prática em cenários reais
Competente - tem experiência em diferentes contextos e cenários reais
Proficiente - pode tomar decisões de maneira consciente e com alto grau de assertividade
Especialista - pode tomar decisões de forma assertiva e sem dificuldades significativas

Qual o seu nível de experiência com a linguagem de programação Scala? *

	Inexperiente	Novato	Iniciante avançado	Competente	Proficiente	Especialista
Scala	○	○	○	○	○	○

Como você avalia seu conhecimento sobre os seguintes conceitos? *

A definição dos conceitos pode ser encontrada nos seguintes links: Programação Reativa (https://en.wikipedia.org/wiki/Reactive_programming), Modelo de Atores (https://en.wikipedia.org/wiki/Actor_model)

	Inexperiente	Novato	Iniciante avançado	Competente	Proficiente	Especilaista
Progamação Reativa	○	○	○	○	○	○
Modelo de atores	○	○	○	○	○	○

Qual o seu nível de experiência com o estilo arquitetural de microsserviços? *

	Inexperiente	Novato	Iniciante Avançado	Competente	Proficiente	Especialista
Estilo arquitetural de microsserviços	○	○	○	○	○	○

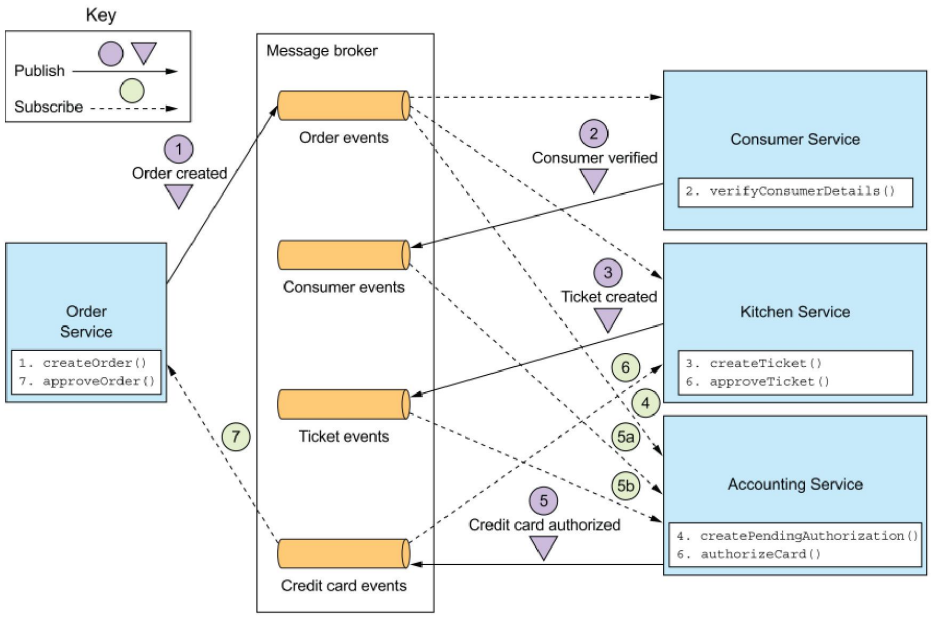
Como você avalia seu conhecimento sobre os seguintes padrões de microsserviços? *

A definição dos padrões pode ser encontrada nos seguintes links: SAGA (https://microservices.io/patterns/data/saga.html), Event Sourcing (https://microservices.io/patterns/data/event-sourcing.html)

	Inexperiente	Novato	Iniciante Avançado	Competente	Proficiente	Especialista
SAGA	○	○	○	○	○	○
Event Sourcing	○	○	○	○	○	○

Arquitetura da SAGA

Observe a figura a seguir que descreve a arquitetura utilizada como objeto de pesquisa deste trabalho. Ela descreve o fluxo de aprovação de um pedido em sistema de entrega de comida online, implementado utilizando o padrão SAGA. A figura a seguir foi retirada do capítulo 4 do livro Microservices Patterns, autorado por Chris Richardson



Legenda

Neste fluxo, um usuário realiza um pedido, e a seguinte cadeia de transações acontece:

1 - O serviço Order cria um pedido com o estado APPROVAL_PENDING, e publica um evento dizendo que o pedido foi criado (OrderCreated).

2- O serviço Consumer consome o evento OrderCreated e verifica se o cliente atende aos requisitos para prosseguir com aquele pedido. Em caso positivo, o evento ConsumerVerified é publicado.

3 - O serviço Kitchen consome o evento OrderCreated, valida o pedido, cria um ticket com estado CREATE_PENDING e publica um evento TicketCreated

4 - O Serviço Accounting consome o evento OrderCreated e cria uma autorização de cartão de crédito em estado PENDING

5 - O Serviço Accounting consome o evento TicketCreated e ConsumerVerified, efetua a cobrança no cartão de crédito do usuário e publica o evento CreditCardAuthorized

6 - O serviço Kitchen consome o evento CreditCardAuthorized, e muda o estado do Ticket para AWAITING_ACCEPTANCE

7 - O serviço Order consome o evento CreditCardAuthorized, e muda o estado do pedido para APPROVED e publica o evento OrderAproved

Comparação de código

Abaixo há trechos de código responsável pelo gerenciamento dos dados armazenados por uma entidade, do microsserviço responsável pelos pedidos, em duas versões: a primeira adota modelo de atores, por meio do Akka, já a segunda segue uma maneira síncrona tradicional

Trecho de código do microsserviço de pedido *sem Akka*

```
def getOrderById(id: String): Option[Order] = {
  val orderId = new ObjectId(id)
  val orderEvents = orders.find(Filters.eq("entity_id", consumerId)).results()
  var order = Option.empty[Order]
  orderEvents.foreach(event => {
    val eventData = event.get("event_data")
    event.get("event_type") match {
      case "OrderCreated" => order = Some(Order(orderId, eventData.get("consumer_id"), "PENDING"))
      case "OrderApproved" => order = Some(order.get.copy(status = "APPROVED"))
      case "ConsumerDeleted" => order = Option.empty[Order]
    }
  })
  order
}

def approveOrder(orderId: String) = {
  val eventId = new ObjectId()
  val eventDoc = Document(
    "_id" -> eventId,
    "event_type" -> "OrderApproved",
    "entity_id" -> orderId,
    "event_data" -> Document(
      "status" -> "APPROVED"
    )
  )
  orders.insertOne(eventDoc).printHeadResult()
}

def createOrder(consumerId: String): String = {
  val eventId = new ObjectId()
  val orderId = new ObjectId()
  val orderDoc = Document(
    "_id" -> eventId,
    "event_type" -> "OrderCreated",
    "entity_id" -> orderId,
    "event_data" -> Document(
      "consumer_id" -> consumerId,
      "status" -> "PENDING"
    )
  )
  orders.insertOne(orderDoc).printHeadResult()
  writeToKafka("order-created", orderDoc.toJson())
  orderId.toString
}
```

Código acima está disponível em:
https://github.com/wanderdasouza/saga-pattern/blob/master/order-service/src/main/scala/br/usp/orderservice/OrderRepository.scala#L17-L60

Como você classifica a legibilidade do código acima? *

Neste estudo, legibilidade é entendida como a facilidade de leitura e compreensão do código

	1	2	3	4	5	
Menos legível	○	○	○	○	○	Mais legível

Como você classifica a organização do código acima? *

Neste estudo, organização é entendida como a separação e modularização do código

	1	2	3	4	5	
Menos organizado	○	○	○	○	○	Mais organizado

Como você classifica a concisão do código acima? *

Neste estudo, concisão é entendida como a expressividade do código

	1	2	3	4	5	
Menos conciso	○	○	○	○	○	Mais conciso

Trecho de código do microsserviço de pedido *com Akka*

```
  private def commandHandler(context: ActorContext[Command], state: State, command: Command):
ReplyEffect[Event, State] = {
    implicit val mat: Materializer = Materializer(context.system)
    command match {
      case GetOrder(replyTo) =>
        Effect
          .reply(replyTo)(GetOrderResponse(Option(Order(state.consumerId, state.orderState))))
      case CreateOrder(order, replyTo) =>
        Effect
          .persist(OrderCreated(order))
          .thenReply(replyTo)(newUserState => newUserState.id)
      case ApproveOrder(replyTo) =>
        Effect
          .persist(OrderApproved)
          .thenReply(replyTo)(newState => newState.id)
      case DeleteOrder(replyTo) =>
        Effect.persist(OrderDeleted).thenReply(replyTo)(newOrderState =>
          newOrderState.id
        )
    }
  }

  private def eventHandler(context: ActorContext[_], state: State, event: Event): State = {
    implicit val mat: Materializer = Materializer(context.system)
    event match {
      case OrderCreated(order) =>
        val f = state.createOrder(order.consumerId)
        OrderProducer.publish("order-created", OrderCreatedToKafka("OrderCreated", f.id,
order.consumerId))
        f
      case OrderApproved => state.approveOrder()
      case OrderDeleted =>
        state.removeOrder()
    }
  }
```

Código acima está disponível em:
https://github.com/wanderdasouza/akka-saga-pattern/blob/master/order-service/src/main/scala/br/usp/OrderPersistence.scala#L30-L62

Como você classifica a legibilidade do código acima? *

Neste estudo, legibilidade é entendida como a facilidade de leitura e compreensão do código

	1	2	3	4	5	
Menos legível	○	○	○	○	○	Mais legível

Como você classifica a organização do código acima? *

Neste estudo, organização é entendida como a separação e modularização do código

	1	2	3	4	5	
Menos organizado	○	○	○	○	○	Mais organizado

Como você classifica a concisão do código acima? *

Neste estudo, concisão é entendida como a expressividade do código

	1	2	3	4	5	
Menos conciso	○	○	○	○	○	Mais conciso

Comente os pontos que julgar relevantes sobre suas respostas das questões anteriores

Sua resposta

Enviar

Limpar formulário

Nunca envie senhas pelo Formulários Google.

Este formulário foi criado em Universidade de São Paulo. Denunciar abuso

Bibliografia

[Bass *et al.* 2013] Len Bass, Paul Clements e Rick Kazman. *Software Architecture in Practice*. 3ª ed. 2013 (citado na pg. 3).

[Bonér 2016] Jonas Bonér. *Reactive Microservices Architecture: Design Principles for Distributed Systems*. O'Reilly, 2016 (citado nas pgs. 1, 7).

[Bonér 2017] Jonas Bonér. *Reactive Microsystems: The Evolution of Microservices at Scale*. O'Reilly, 2017 (citado na pg. 6).

[Evans 2004] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004 (citado na pg. 3).

[Ford *et al.* 2017] Neal Ford, Rebecca Parsons e Patrick Kua. *Building Evolutionary Architectures: Support Constant Change*. O'Reilly Media, Inc., 2017 (citado na pg. 1).

[Fowler, Lewis 2014] Fowler, Lewis. *Microservices*. https://martinfowler.com/articles/microservices.htmls/ Acessado em 2021-02-03. 2014 (citado na pg. 3).

[Newman 2015] Sam Newman. *Building Microservices*. O'Reilly Media, 2015 (citado nas pgs. 1, 3).

[Özkaya 2021] Mehmet Özkaya. *CQRS Design Pattern in Microservices Architectures*. https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c Acessado em 2021-02-03. 2021 (citado na pg. 6).

[Richardson 2018] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publication, 2018 (citado nas pgs. 3–5, 15, 16).

[Roestenburg *et al.* 2015] Raymond Roestenburg, Rob Bakker e Rob Williams. *Akka in Action*. 2015 (citado na pg. 1).

[Rosa 2018] Denis Rosa. *Saga Pattern | Application Transactions Using Microservices – Part I*. https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/ Acessado em 2021-02-03. 2018 (citado na pg. 4).

[Storti 2015] Brian Storti. *The actor model in 10 minutes*. https://www.brianstorti.com/the-actor-model/ Acessado em 2021-02-03. 2015 (citado na pg. 8).

[Vernon 2015] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. Addison-Wesley, 2015 (citado na pg. 7).