Transferência do Cuidado de Pacientes 3.0
Documento de Arquitetura de Software

Alunos:
Guilherme Faleiros de Siqueira
Andrey Dias
Felipe Kafuri

1. Introdução

1.1 Finalidade

Este documento tem como objetivo definir e documentar padrões arquiteturais, tecnologias de implementação, diretrizes, bem como visões arquiteturais pertinentes ao projeto **Transferência do Cuidado de Pacientes 3.0** no contexto da disciplina Padrões de Arquitetura de Software.

1.2 Escopo

Este documento se baseia no documento **Especificação do Trabalho Final PAS 2022.2** para extrair requisitos funcionais e não-funcionais, informações específicas sobre o domínio em questão e a partir disso, encontrar quais os atributos de qualidade devem ser priorizados e alocados para determinados padrões e visões documentadas.

2. Contexto da Arquitetura

2.1 Atributos de Qualidade Prioritários

De maneira geral, o software tem como objetivo priorizar os seguintes atributos de qualidade::

- **Portabilidade**: como estamos lidando com uma software que interage com múltiplos dispositivos e interfaces, como por exemplo, óculos VA/VR, web e eventualmente outros tipos de dispositivos móveis como smartphones e tablets.
- **Desempenho**: se tratando de um sistema ubíquo que lida com alto volume de transferência de dados, é necessário que o software consiga processar esses dados de maneira rápida e otimizando recursos computacionais.
- **Resiliência**: novamente, pensando no aspecto de processamento de dados, é necessário que o software seja confiável e possua tolerância a falhas.
- **Segurança**: se tratando de um software que lida com informações sensíveis sobre dados do histórico de saúde do paciente, é necessário que haja uma camada de segurança adicional.

3. Representação da Arquitetura

3.1 Padrões Arquiteturais escolhidos

Conforme supracitado, alguns atributos de qualidade foram identificados através da leitura do contexto e especificação do software e portanto, foram

utilizados como base para escolher os padrões e estilos arquiteturais a serem adotados. Vejamos:

- **Microsserviços**: dado o contexto, o sistema de software em questão pode ser resumido em componentes isolados que podem interagir de maneira indireta, cada um focando em um subdomínio específico que contém determinadas funcionalidades independentes que em conjunto corroboram para a completude do sistema como um todo, a utilização de microsserviços contempla os atributos de qualidade de **manutenibilidade** e **escalabilidade**. De maneira geral, podemos resumir os seguintes microsserviços:
    - **VA/VR**: o processamento e lógica relativas aos estímulos produzidos pelos aparelhos de VA/VR que os profissionais da saúde possuem.
    - **P-Avatar:** sincronização e gerenciamento do gêmeo digital do paciente no metaverso.
    - **BOT**: processamento de interações e estímulos com o BOT para auxílio de profissionais da saúde no que se diz respeito ao sistema.

- **Arquitetura Orientada a Eventos (Event-Driven Architecture):** para comunicação entre microsserviços, utilizaremos streaming de eventos entre as aplicações de forma assíncrona, de maneira que será possível refletir o estado entre os microsserviços de forma coesa e robusta. Se tratando de sistemas ubíquos, é comum que exista um componente atuando como barramento de eventos produzidos por diversos dispositivos dentro do sistema. De modo que a adoção desse padrão contempla os atributos de qualidade de **resiliência** e **desempenho**, pois permite que os eventos sejam processados de forma assíncrona, evitando que os recursos computacionais sejam sobrecarregados e processados sob-demanda e faça melhor utilização de recursos como CPU e memória.

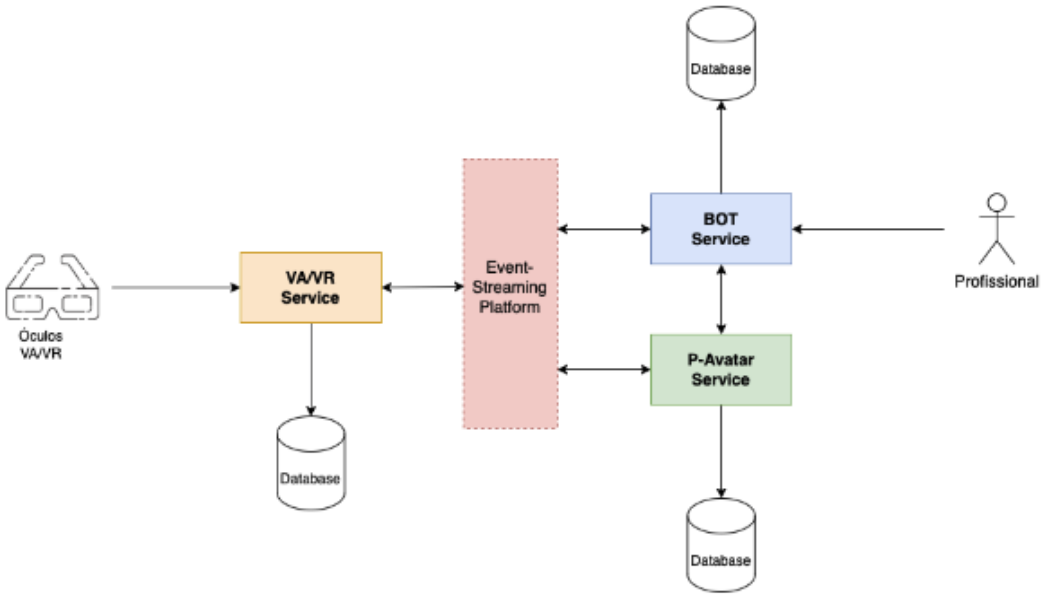
- **Cliente-Servidor:** como se trata de uma aplicação necessariamente web, que também fará interface com dispositivos como óculos VA/VR e outros dispositivos móveis, naturalmente se utiliza uma arquitetura cliente-servidor, em que, especificamente nesse caso, existe uma aplicação rodando no lado do cliente (navegador, óculos VA/VR, smartphone) e outra rodando no lado do servidor, de maneira que os serviços do lado do servidor sejam expostos através de uma interface estável (API) e que seja reutilizável através dos diversos clientes existentes no sistema de software. Um estilo arquitetural comum de se trabalhar em conjunto com este padrão é o **REST**, que permite a definição de uma API reutilizável e padronizada comunicando-se através da rede, de forma que esse estilo arquitetural será adotado neste projeto. Portanto, este padrão contempla o atributo de qualidade **portabilidade,** porém, como este padrão implica comunicação via rede (majoritariamente rede pública - internet), aspectos de segurança ficam expostos a algum tipo

de vazamento, portanto outro padrão será escolhido para contemplar o aspecto de segurança.

- **Arquitetura em camadas:** a arquitetura em camadas permite que o software (neste caso, cada microsserviço) seja modularizado em camadas que possuem uma responsabilidade bem definida e tenham baixo acoplamento entre si. De maneira geral, existe uma separação comum de camadas que prevê as camadas de *apresentação*, *controle*, *negócio* e *persistência*. Porém existem padrões como a Arquitetura Hexagonal e Clean Architecture que prescrevem uma nova visão e separação de responsabilidades de uma arquitetura em camadas. Neste projeto utilizaremos arquitetura hexagonal e sua organização será detalhada em seções posteriores neste mesmo documento. Um aspecto importante a discutir é que, com a possibilidade da construção de camadas fracamente acopladas, é possível que seja construída uma camada de segurança que faça validação de entrada e saída de dados e trabalhe com algum tipo de criptografia entre camadas, se necessário. Portanto, conseguimos contemplar o atributo de qualidade de **segurança** com este padrão.

3.2 Visão simplificada da arquitetura

Abaixo podemos ver uma versão simplificada, identificando principais componentes, contextos e atores da aplicação em questão:



O detalhamento dessa interação entre componentes e detalhes internos do funcionamento de cada um será explorado em seções futuras deste documento.

4. Visão de desenvolvimento

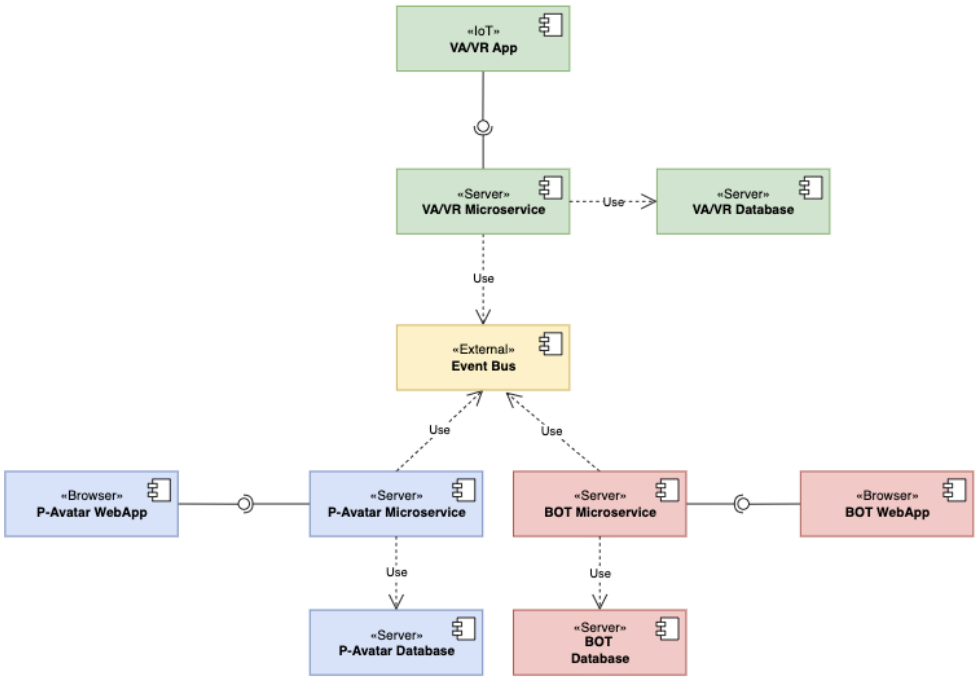
4.1 Visão geral

Esta representação tem como objetivo demonstrar de forma mais ampla quais os principais componentes existentes, as dependências entre si e as interfaces/APIs que cada componente provê. De maneira geral, existem 3 contextos que agrupam funcionalidades específicas:

- **VA/VR**: lidar e processar eventos relativos a estímulos gerados pelo óculos VA/VR utilizado pelos profissionais da saúde.
- **BOT**: lidar com estímulos e agrupar funcionalidades do chatbot/assistente virtual que pode auxiliar no atendimento de pacientes no que diz respeito a instruções que não necessitem de um profissional de saúde humano.
- **P-Avatar**: responsável por lidar com a sincronização e geração de gêmeos digitais de pacientes para simulação de procedimentos e visualizar informações sobre o histórico hospitalar do paciente em questão.

Cada um dos contextos mencionados, possui uma aplicação cliente, uma aplicação do lado do servidor e um banco de dados. De modo que cada contexto consiga expor suas funcionalidades de maneira independente como microsserviços.

4.2 Diagrama de componentes



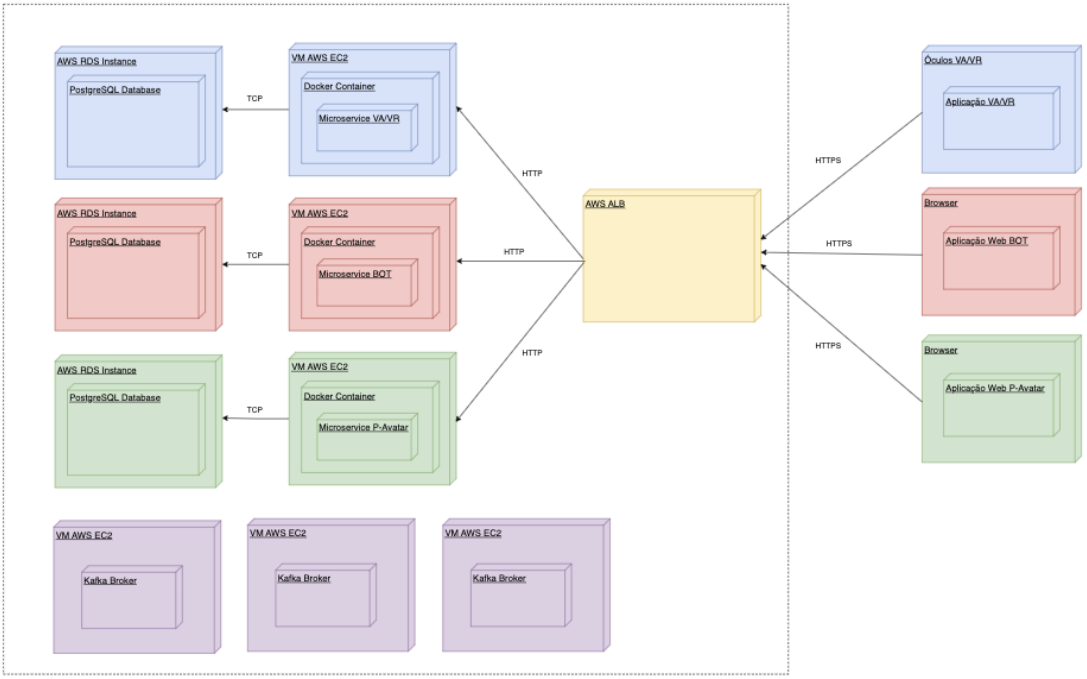
5. Visão Física

5.1 Visão geral

Para atender a alta demanda de processamento de dados, optamos pela utilização de um cloud provider com uma vasta gama de opções de serviços gerenciados e IaaS, a AWS. Projetamos uma infraestrutura que utilize recursos computacionais e serviços gerenciados a fim de minimizar possíveis custos adicionais, por se tratar de uma primeira versão do software, porém sem perder qualidade e restringir a evolução da infraestrutura para algo mais complexo e robusto.

A ideia central da infraestrutura do software é tirar proveito de conceitos como conteinerização para tornar cloud-native os microsserviços projetados e poder facilmente migrar de cloud provider ou tirar proveito de outros serviços da AWS como ECS ou EKS. Por outro lado, foi optado a utilização de recursos nativos da AWS como o Application Load Balancer para balanceamento de carga e roteamento e o RDS para provisionamento completo de banco de dados relacionais, neste caso, foi escolhido a utilização do PostgreSQL.

5.2 Diagrama de implantação



6. Visão lógica

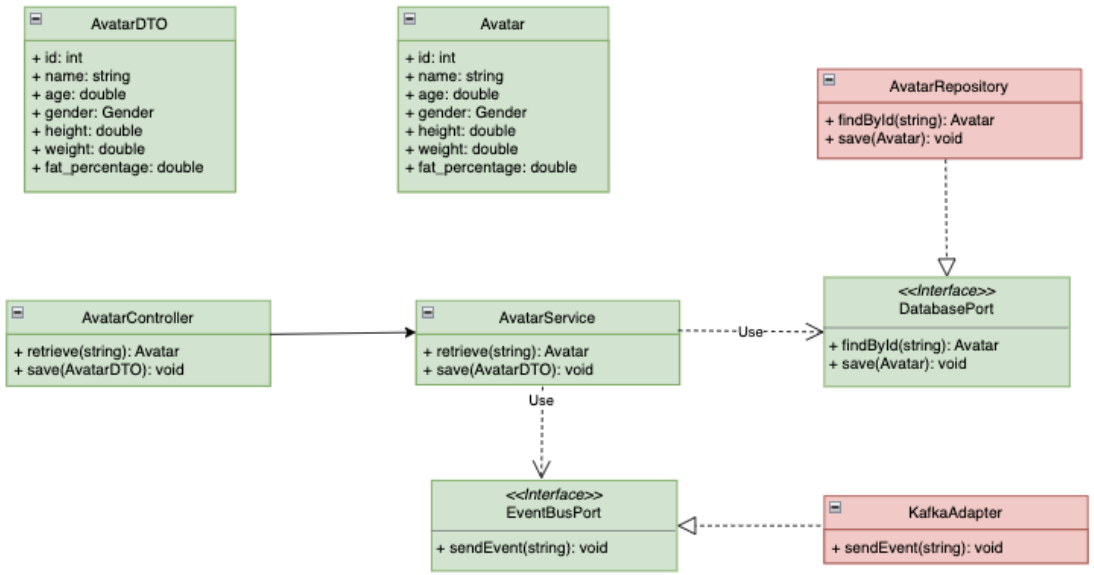
6.1 Visão Geral

A visão lógica tem como objetivo prover uma visão de baixo nível do software em questão, detalhando mais sobre os pequenos componentes do sistema como classes e objetos, mostrando a interação entre esses elementos. Para tal, foi escolhido o diagrama de classes para tal representação.

6.2 Diagrama de classes

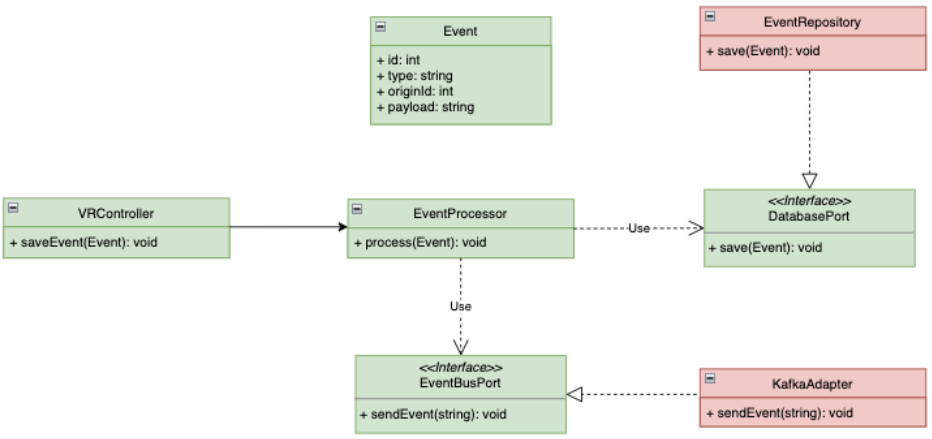
6.2.1 Microsserviço de Avatar

Este serviço tem como objetivo manter a sincronização do avatar do metaverso produzido. De forma que ele oferece funcionalidades para sincronização de dados e recuperação do estado de um avatar salvo para exibição no metaverso.



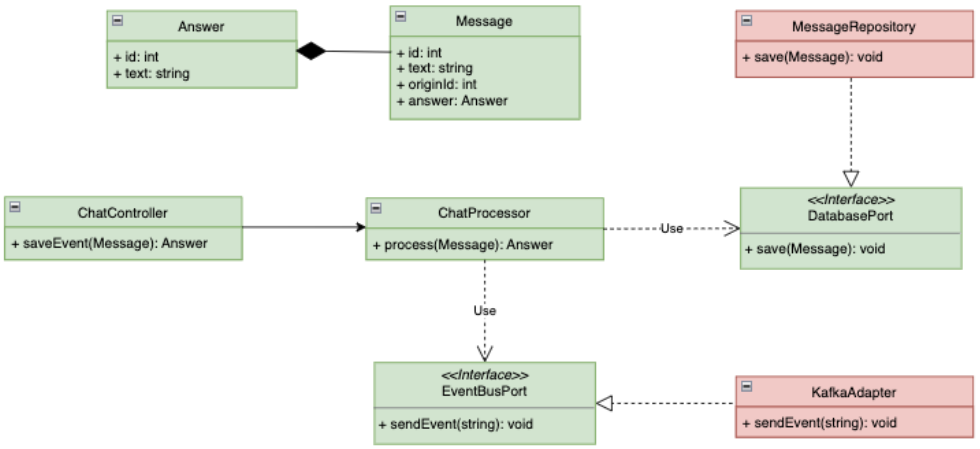
6.2.2 Microsserviço de VA/VR

Este microsserviço tem como objetivo reagir a eventos gerados pelos dispositivos de VA/VR, persistir, processar e alterar o estado de objetos dentro do metaverso de acordo com os estímulos realizados.



6.2.3 Microsserviço de BOT

Este microsserviço tem como objetivo processar mensagens relativas a interações com o chatbot e produzir respostas de acordo com o necessitado. Vejamos:



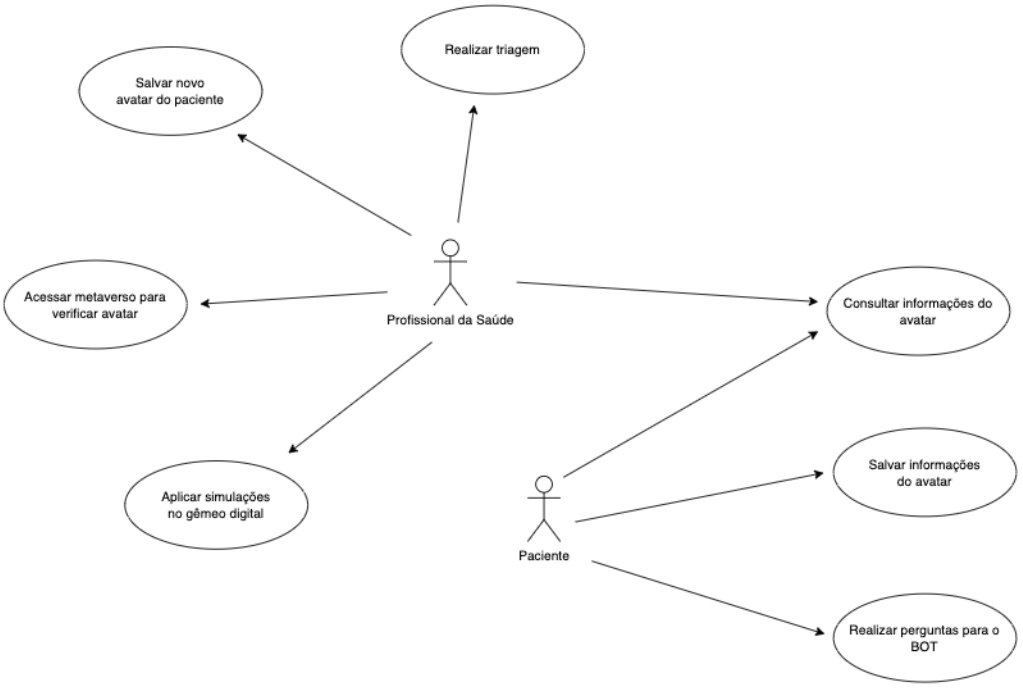
7. Visão de caso de uso

7.1 Visão geral

Esta visão tem como objetivo demonstrar as funcionalidades gerais do sistema de interesse, de modo a demonstrar os principais atores do sistema, bem como as interações com as funcionalidades descritas.

7.2 Diagrama de caso de uso

Através desse diagrama, conseguimos identificar os principais atores do sistema e quais as principais funcionalidades que os mesmos têm acesso:



8. Tecnologias de referência

8.1 EventBus

O componente **EventBus** é uma peça fundamental para o bom funcionamento da arquitetura desejada, uma vez que é através dele que os microsserviços se mantêm sincronizados e conseguem compartilhar eventos entre si. Portanto, é importante que seja um componente com alta resiliência e consiga atingir performance satisfatória não afetando a usabilidade do software. Para isso, foram cogitadas algumas plataformas de streaming de eventos consolidadas no mercado, as principais elencadas foram:

- Apache Kafka
- Apache Pulsar
- AWS Kinesis

Todas cumprem o papel de forma exemplar, mas considerando questões como custo, robustez, resiliência, performance e facilidade de encontrar conteúdo/suporte, o Apache Kafka foi o escolhido para este caso. No geral, o Kafka é amplamente adotado em sistemas envolvendo IoT/Sistemas ubíquos, tendo isso em vista, é uma ótima opção para nossa arquitetura de referência.

8.2 WebApps

No geral, os WebApps irão utilizar a linguagem Typescript e seu ecossistema para construção das aplicações web envolvidas neste processo. Vejamos algumas ferramentas importantes para a construção:

- **React.js**: é uma popular biblioteca para construção de aplicações web, altamente adotada no mercado e uma excelente opção para construção de aplicações web robustas, performáticas e complexas. Possui uma vasta gama de funcionalidades/bibliotecas que são facilmente integráveis e podem facilitar o desenvolvimento.
- **React XR**: uma biblioteca que se integra com o React.js e permite a criação de aplicação com VA/VR, de modo que permitirá a construção, principalmente, da aplicação de P-Avatar.
- **React Three Fiber** : uma biblioteca para React.js baseada em outra biblioteca chamada Three.js que tem como objetivo a construção de cenas 3D para a web e será útil para a construção do metaverso.

8.3 Microsserviços

Aproveitando a expertise com Typescript adquirida no desenvolvimento do front-end, a utilização de Node.js juntamente com Typescript foi escolhida. Algumas ferramentas utilizadas:

- **Express.js**: microframework utilizado para construção de aplicações web com JavaScript/Typescript.
- **TypeORM**: ferramenta de mapeamento objeto-relacional para typescript.
- **PostgreSQL**: banco de dados relacional utilizado para manter o estado dos microsserviços salvos.