**1. Introdução**

**Objetivo do Projeto**: O sistema foi projetado para gerenciar o fluxo de caixa diário de um comerciante, incluindo lançamentos de débitos e créditos e a geração de um relatório consolidado de saldo diário. O foco principal é garantir escalabilidade, resiliência, segurança e integração eficiente entre os serviços.

**Principais Componentes**:

* **DailyCashFlow.Auth**: Serviço de autenticação.
* **DailyCashFlow.WebApi**: Serviço para criação e gerenciamento de transações (controle de lançamentos).
* **DailyCashFlow.Processor**: Serviço de processamento de mensagens para consolidar os lançamentos diários.

### 2. ****Escalabilidade****

**Decisão**: A arquitetura foi planejada com **Kubernetes**, permitindo escalabilidade horizontal dos serviços à medida que o volume de requisições aumenta. Isso garante que, em momentos de pico, como descrito no desafio (50 requisições por segundo), o sistema possa crescer de forma eficiente, adicionando novos pods automaticamente.

**Justificativa**: O Kubernetes oferece **balanceamento de carga automático** e **autoescalabilidade** com base em métricas de uso de CPU e memória. Assim, o sistema mantém um desempenho consistente mesmo sob alta demanda.

**Uso de Cache**: O **Redis** será utilizado como mecanismo de cache para relatórios de transações do dia, aliviando a carga sobre o banco de dados e otimizando a recuperação de informações.

**3. Resiliência**

**Decisão**: A resiliência foi alcançada através da combinação de **Kubernetes** e **RabbitMQ com NServiceBus**. Isso garante que falhas individuais em componentes ou picos de carga sejam tratados sem afetar o funcionamento geral do sistema.

* **Kubernetes**: Em caso de falhas de pod ou esgotamento de recursos, o Kubernetes reinicia automaticamente os pods.
* **RabbitMQ + NServiceBus**: Permite a comunicação assíncrona entre os serviços, desacoplando os sistemas e garantindo que as mensagens sejam processadas mesmo se um dos serviços estiver temporariamente fora do ar.

**Justificativa**: Essa abordagem assegura que o serviço de controle de lançamentos (DailyCashFlow.WebApi) não seja afetado se o serviço de consolidação (DailyCashFlow.Processor) falhar, como exigido no desafio.

**Estratégias de Failover e Retry**: Implementamos políticas de **retry** e **circuit breakers** com ferramentas como **Polly** no .NET, garantindo que, em caso de falhas temporárias, as transações sejam automaticamente reprocessadas quando o sistema se recuperar.

**4. Segurança**

**Decisão**: A segurança foi projetada com foco em proteger os dados e garantir que somente usuários autenticados e autorizados tenham acesso aos serviços. Isso foi alcançado com:

* **Autenticação JWT**: O **DailyCashFlow.Auth** emite tokens JWT para autenticar e autorizar usuários em todos os serviços.
* **Criptografia de Senhas**: Senhas são armazenadas de forma segura, utilizando **hashing** e **salt**.
* **Proteção contra ameaças**: Incluímos **rate limiting**, **validação de entrada** (para evitar SQL injection e XSS), e estamos explorando o uso de **OWASP Security Headers** para proteger contra ataques comuns.

**Justificativa**: Isso garante a confidencialidade e integridade dos dados, além de evitar vulnerabilidades conhecidas.

**5. Padrões Arquiteturais**

**Decisão**: Optamos por uma arquitetura de **microsserviços**, onde cada serviço possui responsabilidade única e pode ser escalado, atualizado e mantido de forma independente.

* **DailyCashFlow.Auth**: Responsável exclusivamente pela autenticação e autorização.
* **DailyCashFlow.WebApi**: Focado em transações e no controle de lançamentos.
* **DailyCashFlow.Processor**: Processa as mensagens e calcula o saldo diário consolidado.

**Justificativa**: A arquitetura de microsserviços é ideal para este projeto, pois permite escalabilidade autônoma de cada serviço, alta disponibilidade e facilidade de manutenção e implantação.

**6. Integração**

**Decisão**: A comunicação entre serviços é feita de forma assíncrona via **RabbitMQ** com **NServiceBus**, garantindo que os serviços possam se comunicar de maneira resiliente e desacoplada.

* **RabbitMQ**: Utilizado como broker de mensagens, assegurando que as transações geradas pela WebApi sejam enviadas de forma confiável para o Processor.
* **NServiceBus**: Facilita a implementação de padrões de mensageria, como **garantia de entrega** (at-least-once) e **reprocessamento de mensagens**.

**Justificativa**: A escolha de RabbitMQ e NServiceBus se justifica pela robustez e confiabilidade dessas ferramentas, que garantem a entrega das mensagens e o processamento correto das mesmas.

**6. Integração**

**Decisão**: A comunicação entre serviços é feita de forma assíncrona via **RabbitMQ** com **NServiceBus**, garantindo que os serviços possam se comunicar de maneira resiliente e desacoplada.

* **RabbitMQ**: Utilizado como broker de mensagens, assegurando que as transações geradas pela WebApi sejam enviadas de forma confiável para o Processor.
* **NServiceBus**: Facilita a implementação de padrões de mensageria, como **garantia de entrega** (at-least-once) e **reprocessamento de mensagens**.

**Justificativa**: A escolha de RabbitMQ e NServiceBus se justifica pela robustez e confiabilidade dessas ferramentas, que garantem a entrega das mensagens e o processamento correto das mesmas.

### 8. ****Documentação****

Além dos diagramas de classes e entidade-relacionamento, criei um **diagrama de comunicação entre microsserviços**, detalhando como o **DailyCashFlow.WebApi** envia transações para o **DailyCashFlow.Processor** via RabbitMQ, e como o **DailyCashFlow.Auth** autentica e autoriza os usuários.

### 9. ****Conclusão****

Com a arquitetura proposta, estamos atendendo aos requisitos de escalabilidade, resiliência e segurança de forma eficiente. Usamos boas práticas arquiteturais e padrões reconhecidos, como microsserviços e mensageria assíncrona, para garantir que o sistema seja robusto, performático e fácil de manter.