Gerenciamento de Lançamentos Financeiros

Autor: Pierre Tinchant Pinto
Data: 11 de Outubro de 2024
Versão: 1.0

1. Resumo Executivo

Este documento tem como objetivo apresentar a arquitetura proposta para a solução de gestão de lançamentos financeiros e consolidação diária. Com foco em escalabilidade, segurança e desempenho, a proposta alinha-se as necessidades de integração dos processos do cliente, otimizando a entrega de valor para a organização.

A solução utiliza **SQL Server** como banco de dados, devido a sua robustez e capacidade de gerenciamento de transações financeiras, em conjunto com **C#**, pela sua sinergia com a plataforma Microsoft e alta performance. O serviço de controle de lançamentos financeiros será o responsável por registrar e monitorar os lançamentos em tempo real, garantindo a consistência e integridade dos dados.

A escolha do **NUnit** para os testes unitários foi feita pela sua integração fluida com o **C#**, permitindo a criação de testes automatizados robustos, o que vai garantir a qualidade e confiabilidade do código. Já os diagramas foram desenvolvidos a fim de proporcionar uma representação visual clara e concisa da arquitetura e dos fluxos de dados.

Histórico de Revisões

Versão	Data	Autor	Descrição
1.0	11/10/2024	Pierre Tinchant Pinto	Criação inicial do documento com arquitetura proposta e detalhamento.

Índice

3. Introdução

Com o crescente volume de dados financeiros e a necessidade de uma gestão eficiente, é imprescindível adotar uma abordagem que considere não apenas a tecnologia, mas também a praticidade e a experiência dos usuários finais.

A proposta aqui apresentada visa atender às demandas do cliente, garantindo que os processos sejam integrados de maneira fluida e eficiente. Além disso, a documentação tem como objetivo servir de base para futuras implementações e melhorias.

4. Objetivo da Solução

O principal objetivo da solução é fornecer uma maneira eficiente para o usuário realizar lançamentos financeiros e solicitar relatórios, com o uso de processamento assíncrono para melhorar o desempenho e garantir que as operações críticas não fiquem bloqueadas em momentos de alta demanda. A arquitetura também facilita a escalabilidade automática e a manutenção contínua, sem impactar os usuários finais.

5. Requisitos Funcionais e Não Funcionais

5.1 Requisitos Funcionais

- **Lançamentos Financeiros**: O sistema deve permitir que o usuário registre lançamentos financeiros por meio de uma interface web.
- **Geração de Relatórios**: O sistema deve gerar relatórios financeiros baseados em parâmetros fornecidos pelo usuário, como data, tipo de transação, etc.
- **Notificações**: O sistema deve notificar o usuário quando um relatório solicitado for gerado e estiver disponível para download.

5.2 Requisitos Não Funcionais

- **Escalabilidade**: A solução deve ser escalável horizontalmente, principalmente os serviços de lançamentos financeiros.
- **Resiliência**: A arquitetura deve garantir alta disponibilidade e recuperação rápida em caso de falhas de componentes.
- **Segurança**: Todos os dados transmitidos e armazenados devem ser protegidos por criptografia (em trânsito e em repouso).
- **Monitoramento e Observabilidade**: Toda a infraestrutura deve ser monitorada com métricas de desempenho e logs detalhados para facilitar o diagnóstico e a resolução de problemas.
- **Desempenho**: A solução deve garantir tempos de resposta de API inferiores a 200ms na maioria das solicitações e suportar no mínimo 50 requisições por segundo, limitando-se a no máximo 5% de perda de requisições, utilizando-se dos recursos de escalabilidade e resiliência previamente descritos.

6. Arquitetura de Infraestrutura

6.1 Visão Geral

A infraestrutura aqui sugerida utiliza os serviços da Azure para orquestrar, armazenar, processar e comunicar entre os componentes. A comunicação é feita totalmente de forma assíncrona para manter a alta disponibilidade e o desempenho sob demandas pesadas..

6.2 Justificativa das Escolhas Tecnológicas

- **Azure Functions**: Foram escolhidas pela sua capacidade de escalabilidade automática e custo reduzido (pagamento por execução). São usadas tanto no processamento de lançamentos quanto na geração de relatórios.
- **Azure Service Bus**: Utilizadas para o gerenciamento de filas para garantir que o processamento das mensagens seja feito de maneira confiável e ordenada. Suporta cenários com grande volume de dados garantindo a entrega.
- **SQL Server**: Serviço de banco de dados, será utilizado para armazenar os lançamentos financeiros e informações dos relatórios gerados. Sua robustez e integração com a Azure proporcionam consistência e alta performance para consultas complexas.
- **Redis Cache**: Será utilizado como uma camada de cache, atuando como um armazenamento temporário de dados, assim trazendo uma melhora no tempo de resposta em operações que exigem alta velocidade de leitura de dados.
- **Blob Storage**: É um recurso de armazenamento em nuvem, será utilizado para armazenar os relatórios gerados, uma solução eficiente para armazenar grandes arquivos de forma barata e escalável.
- **Application Insights**: Será implementado para monitorar e fornecer rastreamento quanto ao desempenho da aplicação, erros e de solicitações.

6.3 Comparação entre Tecnologias

1. Azure Functions vs. API Tradicional (IaaS ou PaaS)

- **Performance**: Azure Functions é uma solução serverless, o que significa que a infraestrutura subjacente escala automaticamente com base na demanda. Isso elimina a necessidade de provisionar manualmente servidores e garante uma resposta rápida durante picos de uso. No entanto, para operações muito intensivas em processamento, pode haver um tempo de inicialização frio (cold start) que pode impactar o tempo de resposta em instâncias ociosas.
- **Preço**: Azure Functions cobra com base na execução e tempo de processamento (pago por uso). Isso é altamente eficiente em termos de custo, especialmente em ambientes com cargas variáveis e imprevisíveis. Uma API tradicional, hospedada em IaaS ou PaaS, tem custos mais previsíveis, mas tende a ser mais cara em cenários de baixa utilização, uma vez que você paga por recursos provisionados, independentemente do uso.
- **Escalabilidade**: Azure Functions automaticamente escala horizontalmente para atender à demanda, sem necessidade de gerenciamento manual de servidores. Isso o torna ideal para cenários de pico e flutuações na carga. Já em uma API tradicional, a escalabilidade precisa ser configurada manualmente, seja em uma infraestrutura gerenciada ou não, e pode requerer planejamento e intervenção.
- **Facilidade de Gerenciamento**: Azure Functions é altamente gerenciada pela própria Microsoft, o que reduz a sobrecarga de gerenciamento de infraestrutura. Com PaaS ou IaaS, a equipe de DevOps precisa monitorar, atualizar e manter o ambiente subjacente, o que pode adicionar complexidade e esforço.
- **Facilidade de Provisão**: A criação de uma Azure Function é rápida e facilmente automatizável por meio do Azure DevOps ou pipelines CI/CD. Para APIs tradicionais, o processo de provisão pode ser mais complexo, exigindo a configuração de servidores, banco de dados e rede de forma mais manual.

Foi escolhido Azure Functions pela combinação de escalabilidade automática, baixo custo e facilidade de gerenciamento em um cenário onde a carga é variável, como no processamento de relatórios ou lançamento de dados financeiros. Em termos de flexibilidade e eficiência, as Azure Functions se destacam por possibilitar uma infraestrutura altamente ágil.

2. SQL Server vs. Azure Cosmos DB vs. PostgreSQL

- **Performance**: SQL Server, uma solução madura e robusta, apresenta um ótimo desempenho para transações ACID e cargas de trabalho relacionais. Em cenários de leitura intensiva ou em sistemas altamente transacionais, o SQL Server pode ser otimizado com índices e cache (como o Redis) para melhorar o desempenho. Azure Cosmos DB oferece baixa latência e escalabilidade horizontal global, ideal para cenários de grande volume de dados distribuídos geograficamente. No entanto, ele é mais caro e não é tão otimizado para transações relacionais pesadas. PostgreSQL, sendo uma alternativa open-source, oferece ótima performance para consultas complexas, especialmente com dados não relacionais e relacionais, mas podem ser necessários ajustes finos em cenários de alta carga transacional.
- **Preço**: SQL Server no Azure pode ser caro, especialmente se for necessária uma grande capacidade de armazenamento e licenças empresariais. Contudo, o preço pode ser mais previsível em comparação com o Cosmos DB, que cobra por throughput provisionado (RUs), que pode aumentar drasticamente dependendo do uso. PostgreSQL tende a ser uma solução mais econômica, principalmente por ser open-source.
- **Escalabilidade**: SQL Server no Azure é altamente escalável verticalmente, com opções de elasticidade para aumentar a capacidade com base na demanda. No entanto, não se adapta tão bem à escalabilidade horizontal (sharding) quanto o Cosmos DB, que foi projetado especificamente para esse tipo de crescimento. PostgreSQL também tem boas opções de escalabilidade, mas pode ser mais difícil de configurar para ambientes altamente distribuídos.
- **Facilidade de Gerenciamento**: SQL Server é amplamente conhecido pelas equipes de infraestrutura e oferece uma excelente integração com ferramentas de monitoramento e backup do Azure. Cosmos DB, por ser uma solução gerenciada pela Microsoft, exige menos intervenção administrativa, mas exige um bom planejamento de throughput para evitar custos excessivos. PostgreSQL é menos gerenciado, mas oferece flexibilidade e uma comunidade rica em suporte e ferramentas.

O SQL Server foi escolhido por sua robustez em cenários transacionais e sua integração com a Azure, especialmente para empresas já investidas no ecossistema da Microsoft. Embora Cosmos DB ofereça vantagens em escala global, ele não é necessário no contexto atual de transações.

3. Redis Cache vs. Armazenamento em Disco

- **Performance**: Redis Cache oferece desempenho superior quando comparado ao armazenamento em disco tradicional, fornecendo acesso rápido a dados frequentemente acessados. Sua latência de microssegundos permite armazenar sessões, configurações ou dados temporários que necessitam de leitura/escrita rápida.
- **Preço**: Redis pode ser mais caro em grandes implementações, principalmente em clusters distribuídos. No entanto, o custo é justificado pela melhora no tempo de resposta das aplicações e redução da carga em outros serviços, como o banco de dados SQL. O armazenamento em disco, por outro lado, é uma opção mais econômica, mas inadequada para dados que requerem acesso rápido.
- **Escalabilidade**: Redis escala horizontalmente com facilidade, suportando grandes volumes de dados em memória. Isso o torna ideal para sistemas com grandes volumes de leitura e gravação. O armazenamento em disco tradicional não é projetado para escalabilidade em tempo real.
- **Facilidade de Gerenciamento**: Redis gerenciado pelo Azure facilita a configuração e o gerenciamento de cache, com suporte para backup e recuperação automática. O armazenamento em disco, em comparação, oferece menos flexibilidade e requer gerenciamento constante por parte da equipe de infraestrutura.

O Redis Cache foi escolhido pela alta performance na leitura de dados frequentemente acessados e pela capacidade de reduzir a carga sobre o SQL Server, resultando em melhor desempenho geral do sistema.

4. Azure Application Insights vs. ELK Stack (Elastic, Logstash, Kibana)

- **Performance:** Azure Application Insights é uma solução robusta, altamente integrada com a infraestrutura do Azure, projetada para monitoramento de performance, telemetria e diagnósticos de aplicações em tempo real. Ele oferece baixo impacto na performance da aplicação, enquanto coleta métricas, logs e rastreamentos.
  O ELK Stack, uma solução de código aberto, também oferece uma solução poderosa de monitoramento, mas pode adicionar complexidade e peso à infraestrutura, exigindo mais recursos para o processamento de logs.
- **Preço:** Application Insights cobra com base na quantidade de dados ingeridos e armazenados, e geralmente é mais econômica em comparação com a manutenção e operação de uma pilha ELK personalizada, que pode exigir provisionamento e manutenção de servidores dedicados.
- **Escalabilidade:** Application Insights escala automaticamente com as aplicações no Azure, enquanto o ELK Stack exige mais planejamento e ajuste manual para lidar com grandes volumes de dados.
- **Facilidade de Gerenciamento:** Application Insights oferece fácil configuração com o ecossistema Azure, reduzindo a necessidade de configuração manual e permitindo uma integração fluida com outras ferramentas de monitoramento e alertas. O ELK Stack, por outro lado, requer maior esforço de configuração e gestão, além de monitoramento ativo da infraestrutura.

Application Insights foi escolhido pela facilidade de integração, escalabilidade automática e custos menores de operação em comparação ao ELK Stack, tornando-o uma solução mais apropriada para ambientes monitorados dentro do Azure.

6.4 Componentes Principais da Solução

- **Aplicação Web (Portal)**: O ponto de interação do usuário com a plataforma. Utiliza **HTTPS** para comunicações seguras e suporta autenticação via **OAuth 2.0**.
- **API REST**: Ponto de entrada que processa todas as requisições dos usuários e integra com os demais serviços.
- **Azure Service Bus**: Gerencia as mensagens de requisição de relatórios e lançamentos, garantindo seu processamento assíncrono.
- **Azure Functions (Geração de Relatórios e Lançamentos)**: Functions desacopladas para lidar separadamente com os lançamentos e relatórios, otimizando o processamento em paralelo.
- **SQL Server**: Armazena os dados financeiros de maneira relacional, garantindo integridade e persistência.
- **Redis Cache**: Cache de memória de alto desempenho utilizado para otimizar consultas frequentes.
- **Blob Storage**: Armazena os arquivos de relatório para posterior recuperação.
- **Application Insights**: Usado para monitorar métricas de performance, erros e alertas de saúde da aplicação.

6.5 Segurança

- **Autenticação e Autorização**

Para garantir que apenas usuários autorizados tenham acesso aos recursos da aplicação, adotamos medidas de segurança robustas. Utilizamos o OAuth 2.0, que oferece uma maneira segura de autenticar usuários externos por meio de tokens de acesso. Para as empresas, implementamos a autenticação via Active Directory (AD), integrando-a com o Azure AD, o que facilita o gerenciamento centralizado de contas e permissões. A autorização é feita através do modelo RBAC (Controle de Acesso Baseado em Funções), assegurando que cada usuário tenha apenas os acessos necessários.

As credenciais sensíveis, como tokens e senhas, são armazenadas com segurança no Azure Key Vault, que fornece uma camada extra de proteção por meio de criptografia e gerenciamento centralizado de chaves.

- **Criptografia**

Para proteger os dados que estão sendo transmitidos, implementamos o protocolo TLS (Transport Layer Security), garantindo que as informações permaneçam íntegros e confidenciais entre o cliente e o servidor. A gestão das chaves de criptografia será feita pelo Azure Key Vault, assegurando que essas chaves fiquem protegidas contra acessos não autorizados e que sejam trocadas regularmente, seguindo as melhores práticas de segurança.

- **Gerenciamento de Identidades e Acessos (IAM)**

Gerenciamos usuários e suas permissões através do Azure AD, que centraliza a identidade e o controle de acesso. Automatizaremos o processo de criação e remoção de contas, garantindo que apenas usuários ativos tenham acesso ao sistema. Além disso, aplicamos regras de acesso com base em políticas de segurança, evitando a concessão de permissões desnecessárias.

- **Proteção contra Ataques**

Implementaremos medidas proativas para proteger a aplicação contra ataques comuns. Utilizaremos prepared statements para evitar injeções de SQL, políticas de Content Security Policy (CSP) para prevenir XSS (Cross-Site Scripting) e implementações de proteção contra CSRF (Cross-Site Request Forgery) em todas as requisições. Também configuraremos firewalls e balanceadores de carga para mitigar riscos de ataques DDoS (Distributed Denial of Service). Além disso, a aplicação gerenciará erros e exceções de forma centralizada, assegurando que informações sensíveis não sejam expostas em mensagens de erro.

- **Varreduras de Vulnerabilidades**

No nosso pipeline de CI/CD, vamos integrar ferramentas de SAST (Static Application Security Testing) e DAST (Dynamic Application Security Testing) para identificar vulnerabilidades durante o desenvolvimento e a execução. Ferramentas como SonarQube (para SAST) e OWASP ZAP (para DAST) serão utilizadas para realizar análises automáticas de segurança, garantindo que a segurança esteja sempre em dia com cada nova versão da aplicação.

6.6 Testes e Qualidade

- **Testes Unitários e de Integração**

A solução conta com uma ampla cobertura de testes unitários e de integração, que asseguram a qualidade do código ao validar o comportamento correto das funções e a interação entre componentes. Esses testes serão automatizados e farão parte do nosso processo de CI/CD, garantindo que cada nova alteração passe por rigorosas validações antes de ser incorporada ao código.

- **Testes de Segurança**

Realizaremos testes de segurança regularmente, incluindo pentests e varreduras de vulnerabilidades, para garantir que a aplicação esteja sempre protegida contra ameaças conhecidas e desconhecidas. Essas análises ocorrerão ao menos uma vez por ciclo de versão e em momentos críticos, como grandes atualizações de funcionalidades.

- **Monitoramento e Alerta**

Vamos monitorar constantemente métricas de segurança essenciais, como tentativas de login falhas e acessos não autorizados. Essas métricas serão acompanhadas de alertas automáticos para eventos que exijam atenção imediata, permitindo uma resposta rápida a qualquer ameaça detectada.

6.7 Melhorias Adicionais

- **Resiliência**

Para aumentar a resiliência da aplicação, adotaremos estratégias sólidas de recuperação de desastres, como backups automáticos e replicação de dados.

- **Governança**

Implementaremos um processo de governança eficaz para o controle de versões e aprovação de mudanças. Todas as alterações no código passarão por revisões formais, assegurando que estejam alinhadas com os objetivos do projeto. Também teremos um plano de manutenção que inclui atualizações de segurança e melhorias regulares, mantendo a solução sempre atualizada e eficiente.

- **Documentação**

A documentação estará sempre atualizada e acessível a todos os membros do projeto. Usaremos ferramentas automatizadas, como Swagger, para gerar documentação de API, facilitando o acesso a detalhes técnicos e integrações. Esse processo garante que a equipe de desenvolvimento, operações e outros stakeholders tenham informações relevantes para dar suporte e evoluir a aplicação.

7. Fluxo de Processos

7.1 Processamento de Lançamentos

1. O usuário realiza um lançamento via **Portal**.
2. O **Portal** envia a solicitação para a **API REST**, que por sua vez envia uma mensagem para a fila de **Service Bus**.
3. A **Azure Function Lançamentos** consome a mensagem, realiza as validações necessárias e grava os dados no **SQL Server**. Um cache temporário pode ser gravado no **Redis** para otimizar a leitura subsequente e utilização de dados obtidos em validações.

7.2 Geração de Relatórios

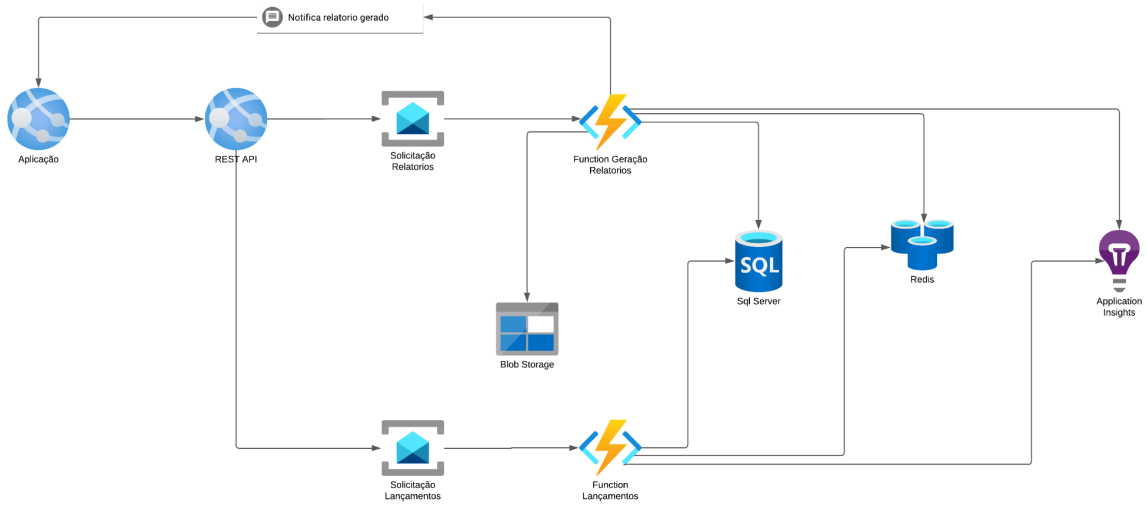
1. O usuário solicita um relatório por meio do **Portal**.
2. A solicitação é enviada para a **API REST**, que emite uma mensagem para o **Service Bus**.
3. A **Azure Function de Relatórios** consome essa mensagem, coleta os dados necessários do **SQL Server** e armazena o relatório gerado no **Blob Storage**.
4. Após a conclusão, uma mensagem é enviada ao usuário informando que o relatório está disponível.

7.3 Notificação de Relatório Gerado

1. O sistema utiliza o **Service Bus** para enviar uma notificação ao portal.
2. O **Portal** exibe uma mensagem para o usuário quando o relatório estiver pronto.

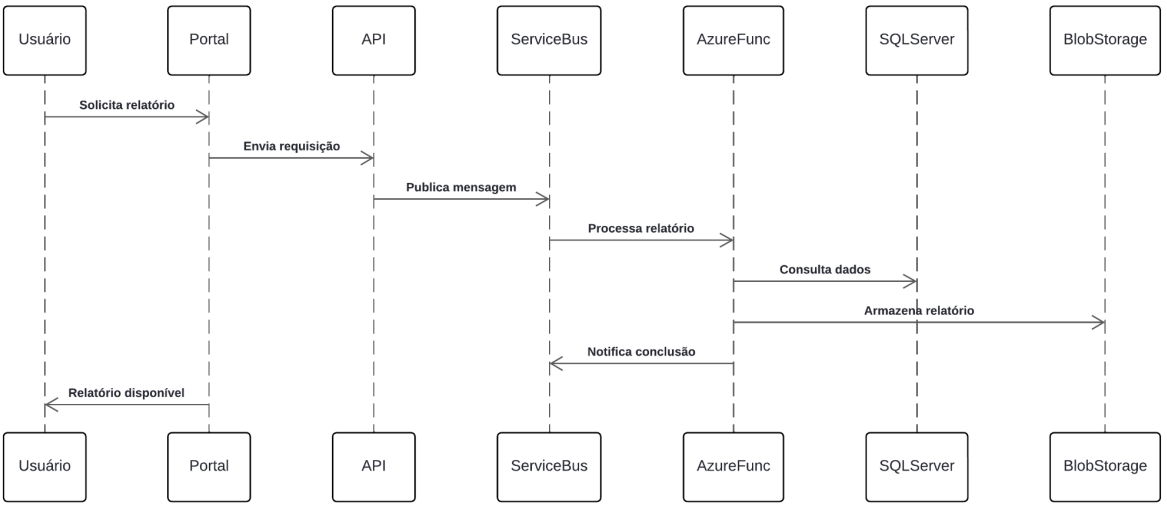
8. Diagramas

8.1 Diagrama de Componentes

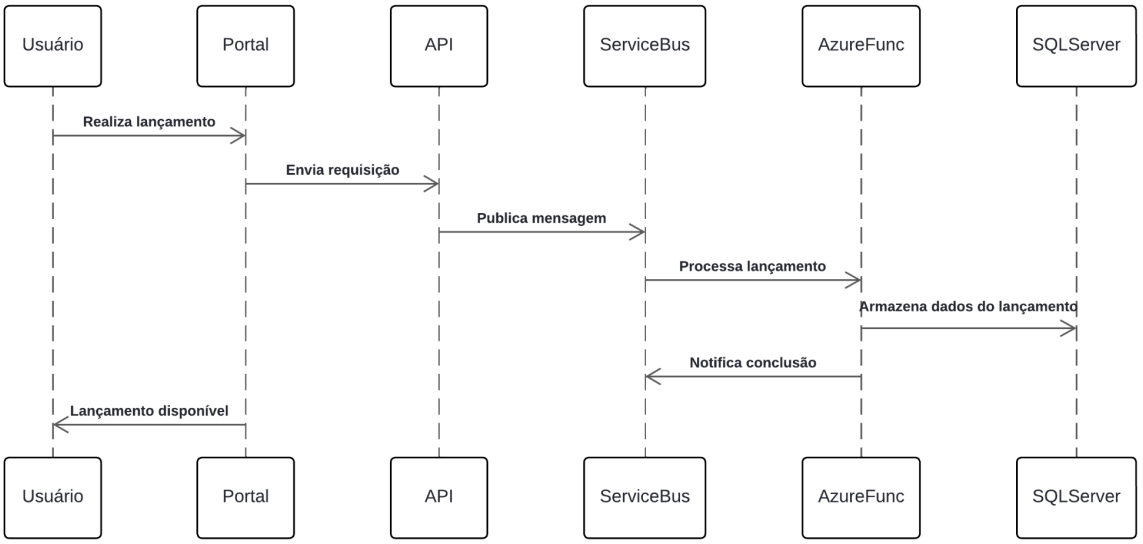


8.2 Diagramas de Sequência

1. Geração de relatórios



2. Realização de lançamentos



9. Implementação

A implementação será dividida nas seguintes fases:

1. **Configuração do Ambiente**: Criar a infraestrutura na Azure, incluindo o SQL Server, Azure Functions, Redis e Blob Storage.
2. **Desenvolvimento da Aplicação**: Desenvolver o frontend e backend da aplicação web, utilizando frameworks apropriados.
3. **Testes e Validação**: Realizar testes de desempenho e segurança para garantir que os requisitos sejam atendidos.
4. **Implantação**: Publicar a solução em produção, monitorando o desempenho e a resposta dos usuários.

10. Considerações sobre Performance

O uso de cache com **Redis** e a arquitetura desacoplada via **Service Bus** garantem tempos de resposta rápidos e processamento eficiente. As **Azure Functions** escalam automaticamente para lidar com picos de uso, assegurando um tempo de resposta abaixo de 200ms para a maioria das requisições.

11. Conclusão

A arquitetura proposta é robusta, segura e escalável, atendendo aos requisitos de controle de lançamentos e geração de relatórios de maneira eficiente. A integração com o ecossistema da Azure facilita o gerenciamento e o monitoramento contínuo, garantindo a entrega de valor para o cliente.