

DOCUMENTO DE VISÃO DA SOLUÇÃO - CHALLENGE VIVO 2024 - 4SIS - FIAP

O documento de visão deve ser objetivo, com as informações que vocês têm no momento, ou seja, deve apresentar o que pretendem realizar. Embora não haja um *template* fixo para o Documento de Visão, recomendamos alguns itens que um bom Documento precisa ter, minimamente.

Importante: cada item precisa trazer a quantidade de informações suficientes para seu entendimento, não economize na descrição, mas seja objetivo.

Nome e Descrição do Cliente (Empresa):

Vivo Telecomunicações

A Vivo Telecomunicações é uma das principais empresas de telecomunicações do Brasil, oferecendo uma ampla gama de serviços, incluindo telefonia móvel, internet fixa e móvel, TV por assinatura e serviços digitais. Com uma vasta base de clientes e uma variedade de produtos e pacotes disponíveis, a Vivo busca otimizar a integração e normalização de dados de seus sistemas legados para fornecer uma experiência mais consistente e personalizada aos seus usuários finais. Este projeto visa atender às necessidades da Vivo Telecomunicações, garantindo a entrega eficiente e segura de dados canônicos para seus aplicativos e serviços.

Nome do projeto:

Plataforma de Integração e Normalização de Dados da Vivo

Nome do time e integrantes:

Carlos Eduardo RM 87385, Gean Pfefer RM 88138, Henrique Sartori RM 87091, Matheus Rebola RM 81030, Ygor Calimanis RM 86587

Objetivo deste documento:

Este documento tem como objetivo apresentar uma visão abrangente do projeto da Plataforma de Integração e Normalização de Dados da Vivo. Nele, serão descritos o propósito, escopo, clientes envolvidos, tecnologias utilizadas, bem como os principais componentes e funcionalidades do sistema. Além disso, serão abordadas as etapas de desenvolvimento, implementação e operação da solução, destacando os benefícios esperados e os desafios a serem enfrentados durante o processo.

Histórico de revisões do modelo:

Versão (0.0.0)	Data (DD/MM/YYYY)	Autores	Descrição (O que fez no doc?)
0.0.1	15/04/2024	Matheus Rebola	Criei a primeira versão
0.0.2	02/09/2024	Matheus Rebola / Ygor Calimanis	Acréscimo da Arquitetura

Descrição (propósito, justificativa e motivação do desafio):

Propósito:

O objetivo deste projeto é criar uma solução robusta e escalável para fornecer dados canônicos homogeneizados de diversos sistemas legados, garantindo alta disponibilidade, consistência e baixa latência para serem consumidos por APIs que atendem os clientes através de um aplicativo. A solução visa centralizar e normalizar os dados provenientes de sistemas legados heterogêneos, fornecendo uma fonte confiável e consistente de informações para os clientes.

Justificativa:

Muitas empresas enfrentam o desafio de lidar com dados dispersos em diferentes sistemas legados, cada um com sua própria estrutura e formato. Isso dificulta a integração e a disponibilização desses dados para aplicativos modernos que exigem informações consistentes e atualizadas em tempo real. A criação de uma solução centralizada de entrega de dados canônicos não apenas simplifica o acesso aos dados, mas também melhora a experiência do usuário final, permitindo que os aplicativos ofereçam recursos avançados e personalizados.

Motivação do Desafio:

O desafio deste projeto reside na necessidade de integrar sistemas legados diversos, cada um com suas próprias peculiaridades e formatos de dados, em uma arquitetura moderna e escalável. Além disso, é crucial garantir a consistência e a integridade dos dados, mesmo em face de falhas nos sistemas legados. A motivação principal é proporcionar uma experiência contínua e confiável para os usuários finais, permitindo que eles acessem informações precisas e atualizadas a qualquer momento, independentemente das complexidades dos sistemas legados subjacentes.

Escopo (o que o desafio abrange e contempla):

O Desafio Aborda:

1. **Integração de Sistemas Legados:** Desenvolvimento de microsserviços para integrar e extrair dados de sistemas legados heterogêneos, utilizando técnicas assíncronas para garantir a captura em tempo real das atualizações.
2. **Normalização de Dados:** Implementação de um microsserviço dedicado para normalizar os dados provenientes dos sistemas legados para um formato canônico, garantindo consistência e integridade.
3. **Entrega de Dados Canônicos:** Criação de um microsserviço de entrega de dados que expõe APIs para que os clientes possam acessar os dados canônicos de forma segura e eficiente, garantindo baixa latência e alta disponibilidade.
4. **Ressincronização de Dados:** Implementação de um mecanismo de ressincronização de dados para lidar com falhas nos sistemas legados e garantir que as atualizações sejam refletidas corretamente na solução centralizada.
5. **Escalabilidade e Resiliência:** Projeto da arquitetura para ser escalável horizontalmente e resiliente a falhas, utilizando ferramentas como Apache Kafka e microsserviços distribuídos.
6. **Segurança e Autenticação:** Implementação de mecanismos robustos de autenticação e autorização para garantir que apenas usuários autorizados tenham acesso aos dados.

O Desafio Não Aborda:

1. **Refatoração de Sistemas Legados:** Não inclui a refatoração dos sistemas legados em si, mas sim a integração e normalização de seus dados.
2. **Desenvolvimento do Aplicativo Cliente:** Não inclui o desenvolvimento do aplicativo cliente que irá consumir os dados canônicos fornecidos pela solução.
3. **Gestão de Dados Sensíveis:** Não inclui a gestão de dados sensíveis ou regulamentados, como dados pessoais protegidos por leis de privacidade.

Não Escopo (o que não será feito deve ser esclarecido para alinhar expectativas):

Desenvolvimento de funcionalidades não relacionadas à integração e normalização de dados dos sistemas legados da Vivo.

- Implementação de recursos ou serviços não solicitados pelos stakeholders do projeto.
- Intervenção nos sistemas legados da Vivo além do necessário para a captura e integração de dados.
- Desenvolvimento de interfaces de usuário ou aplicativos finais para os clientes da Vivo.
- Suporte a sistemas legados de terceiros que não estejam diretamente relacionados aos objetivos do projeto.
- Implementação de processos de negócio ou lógica complexa dentro da plataforma, a menos que estritamente necessário para a integração e normalização de dados.

Oportunidades de Negócio identificadas (alto nível - preliminares):

1. **Melhoria da Experiência do Cliente:** A integração e normalização eficientes dos dados dos sistemas legados permitirão à Vivo oferecer uma experiência mais consistente e personalizada aos seus clientes, fornecendo informações precisas e atualizadas sobre produtos e serviços.
2. **Aumento da Eficiência Operacional:** Ao centralizar e homogeneizar os dados dos sistemas legados, a Vivo poderá simplificar seus processos internos, reduzindo a duplicação de esforços e melhorando a eficiência operacional.
3. **Tomada de Decisão Baseada em Dados:** Com acesso a dados mais confiáveis e integrados, os tomadores de decisão na Vivo poderão realizar análises mais precisas e embasadas, auxiliando na definição de estratégias de negócio e no planejamento de ações futuras.
4. **Desenvolvimento de Novos Produtos e Serviços:** A disponibilidade de dados integrados e normalizados pode abrir oportunidades para o desenvolvimento de novos produtos e serviços personalizados, alinhados às necessidades e preferências dos clientes da Vivo.
5. **Competitividade no Mercado:** Ao investir em uma plataforma robusta de integração e normalização de dados, a Vivo pode se destacar no mercado, oferecendo serviços mais inovadores e competitivos em comparação com seus concorrentes.

Posicionamento

Descrição do problema

O problema detectado	A Vivo, uma das principais empresas de telecomunicações do Brasil, enfrenta desafios significativos relacionados à integração e normalização dos dados provenientes de seus diversos sistemas
----------------------	---

	legados. Atualmente, esses sistemas operam de forma descentralizada, resultando em inconsistências, redundâncias e dificuldades na obtenção de uma visão unificada dos dados.
Afeta	Os sistemas legados da Vivo abrangem uma variedade de plataformas e tecnologias, cada uma com seu próprio formato e estrutura de dados. Isso dificulta a integração entre eles e gera obstáculos na disponibilização de informações consistentes para os clientes e partes interessadas internas.
O impacto é	A falta de padronização nos dados dos sistemas legados impacta negativamente na capacidade da Vivo de responder rapidamente às mudanças no mercado e às necessidades dos clientes. A inconsistência nos dados pode levar a decisões erradas e afetar a qualidade dos serviços oferecidos pela empresa.
A solução ideal seria	Portanto, a necessidade de uma solução abrangente que integre, normalize e forneça dados canônicos dos sistemas legados da Vivo é crucial para melhorar a eficiência operacional, a experiência do cliente e a competitividade no mercado de telecomunicações.

Visão Geral do Produto:

A solução consistirá em uma arquitetura de microsserviços baseada em tecnologias modernas, como Spring Boot, Spring Integration, Apache Kafka, MongoDB, entre outras. Essa arquitetura permitirá a captura assíncrona de dados dos sistemas legados, a normalização dos dados para um formato canônico e a disponibilização desses dados por meio de APIs seguras e eficientes.

Ao adotar essa solução, a Vivo poderá alcançar os seguintes benefícios:

1. **Consistência e Integridade dos Dados:** Os dados provenientes dos sistemas legados serão normalizados para um formato canônico, garantindo consistência e integridade em toda a plataforma.
2. **Agilidade e Resiliência:** A arquitetura de microsserviços permitirá uma abordagem modular e escalável, possibilitando rápida adaptação às mudanças nos sistemas legados e garantindo resiliência mesmo em caso de falhas.
3. **Baixa Latência e Alta Performance:** A utilização de tecnologias como Apache Kafka e MongoDB garantirá baixa latência e alta performance na entrega dos dados, atendendo aos requisitos de near-realtime e oferecendo uma experiência superior aos clientes.
4. **Facilidade de Manutenção e Evolução:** A solução será projetada com foco na manutenibilidade e extensibilidade, permitindo a fácil incorporação de novos requisitos e a evolução contínua da plataforma ao longo do tempo.

Com essa solução, a Vivo estará posicionada de forma competitiva no mercado, sendo capaz de oferecer serviços inovadores e de alta qualidade aos seus clientes, ao mesmo tempo em que otimiza seus processos internos e maximiza sua eficiência operacional.

Regras e/ou Restrições a considerar:

1. **Segurança dos Dados:** A solução deve garantir a segurança e a privacidade dos dados dos clientes da Vivo em conformidade com as regulamentações vigentes, como a Lei Geral de Proteção de Dados (LGPD).
2. **Alta Disponibilidade:** A plataforma deve ser altamente disponível para garantir o acesso contínuo aos dados, minimizando o tempo de inatividade e interrupções no serviço.
3. **Escalabilidade:** A solução deve ser capaz de escalar horizontalmente para lidar com aumentos na carga de trabalho e no volume de dados, garantindo o desempenho e a eficiência mesmo em períodos de pico.
4. **Monitoramento e Manutenção:** Deve ser implementado um sistema abrangente de monitoramento e alerta para acompanhar o desempenho da plataforma, identificar problemas rapidamente e tomar medidas corretivas.
5. **Compatibilidade com Padrões de Indústria:** A solução deve ser compatível com os padrões e práticas de desenvolvimento de software da indústria, garantindo a interoperabilidade e facilitando a integração com sistemas externos.
6. **Documentação Completa:** Todos os aspectos da solução, incluindo arquitetura, APIs, fluxos de dados e processos de manutenção, devem ser documentados de forma abrangente para facilitar a compreensão e a manutenção contínua do sistema.
7. **Custos e Orçamento:** Deve-se considerar o orçamento disponível para o desenvolvimento, implantação e manutenção da solução, buscando otimizar os custos sem comprometer a qualidade ou a eficácia do produto final.
8. **Compatibilidade com Sistemas Legados:** A solução deve ser capaz de integrar-se harmoniosamente com os sistemas legados existentes da Vivo, aproveitando e estendendo os investimentos tecnológicos já realizados pela empresa.

Requisitos Funcionais e Não funcionais da solução:

Nº	Nome	Descrição
RF001	Captura de Dados	Os sistemas legados devem ser capazes de enviar dados de forma assíncrona para a plataforma centralizada de captura de dados.
RF002	Normalização de Dados	Os dados recebidos devem ser normalizados para um formato canônico consistente antes de serem armazenados.
RF003	Armazenamento de Dados Canônicos	Os dados canônicos devem ser armazenados de forma escalável e de alta disponibilidade em um banco de dados adequado.
RF004	APIs de Entrega de Dados	APIs devem ser disponibilizadas para permitir o acesso seguro e eficiente aos dados canônicos por parte dos clientes.
RF005	Resiliência e Ressincronização	A solução deve ser resiliente a falhas nos sistemas legados, garantindo que os dados sejam capturados e entregues mesmo em caso de interrupções temporárias. Além disso, um mecanismo de ressincronização deve ser implementado para garantir a consistência dos dados após períodos de indisponibilidade.
RF006	Segurança	Mecanismos robustos de autenticação e autorização devem ser implementados para garantir que apenas usuários autorizados tenham acesso aos dados.
RF007	Monitoramento e Alerta	Deve ser implementado um sistema de monitoramento contínuo para acompanhar o desempenho da solução e detectar eventuais problemas em tempo real.

Nº	Nome	Descrição
RNF001	Desempenho	A solução deve ser otimizada para garantir tempos de resposta rápidos e baixa latência na entrega dos dados.
RNF002	Escalabilidade	A plataforma deve ser capaz de escalar horizontalmente para lidar com aumentos na carga de trabalho e no volume de dados.
RNF003	Disponibilidade	Alta disponibilidade é fundamental, garantindo que a solução esteja sempre disponível para os usuários, mesmo durante períodos de manutenção ou falhas.
RNF004	Segurança dos Dados	Mecanismos de criptografia, controle de acesso e monitoramento de atividades suspeitas devem ser implementados para proteger os dados sensíveis dos clientes.
RNF005	Confiabilidade	A solução deve ser altamente confiável, garantindo a integridade e a consistência dos dados em todas as circunstâncias.
RNF006	Manutenibilidade	O código-fonte e a infraestrutura da solução devem ser facilmente mantidos e atualizados para acompanhar as mudanças nos requisitos e nas tecnologias.
RNF007	Documentação	Todos os aspectos da solução devem ser documentados de forma abrangente para facilitar a compreensão e a manutenção contínua do sistema.

Os requisitos funcionais descrevem o comportamento da solução, ou seja, descrevem o trabalho que a solução deve realizar em itens, o que a solução faz, por exemplo, inserir, alterar dados, gerar relatório, consultar dados, relacionar dados, etc. Os requisitos não funcionais são os atributos de qualidade, por exemplo, velocidade, usabilidade, portabilidade, design, acessibilidade, etc.

Tecnologias e linguagens previstas para desenvolver a solução:

1. **Java 21:** Utilizado para desenvolvimento de microsserviços e aplicativos empresariais devido à sua robustez, portabilidade e vasta comunidade de desenvolvedores.
2. **Spring Boot:** Framework amplamente utilizado para desenvolvimento rápido e fácil de microsserviços em Java, oferecendo recursos abrangentes para integração, segurança, gerenciamento de dependências, entre outros.
3. **Spring Integration:** Utilizado para integração de sistemas legados, fornecendo suporte a diversas tecnologias de integração, como JDBC, FTP e HTTP.
4. **Apache Kafka:** Plataforma de streaming distribuída para publicação e subscrição de streams de registros, essencial para captura de dados em tempo real e comunicação entre microsserviços.
5. **Spring Kafka:** Extensão do Spring para integração fácil e eficiente de aplicativos Spring com o Kafka.
6. **MongoDB:** Banco de dados NoSQL escalável e flexível, adequado para armazenamento de dados estruturados em grande volume.
7. **Spring Data MongoDB:** Facilita o acesso e a manipulação de dados no MongoDB através do Spring.
8. **Spring MVC ou Spring WebFlux:** Utilizado para construção de APIs RESTful ou reativas para fornecer acesso aos dados canônicos.
9. **Spring Security:** Framework para implementação de autenticação e autorização em APIs Spring, garantindo a segurança dos dados.
10. **Apache Kafka Connect:** Estrutura para streaming de dados que permite a integração de sistemas externos, como bancos de dados, com o Apache Kafka de forma escalável e confiável.
11. **Debezium:** Conjunto de conectores Kafka para capturar mudanças nos bancos de dados, suportando uma variedade de bancos de dados, incluindo MySQL, PostgreSQL e MongoDB.
12. **Spring Boot Actuator:** Utilizado para monitoramento de aplicativos Spring Boot.
13. **Micrometer:** Utilizado para coleta de métricas e integração com sistemas de monitoramento, como Prometheus ou Grafana.

Essas tecnologias foram selecionadas com base na sua adequação para os requisitos do projeto, oferecendo um conjunto abrangente de ferramentas para desenvolvimento, integração, armazenamento e monitoramento da solução proposta.

Benefícios previstos ou esperados com a solução:

1. **Integração Eficiente:** A solução permitirá a integração eficiente de sistemas legados, possibilitando a captura de dados em tempo real e a normalização para um formato canônico.
2. **Atualização Contínua dos Dados:** Através do uso de ferramentas de streaming como o Apache Kafka, os dados serão sempre atualizados e prontos para serem consumidos pelos microsserviços.
3. **Escalabilidade:** A arquitetura de microsserviços proposta permite escalabilidade horizontal conforme a demanda cresce, garantindo que a solução possa lidar com volumes de dados cada vez maiores.
4. **Alta Disponibilidade e Resiliência:** O uso de tecnologias como o Apache Kafka e o Spring Boot contribui para a alta disponibilidade e resiliência da solução, minimizando o tempo de inatividade e garantindo a continuidade do serviço.
5. **Segurança:** A implementação de autenticação e autorização através do Spring Security garante que apenas usuários autorizados tenham acesso aos dados, protegendo a integridade e confidencialidade das informações.
6. **Monitoramento e Gerenciamento Simplificados:** Com o Spring Boot Actuator e o Micrometer, será possível monitorar e gerenciar a solução de forma simplificada, facilitando a identificação de problemas e a tomada de decisões.
7. **Agilidade e Facilidade de Desenvolvimento:** O uso de frameworks como o Spring Boot e o Spring Integration acelera o desenvolvimento de novos microsserviços e a integração com sistemas legados, permitindo uma resposta rápida às demandas do negócio.
8. **Documentação Automática:** Ferramentas como o Spring REST Docs e o Swagger/OpenAPI possibilitam a geração automática de documentação das APIs, facilitando a compreensão e o uso da solução por parte dos desenvolvedores e usuários.

Cronograma de execução:

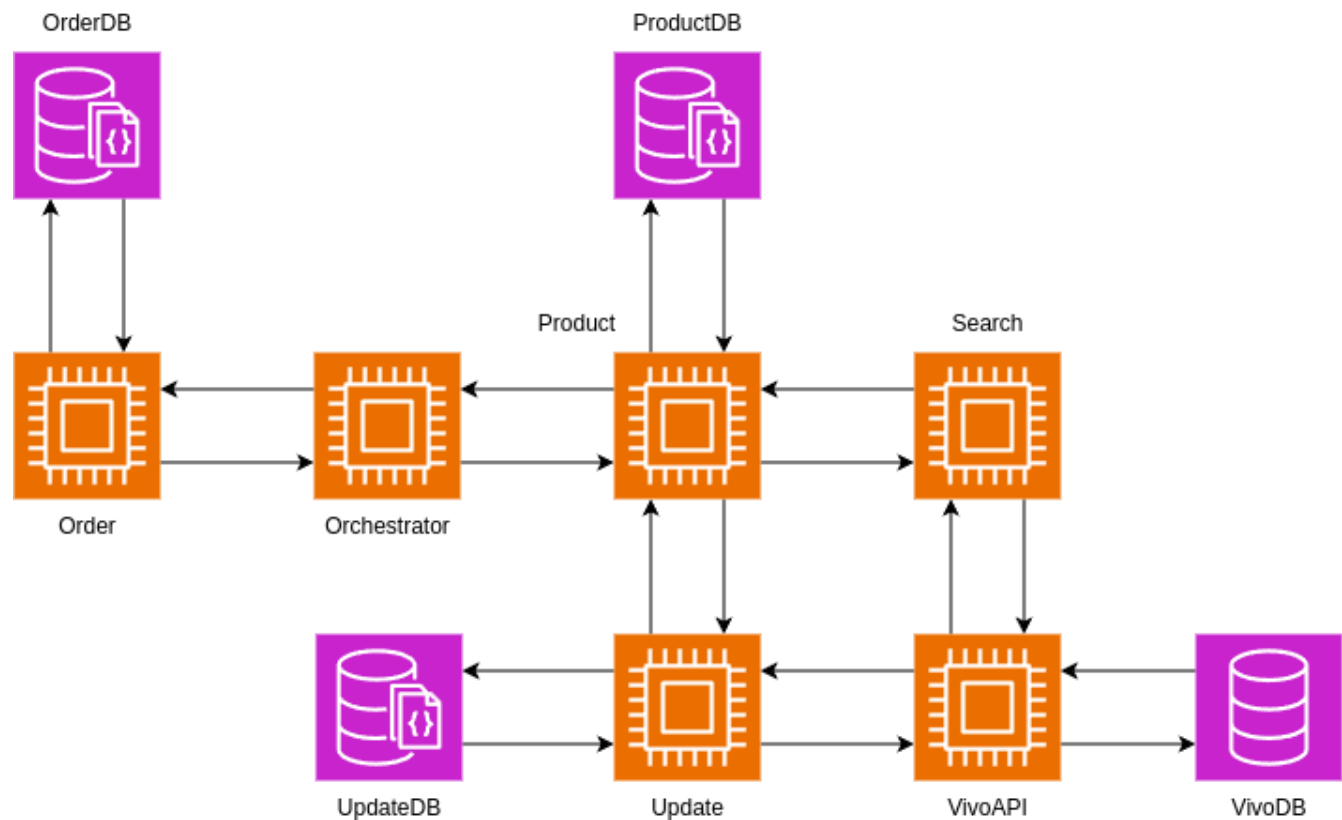
Atividade	Março	Abril
Análise de Requisitos e Planejamento	Esta fase envolve a compreensão detalhada dos requisitos do cliente, a definição do escopo do projeto, a identificação de tecnologias adequadas e a elaboração de um plano detalhado para a execução do projeto. (Duração: 2 semanas)	X
Desenvolvimento dos Microsserviços de Captura de Dados	X	Nesta etapa, serão desenvolvidos os microsserviços responsáveis por capturar os dados dos sistemas legados e enviá-los para o barramento de mensagens. Isso inclui a integração com os sistemas legados e a implementação de lógica de normalização dos dados. (Duração: 4

		semanas)
Atividade	Maio	
Implementação do Barramento de Mensagens	Aqui, será configurado e implementado o barramento de mensagens utilizando tecnologias como o Apache Kafka. Serão definidos os tópicos e as configurações necessárias para garantir a entrega confiável e assíncrona dos dados entre os microserviços. (Duração: 2 semanas)	
Desenvolvimento do Microserviço de Normalização	Será desenvolvido o microserviço responsável por receber os eventos do barramento de mensagens e normalizá-los para o formato canônico desejado. Isso inclui a implementação de regras de negócio para transformação de dados e a garantia de consistência e integridade dos dados. (Duração: 3 semanas)	
Atividade	Junho	
Implementação do Armazenamento de Dados Canônicos	Nesta etapa, será configurado e implementado o armazenamento de dados canônicos utilizando um banco de dados NoSQL como o MongoDB. Será definida a estrutura de dados e configurada a replicação em várias regiões para garantir resiliência e baixa latência. (Duração: 2 semanas)	
Desenvolvimento do Microserviço de Entrega de Dados	Será desenvolvido o microserviço responsável por expor APIs para acesso aos dados canônicos de forma segura e eficiente. Isso inclui a implementação de caches para melhorar a latência e garantir alta disponibilidade das APIs. (Duração: 3 semanas)	
Atividade	Agosto	Setembro
Implementação da Ressincronização de Dados	Será implementado o mecanismo de ressincronização de dados utilizando ferramentas como o Apache Kafka Connect e o Debezium. Serão configurados conectores para capturar alterações nos sistemas legados e replicá-las no Kafka, garantindo a ressincronização após a retomada dos sistemas legados. (Duração: 3 semanas)	X

Testes e Qualidade	X	Serão realizados testes de unidade, integração e sistema para garantir a qualidade e o funcionamento correto da solução. Será avaliada a cobertura de testes e serão feitas correções e melhorias conforme necessário. (Duração: 2 semanas)
Atividade	Outubro	
Documentação e Entrega	Será gerada documentação detalhada da arquitetura, APIs e processos de manutenção da solução. Serão preparados pacotes de entrega e realizada a entrega final ao cliente. (Duração: 1 semana)	

Este cronograma é uma estimativa inicial e pode ser ajustado conforme o andamento do projeto e as necessidades específicas do cliente. A colaboração estreita entre a equipe de desenvolvimento e o cliente é essencial para garantir o sucesso do projeto dentro do prazo e do orçamento estabelecidos.

Arquitetura da Solução



O cliente que deseja consultar informações deve primeiro se autenticar, fornecendo um token JWT de segurança. Nossas APIs são **stateless** (não mantêm estado).

Após a autenticação, o cliente é direcionado para o microserviço **Order**, que registra as requisições em um banco de dados não relacional. O microserviço **Order** então encaminha essas requisições para o **Orchestrator**, que coordena o processamento. O **Orchestrator** é responsável por definir quais APIs serão chamadas, conversando com todas, inclusive **Order**, por mensageria, como o processo será finalizado e o que deve ser feito em caso de falhas. Utilizamos o padrão **Saga Orquestrado**, centralizando a lógica de comunicação no **Orchestrator**, o que facilita a manutenção e a adição de novas APIs.

Uma vez processada a requisição, o **Orchestrator** distribui os dados para os microserviços responsáveis pela normalização e armazenamento (Produto e Search), conforme o arquivo de especificações **swagger.yml**. O fluxo segue da seguinte maneira:

1. O sistema consulta primeiro o microserviço **Produto**. Verificando se há aquele **Produto** em nossa base dados. Se houver, ele retorna o resultado ao **Order** que finaliza a Saga.
2. Caso não tenha o **Produto** em nossa base, ele segue a Saga para o Microserviço de **Search**, que envia uma requisição GET e o Token para a **API Legada**, buscando pelo Produto. Caso o produto exista, ele retorna os dados para o microserviço de **Produto**, que salva os dados em nossa base NoSQL de **Produto** e envia o ID para o microserviço de **Update** e retorna o processo para o **Orchestrator** e **Order**, encerrando a Saga.
3. Caso o Produto não seja encontrado em nenhuma base de dados, o **Orchestrator** retorna um erro HTTP adequado.

Os dados recuperados são então armazenados em nosso banco não relacional por um período de **35 dias**, para evitar duplicação de informações. Caso o cliente faça uma nova consulta dentro desse período, os dados já estarão disponíveis sem a necessidade de outra consulta ao sistema legado.

Para garantir consistência dos dados, iremos implementar um **CDC (Change Data Capture)** com ferramentas como **Debezium**. O microserviço é chamado de Update. Ele irá monitorar os IDs que estarão armazenados em seu banco de dados, garantindo assim que, sempre que houver uma alteração, os dados sejam atualizados automaticamente. Ele fará isso enviando uma requisição PUT para **Produto**. Ele não precisará de Token JWT pois foi colocado na lista de serviços que pode enviar requisições sem o mesmo.

Considerações e melhorias sugeridas pelos professores:

1) Lei Geral de Proteção de Dados (LGPD):

A Lei Geral de Proteção de Dados (LGPD) brasileira não define um tempo mínimo específico de armazenamento de dados de usuários em um banco de dados, seja ele NoSQL ou relacional. Em vez disso, a LGPD estabelece princípios que guiam o tratamento de dados, como os princípios da necessidade, adequação e finalidade. Esses princípios indicam que os dados pessoais devem ser armazenados apenas pelo tempo necessário para atingir os objetivos para os quais foram coletados.

Ponto-chave da LGPD sobre retenção de dados:

- Princípio da Necessidade: O armazenamento dos dados deve ser feito apenas enquanto for necessário para cumprir com a finalidade específica para a qual foram coletados.
- Princípio da Finalidade: O uso e armazenamento de dados devem ser limitados à finalidade específica informada ao titular no momento da coleta.
- Princípio da Adequação: O armazenamento e tratamento de dados devem ser adequados ao propósito original, e excessos ou coleta desnecessária são proibidos.

Regras específicas:

Embora a LGPD não defina um período mínimo obrigatório, o tempo de retenção de dados pode ser regido por outras legislações ou regulamentos setoriais. Por exemplo:

- **Tributário ou financeiro:** Algumas leis exigem que certos dados, como informações fiscais, sejam armazenados por até 5 ou 10 anos, dependendo da regulamentação.
- **Relacionamento contratual:** Em algumas circunstâncias, dados podem precisar ser retidos até que se cumpram obrigações contratuais ou legais.

Eliminação de dados:

Quando os dados não forem mais necessários para a finalidade para a qual foram coletados, devem ser eliminados, salvo se houver uma obrigação legal para mantê-los por mais tempo.

Portanto, o armazenamento de dados deve ser justificado pela necessidade legal, contratual ou de operação do serviço, não havendo uma exigência mínima expressa na LGPD para a retenção de dados em bancos NoSQL ou de outro tipo.

2) Busca pelo mais consultado:

Para implementarmos uma funcionalidade que permita buscar pelo "produto mais consultado" no microserviço de **Produtos**, vamos adotar a seguinte estratégia:

1. Adicionar um campo de contagem de acessos:

No banco de dados, vamos adicionar um campo chamado **consultas**, no qual você irá armazenar o número de vezes que cada produto foi consultado. Toda vez que um produto for acessado, esse contador será incrementado.

2. Buscar o produto mais consultado:

Iremos criar uma nova API para buscar o produto mais consultado. Essa API executará uma query no banco de dados para ordenar os produtos pelo campo **consultas** de forma decrescente e retornar o produto com o maior valor.

3. Implementação do API:

Iremos configurar um endpoint, como **GET /produtos/mais-consultado**, que retorna o produto com o maior número de consultas:

5. Monitoramento e Relatórios:

Pensamos também em armazenar e monitorar essas consultas ao longo do tempo, utilizando ferramentas como **Elasticsearch** para indexar esses acessos e, posteriormente, gerar relatórios e dashboards mais detalhados.

Considerações:

- **Performance:** Como é um sistema de alta escala, estamos considerando armazenar esse contador de consultas em um banco de dados otimizado para leitura e escrita rápida, como Redis.

- **Persistência de dados:** Se necessário, vamos armazenar esses dados em uma tabela/camada de cache separada, consolidando-os periodicamente para não sobrecarregar a camada principal de banco de dados.

3) Migração de sistemas legados:

Nossa aplicação utiliza o banco de dados legado apenas para fins de cache e otimização de performance nas requisições. **Não temos a intenção de migrar ou substituir o banco de dados legado;** ele continuará sendo a fonte principal dos dados.

O que fazemos é, quando um dado não é encontrado em nosso banco de dados NoSQL, realizamos uma consulta ao banco de dados legado para recuperar essas informações. Esses dados são então armazenados em cache no nosso banco por um período limitado (30 dias, por exemplo), de forma que, se houver uma nova consulta a esse dado dentro desse período, podemos retornar a resposta de forma mais rápida, sem necessidade de acessar o serviço legado novamente. Isso reduz o tempo de resposta e a carga no banco legado, sem comprometê-lo.

Em resumo, o banco de dados da nossa aplicação é usado para **armazenamento temporário (cache)**, com o objetivo de aumentar a velocidade das requisições, mas **o serviço legado permanece intocado e sendo a fonte de verdade para os dados originais.**

4) Autenticação JWT:

1. Como funciona o processo de autenticação usando JWT?

a. Autenticação (Login)

1. **Cliente envia credenciais:** O cliente (aplicativo ou usuário) envia suas credenciais (como email e senha) para a API de autenticação.
2. **API verifica as credenciais:** O servidor valida essas credenciais comparando com os dados armazenados em seu banco de dados.
3. **Geração do JWT:** Se as credenciais forem válidas, a API de autenticação gera um **JWT** contendo as informações do usuário no payload e o devolve ao cliente.
4. **Cliente armazena o JWT:** O cliente armazena esse token, geralmente no localStorage ou nos cookies (para aplicações web).

b. Autorização (Uso de outras APIs)

1. **Cliente faz uma requisição com o JWT:** Para acessar outros endpoints (como `/order`, `/produto`), o cliente inclui o **JWT** no cabeçalho da requisição HTTP.
2. **API valida o JWT:** A API que recebe a requisição valida o token, verificando sua assinatura e, opcionalmente, checando claims como expiração (`exp`) ou escopos de permissão.
3. **Acesso concedido:** Se o token for válido, a API processa a requisição e retorna a resposta desejada. Caso contrário, retorna um erro de autorização (401 Unauthorized).

c. Fluxo de Requisição usando JWT:

- **Login:** O cliente solicita um token (fornecendo credenciais) na API de autenticação.
- **Token recebido:** O servidor responde com um JWT.
- **Envio do token:** O cliente usa esse JWT para acessar APIs subsequentes, passando

o token no cabeçalho de autorização.

- **Validação do token:** As APIs verificam o JWT antes de processar a requisição.

2. Vantagens de usar JWT em APIs stateless

- **Descentralizado:** Diferente de sessões, o JWT não precisa ser armazenado no servidor. Todas as informações necessárias estão dentro do token, permitindo que o servidor seja stateless.
- **Segurança:** A assinatura garante que o token não foi modificado. Qualquer alteração no payload invalida o token.
- **Eficiência:** Como os dados do usuário estão no token, o servidor não precisa consultar o banco de dados ou outro armazenamento para validar a identidade do usuário a cada requisição.
- **Escalabilidade:** Permite que APIs distribuídas, hospedadas em diferentes servidores, autenticem requisições de forma independente, já que o estado não é armazenado no servidor.

3. Segurança com JWT

- **Criptografia da assinatura:** A assinatura do JWT deve ser gerada usando uma chave secreta que é mantida segura no servidor (ou chave privada, em casos de criptografia assimétrica).
- **Claims sensíveis:** Não armazene dados sensíveis no payload do JWT (como senhas ou informações bancárias), já que ele pode ser decodificado por qualquer pessoa que tenha acesso ao token. A assinatura garante a integridade, mas não o sigilo do conteúdo.
- **Expiração do token:** Sempre configure uma data de expiração (**exp**) para o token, limitando sua validade a um curto período de tempo.
- **Refresh Token:** Para sessões mais longas, utilize uma estratégia de **refresh token**. O refresh token permite que o cliente obtenha um novo JWT quando o token original expirar, sem precisar refazer o login.

4. Conclusão

A autenticação JWT é uma solução ideal para APIs stateless, como no seu sistema, onde cada requisição é independente. Ao usar JWT, você elimina a necessidade de manter sessões no servidor, tornando o sistema mais escalável e distribuído. Com a configuração correta de segurança e expiração, você garante a integridade e a segurança do processo de autenticação.

6) Utilização de Docker + Kubernetes:

O uso de **Docker** e **Kubernetes** na aplicação oferece uma abordagem eficiente para o desenvolvimento, implantação e gerenciamento de seus microserviços em um ambiente de produção.

1. Docker: Containerização dos Microserviços

Docker é uma plataforma de containerização que permite empacotar sua aplicação, juntamente com todas as suas dependências (bibliotecas, variáveis de ambiente, etc.), em uma unidade padronizada chamada **container**. Isso garante que a aplicação funcione da mesma maneira em diferentes ambientes (desenvolvimento, testes e produção), evitando problemas de compatibilidade.

Vantagens de usar Docker:

- **Isolamento:** Cada microserviço é isolado em um container separado, garantindo que mudanças em um microserviço não afetem outros.
- **Portabilidade:** Como o container inclui todas as dependências, ele pode ser executado em qualquer lugar onde o Docker esteja instalado.
- **Reprodutibilidade:** Você pode garantir que o mesmo código, rodando no mesmo ambiente, se comporte de forma idêntica em diferentes máquinas.

2. Kubernetes: Orquestração dos Containers

Kubernetes (K8s) é uma plataforma de orquestração de containers que permite gerenciar, escalar e monitorar seus containers Docker em ambientes de produção distribuídos. Ele é particularmente útil em aplicações de microserviços, onde diversos containers precisam trabalhar em conjunto de maneira eficiente.

O que o Kubernetes faz:

- **Gerenciamento de containers:** Ele lida com o ciclo de vida dos containers, garantindo que estejam sempre rodando de forma estável e que possam ser automaticamente reiniciados em caso de falha.
- **Escalabilidade automática:** Se houver aumento de tráfego, o Kubernetes pode aumentar o número de réplicas de containers para suportar a carga adicional.
- **Balanceamento de carga:** Kubernetes distribui automaticamente o tráfego de rede entre os containers que estão rodando.
- **Monitoramento e autocorreção:** Ele monitora o estado dos containers e os reinicia ou substitui automaticamente se detecta falhas ou inatividade.

Componentes do Kubernetes:

- **Pods:** São as menores unidades de execução no Kubernetes e podem conter um ou mais containers. Cada microserviço será executado dentro de um pod.
- **Deployments:** Controlam o ciclo de vida dos pods, garantindo que o número correto de réplicas esteja sempre rodando.
- **Services:** Exponibilizam os pods para fora do cluster, permitindo o acesso aos microserviços.
- **ConfigMaps e Secrets:** Armazenam configurações e dados sensíveis como senhas, que podem ser injetados nos containers.

3. Fluxo de Trabalho com Docker e Kubernetes

a. Desenvolvimento com Docker:

1. Cada microserviço é empacotado em um container Docker.
2. O ambiente local (desenvolvimento) pode rodar esses containers para testes antes de subir para produção.
3. Docker Compose pode ser usado para rodar vários microserviços juntos no ambiente local.

b. Implantação em Produção com Kubernetes:

1. As imagens Docker são enviadas para um **Docker Registry** (como o Docker Hub ou um registro privado).
2. O **Kubernetes** puxa essas imagens do registro e as executa dentro de **Pods**.
3. O **Kubernetes** gerencia o balanceamento de carga, o monitoramento e a escalabilidade dos containers.
4. Em caso de falhas, o Kubernetes recria automaticamente os pods para garantir a alta disponibilidade.

4. Integração de Docker e Kubernetes com sua aplicação

Na nossa aplicação de microserviços, o Docker facilita o empacotamento e execução dos serviços individualmente, enquanto o Kubernetes coordena e orquestra esses containers em produção, garantindo que seu sistema seja escalável, resiliente e de fácil manutenção.

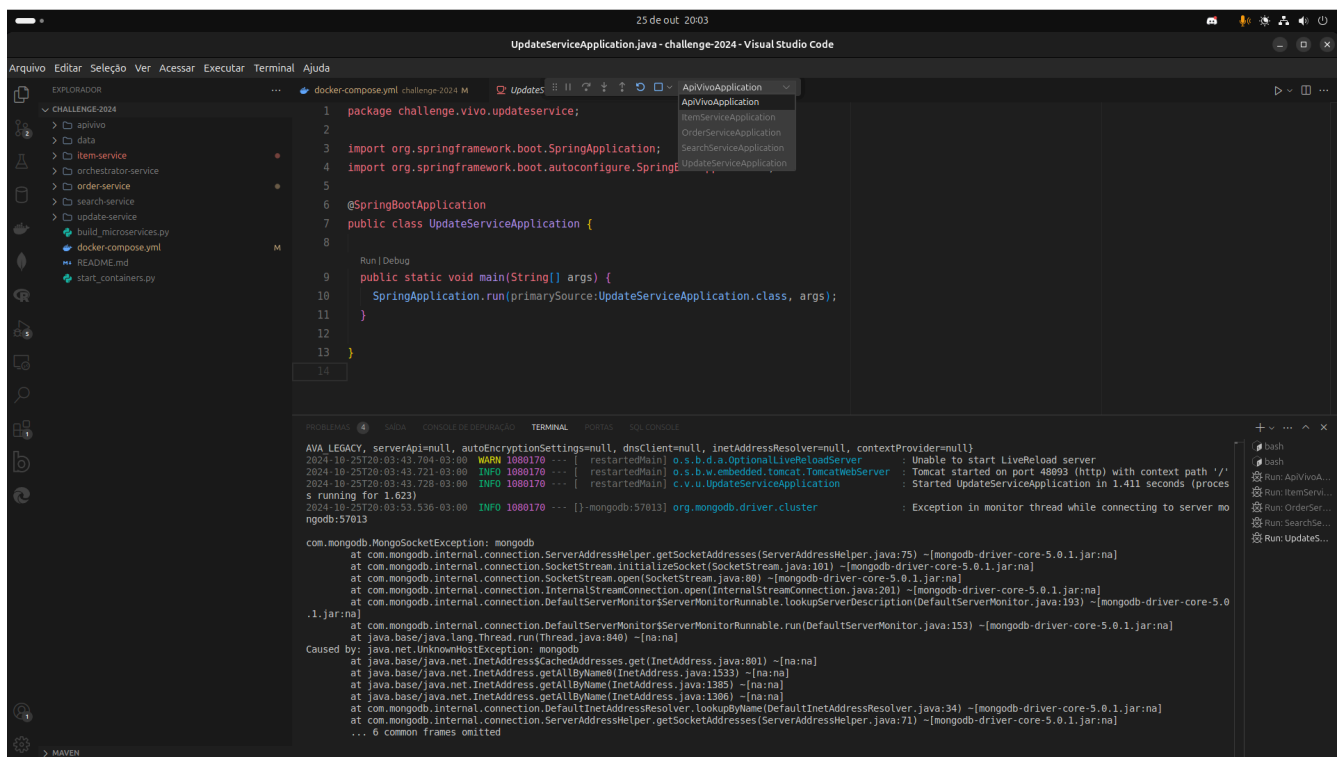
Por exemplo:

- O **Orchestrator** e os microserviços de **Order**, **Produto**, **Descrição**, e **Preço** seriam executados em containers Docker.
- O Kubernetes gerencia a comunicação entre esses containers, garantindo que as APIs estejam sempre disponíveis e que, em caso de falha, as operações continuem sem interrupções.
- Kubernetes pode escalar automaticamente os microserviços baseados no uso, garantindo que as APIs lidem com altos volumes de requisições sem sobrecarregar o sistema.

Conclusão

O uso do **Docker** permite que cada microserviço seja containerizado de forma eficiente e independente, enquanto o **Kubernetes** gerencia a execução desses containers em um ambiente de produção. Isso proporciona uma aplicação escalável, resiliente e fácil de gerenciar, especialmente em um cenário de microserviços como o seu.

Aqui estão algumas imagens do funcionamento do App:



The screenshot displays the Visual Studio Code interface with a Java Spring Boot application open. The Explorer on the left shows the project structure, including a 'challenge-2024' directory with sub-projects like 'apivivo', 'data', 'item-service', 'orchestrator-service', 'order-service', 'search-service', and 'update-service'. The main editor shows the 'UpdateServiceApplication.java' file, which is a Spring Boot application. The code includes package declarations, imports for Spring Boot and Spring Cloud, and a main method that runs the application. The terminal at the bottom shows the output of the application, including warnings about the LiveReload server and information about the application starting on port 48893. The terminal also shows the output of the 'mvn' command, indicating that the application is running successfully.

```
25 de out. 2003
UpdateServiceApplication.java - challenge-2024 - Visual Studio Code

Arquivo Editar Seleção Ver Acessar Executar Terminal Ajuda

EXPLORADOR
challenge-2024
├── apivivo
├── data
├── item-service
├── orchestrator-service
├── order-service
├── search-service
├── update-service
├── build_microservices.py
├── docker-compose.yml
├── README.md
└── start_containers.py

UpdateServiceApplication.java
1 package challenge.vivo.update.service;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class UpdateServiceApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(primarySource: UpdateServiceApplication.class, args);
11     }
12 }
13
14

TERMINAL
2024-10-25T20:03:43:704-03:00 WARN 1680178 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : Unable to start LiveReload server
2024-10-25T20:03:43:721-03:00 INFO 1680178 --- [ restartedMain] o.s.b.w.e.tomcat.TomcatWebServer : Tomcat started on port 48893 (http) with context path '/'
2024-10-25T20:03:43:728-03:00 INFO 1680178 --- [ restartedMain] c.v.u.UpdateServiceApplication : Started UpdateServiceApplication in 1.411 seconds (process running for 1.623)
2024-10-25T20:03:53:536-03:00 INFO 1680178 --- [ mongodb:57013] org.mongodb.driver.cluster : Exception in monitor thread while connecting to server mongodb:57013
com.mongodb.MongoSocketException: mongodb
    at com.mongodb.internal.connection.ServerAddressHelper.getSocketAddresses(ServerAddressHelper.java:75) ~[mongodb-driver-core-5.0.1.jar:na]
    at com.mongodb.internal.connection.SocketStream.initializeSocket(SocketStream.java:101) ~[mongodb-driver-core-5.0.1.jar:na]
    at com.mongodb.internal.connection.SocketStream.open(SocketStream.java:80) ~[mongodb-driver-core-5.0.1.jar:na]
    at com.mongodb.internal.connection.InternalStreamConnection.open(InternalStreamConnection.java:201) ~[mongodb-driver-core-5.0.1.jar:na]
    at com.mongodb.internal.connection.DefaultServerMonitor$ServerMonitorRunnable.lookupServerDescription(DefaultServerMonitor.java:193) ~[mongodb-driver-core-5.0.1.jar:na]
    at com.mongodb.internal.connection.DefaultServerMonitor$ServerMonitorRunnable.run(DefaultServerMonitor.java:153) ~[mongodb-driver-core-5.0.1.jar:na]
    at java.base/java.lang.Thread.run(Thread.java:840) ~[na:na]
Caused by: java.net.UnknownHostException: mongodb
    at java.base/java.net.InetAddressCachedAddresses.get(InetAddress.java:801) ~[na:na]
    at java.base/java.net.InetAddress.getAllByName(InetAddress.java:1533) ~[na:na]
    at java.base/java.net.InetAddress.getAllByName(InetAddress.java:1385) ~[na:na]
    at java.base/java.net.InetAddress.getAllByName(InetAddress.java:1366) ~[na:na]
    at com.mongodb.internal.connection.DefaultInetAddressResolver.lookupByName(DefaultInetAddressResolver.java:34) ~[mongodb-driver-core-5.0.1.jar:na]
    at com.mongodb.internal.connection.ServerAddressHelper.getSocketAddresses(ServerAddressHelper.java:71) ~[mongodb-driver-core-5.0.1.jar:na]
    ... 6 common frames omitted
```

