PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
NÚCLEO DE EDUCAÇÃO A DISTÂNCIA
Pós-graduação *Lato Sensu* em Arquitetura de Software Distribuído

Rian de Vasconcelos Machado

EVENT DRIVEN COM MICROSERVICE USANDO COMMAND QUERY RESPONSABILITY SEGREGATION

Recife
2021

RIAN DE VASCONCELOS MACHADO

EVENT DRIVEN E MICROSERVICE USANDO COMMAND QUERY RESPONSABILITY SEGREGATION

Trabalho de Conclusão de Curso de Especialização em Arquitetura de Software Distribuído como requisito parcial à obtenção do título de especialista.

Orientador(a): **Luiz Alberto Ferreira**

Recife

2021

*Dedico este trabalho ao amigo **Luciano Gomes da Silva**. Pessoa que tenho enorme respeito e gratidão profissional. Encontrar profissionais diferenciados, disruptivos e que elevam o potencial dos seus colegas de trabalho é cada vez mais raro. Fica registrado neste trabalho de especialização meus sinceros agradecimentos.*

AGRADECIMENTOS

Agradeço aos meus filhos **Pedro Oliveira de Vasconcelos**, **Júlia Oliveira de Vasconcelos, Ana Adélia Oliveira e minha mãe Maria Nair de Vasconcelos.**

RESUMO

No mundo competitivo no qual vivemos falar de qualidade é mais que obrigação principalmente quando queremos manter nossos produtos/serviços competindo no mercado. Para isso é necessário ter uma gestão da qualidade direcionada aos dados gerados pelas empresas. Quando não usufruímos desses dados tão valiosos muito provavelmente teremos apenas uma gestão da qualidade automatizada por processos computacionais.

O Easy Quality através de sua suíte de ferramentas e dados integrados permite às empresas fazerem gestão da qualidade de acordo com a estratégia de negócio que vivem no momento. Módulos desenhados no formato de escada de produtos, nos permite incrementar ou remover módulos de software levando assim aos clientes melhor experiência na sua jornada, pois a transformação digital que estamos vivendo impõem um time to marketing agressivo. A partir disso foi desenhado uma arquitetura de software que possibilitasse a escalabilidade, extensibilidade, disponibilidade, performance, tolerância a falha e resiliência, tudo isso implementado em nuvem sobre tecnologias de orquestração de containers.

Palavras-chave: Micro Serviço, Event Driven, Atributos de Qualidade, Kubernetes,

SUMÁRIO

1. Objetivos do trabalho

O objetivo deste trabalho é descrever o projeto arquitetural da aplicação **Easy Quality**. Aplicação que faz a **gestão da qualidade de processos organizacionais**, detalhando quais componentes de software são necessários e como eles interagem para atender as estratégias de negócio. Os objetivos específicos são:

- Redução de Riscos
- Resiliência e escalabilidade aos negócios
- Event Source
- Redução do Lead Time e Micro Serviços
- Comunicação assíncrona
- Command Query Responsibility Segregation

2. Descrição geral da solução

Em uma orquestra sinfônica cada instrumento dentro de suas características executa de forma bem definida trechos musicais. A execução de todos esses instrumentos ao mesmo tempo levará aos nossos ouvidos uma bela sinfonia. É uma boa analogia para começar a falar da proposta arquitetural introduzida na solução **Easy Quality**. Nossa arquitetura trabalha com peças de software pequenas e bem definidas em contextos de negócio o que possibilita "plugar" cada uma das peças continuamente sem causar efeitos colaterais em features do produto que já estão em funcionamento. Essas "pequenas peças" quando utilizadas de forma orquestrada dentro de contextos negocias podemos definir como Arquitetura de Micro Serviços.

A natureza da comunicação entre micro serviços nos coloca diante de características síncrona e assíncrona. Quando optamos por comunicação síncrona geralmente o caminho mais utilizado é a comunicação utilizando APIs baseadas em REST. Manter um ecossistema com vários serviços independentes comunicado - se uns com os outros trará pontos de atenção a serem observados. Como por exemplo: disponibilidade do serviço requisitado, segurança na comunicação e descoberta de novos serviços são assuntos comuns quando adotamos arquitetura de micro serviços.

A complexidade na comunicação entre os serviços é um assunto bastante discutido e que geralmente acaba sendo implementado na camada da aplicação. Levando ainda mais responsabilidades ao software além das implementações de negócio. Uma forma de minimizarmos a complexidade da comunicação é delegar esse aspecto para um barramento de mensagens assíncrono.

O Event Driven Architecture é um estilo arquitetural no qual a comunicação entre os componentes ocorre usando Streams de Eventos propondo-se a realizar notificações de mudança de estado(Data Changes) da aplicação e dos componentes(microsserviços) promovendo o baixo acoplamento. Esta interação ocorre através de um barramento de eventos que entrega mensagens a todos os componentes inscritos em determinados acontecimentos.

"Event-driven e a assincronicidade que este estilo de arquitetura promove, refere-se a otimização de tempo, onde não temos bloqueio de recursos para o atendimento das requisições enviadas, como tradicionalmente acontece em microsserviços que utilizam natureza de comunicação síncrona eliminando a ociosidade que encontramos quando implementamos chamadas tradicionais bloqueantes". **(Marcelo M. Gonçalves, 2011, Arquitetura Micro Serviço)**

CQRS é outro Design Patterns usado na arquitetura de microsserviços que separa em camadas distintas operações de leitura e escrita. Dessa forma delegamos responsabilidades coerentes às características de cada camada. Com isso podemos destacar algumas vantagens do CQRS integrado ao fornecimento de eventos e microsserviços.

- Microsserviços com banco de dados separados para atender leitura e escrita.
- Manter dados históricos de auditoria para análises com a implementação do event sourcing.
- Modelos de serviços separados para operações de leitura e inserção.
- A carga de solicitação pode ser distribuída entre as operações de leitura e inserção.
- O modelo de leitura ou DTO não precisa ter todos os campos como um modelo de comando, e um modelo de leitura pode ter campos obrigatórios pela visualização do cliente, o que pode economizar a capacidade do armazenamento de leitura e também reduzir tráfego.

2.1. Apresentação do problema

Os produtos de mercados avaliados apresentaram grandes peças de software com alto acoplamento tornando inviável entregar valor aos nossos clientes quando pensamos lead time pequeno, o que torna a estratégia de marketing e visibilidade da jornada do cliente lenta. Esse cenário não nos permite modelar o negócio e oferecer aos clientes experiências baseada em uma escada de produtos e diante desse cenário ficamos cada vez mais distantes do mercado competitivo. Quando falamos em processo da gestão da qualidade corporativo encontramos uma grande quantidade de processos segregado, interligados e que precisam ser orquestrados baseados na estratégia de negócio de vários produtos ofertados e agregados de acordo com real necessidade dos clientes.

2.2. Descrição geral do software

A finalidade do **Easy Quality** é levar ao cliente uma suíte para gestão de qualidade, conduzindo sua jornada através de vários módulos interligados. Permitir que cada cliente escolha os módulos de acordo com sua real jornada de negócio é a principal característica da nossa suíte de ferramentas, o que possibilita também ao cliente "degustar" da escada de produtos ofertada. Tendo como pilar as features **Conformidades** e **Plano de ação** o **Easy Quality** oferecerá também módulos opcionais como **Gestão de Indicadores e Gestão de documentos**.

3. Definição conceitual da solução

Na sequência apresentarei: requisitos funcionais, não funcionais, restrições e mecanismos arquiteturais considerados.

3.1. Requisitos Funcionais

Use Case Manter Conformidades	
Objetivo	Permite ao usuário criar, editar e consultar questionários modelo para serem aplicados em inspeções de conformidade.
Requisitos	Cadastrar questionários modelos, Listar questionários modelos, Atualização de questionários modelos, Criação de inspeção e Listar de inspeções
Prioridade	Alta

Use Case Criar Plano de Ação	
Objetivo	Permite ao usuário criar plano de ação a partir de um questionário respondido e que não atingiu o nível mínimo de conformidade aceito.
Requisitos	Cria Issue de erro e Cria Check List
Prioridade	Alta

Casos de uso Manter Conformidades:

- **Requisito** Cadastrar questionários modelos

Permite criar um conjunto de perguntas atribuindo um título. Dessa forma teremos o modelo de questionário pronto para ser aplicado às inspeções.

- **Requisito** Listar questionários modelos

Permite listar todos os questionários cadastrados

- **Requisito** Atualização de questionários modelos

Permite realizar alteração do questionário a partir de um identificador.

- **Requisito** Criação de inspeção

Permite criar inspeções de conformidade a partir de um modelo de questionário. Sendo assim é necessário informar um identificador do questionário modelo.

- **Requisito** Listar de inspeções

Permite listar inspeções por data de criação, ordenando pelas mais recentes.

Casos de uso Criar Plano de Ação:

- **Requisito** Cria Issue de erro

A partir das inspeções que não atingiram conformidade acima de 70% criar **issue de erro** com níveis de prioridade, ***ALTA, MÉDIA ou BAIXA*** associando a inspeção que originou a não conformidade.

- **Requisito** Cria Check List

A partir da **issue de erro** criar check list com data limite e as possibilidades de status **FEITO, A FAZER ou CANCELADO**

- **Requisito** Atualiza plano de ação

Permite atualizar as informações do check list exceto remover ou adicionar novos ítens.

3.2 Requisitos Não-Funcionais

- Dada um conjunto de solicitações do usuário o gerenciador de container (Kubernets) deverá monitorar os pods e realizar o horizontal pod autoscaler caso a aplicação atinja **oitenta por cento** de processamento.
- Dada solicitação realizada pelo usuário, o sistema deve responder em até três segundos.
- Dada solicitação realizada pelo usuário, o sistema estará sempre disponível para resposta.
- Dada solicitação realizada pelo usuário, o sistema deverá responder através de um gateway de borda.
- Dada solicitação realizada pelo usuário, o sistema criará eventos que serão capturados por um listener. Por fim, os dados da solicitação deverão ser armazenados em tópico Kafka no formato JSON.
- Dada qualquer mensagem gerada pelo sistema, o Kafka deverá ser a plataforma de mensagem utilizada a partir de algum padrão baseado em event bus.
- Dada solicitação de consulta o sistema direciona para o módulo de leitura.
- Dada solicitação de deleção, atualização ou inserção o sistema direciona para o módulo de escrita.

3.3. Restrições Arquiteturais

- O sistema deverá ser implementado com base em tecnologias de containers.
- O sistema deverá rodar em nuvem de forma agnóstica
- O sistema não poderá apresentar comunicação síncrona entre seus micro serviços

3.4. Mecanismos Arquiteturais

- **Gestão de Eventos**

  Suportar a utilização de eventos assíncronos dentro do Easy Quality.

  Mecanismo de Análise: Condições internas do sistema são observadas

  Mecanismo de Design: O pattern CRQS é escolhido

  Mecanismos de Implementação: Implementação das camadas **command e query** baseando-se na arquitetura hexagonal. Listener devem ser implementados com Spring Api Async ou CompletableFuture da linguagem Java o que possibilitará a gestão de eventos assíncronos.

- **Processamento de Eventos**

  Suportar processamento de eventos assíncronos dentro do Easy Quality.

  Mecanismo de Análise: Os micro serviços farão comunicação assíncrona

  Mecanismo de Design: Padrão baseado em troca de mensagem(publish e subscribe) possibilitará a interoperabilidade com resiliência.

  Mecanismos de Implementação: O Spring Kafka possibilitará a implementação de produtores e consumidores baseados em configurações nativas do Apache Kafka.

- **Escalabilidade**

  Suportar o aumento de usuários sem diminuir o tempo de resposta da aplicação Easy Quality.

  Mecanismo de Análise: Monitoramento de aplicação.

  Mecanismo de Design: Desenvolvimento dos micro serviços utilizando soluções container.

  Mecanismos de Implementação: Kubernetes através do servidor de métricas analisará CPU/Memória para realizar o auto scaling da aplicação automaticamente.

- **Persistência de dados**

  Suportar persistência de dados NoSQL

  Mecanismo de Análise: Os dados de cada micro serviço serão persistido

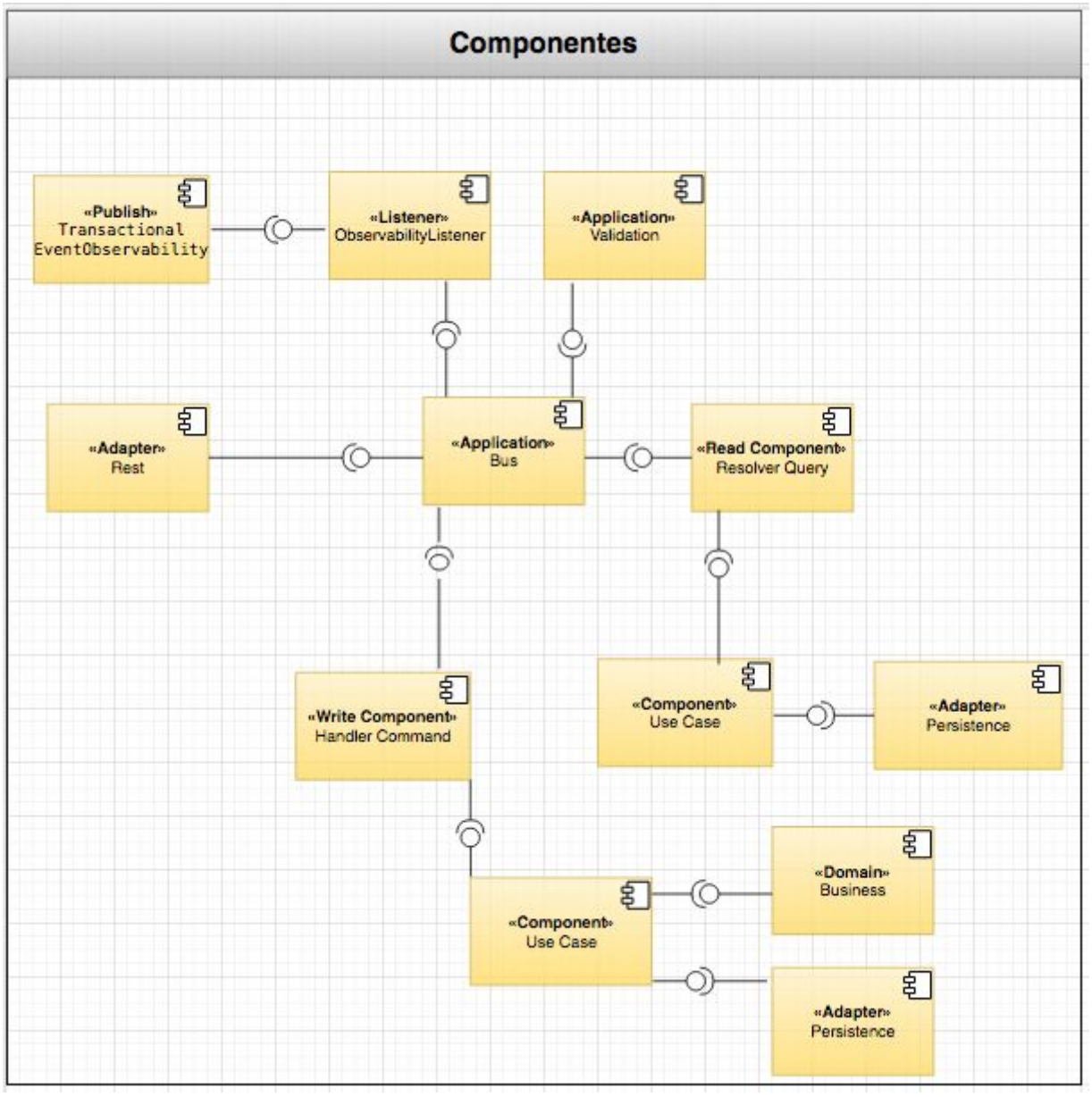
  Mecanismo de Design: Os dados serão armazenados em banco de dados NoSQL orientados a documentos.

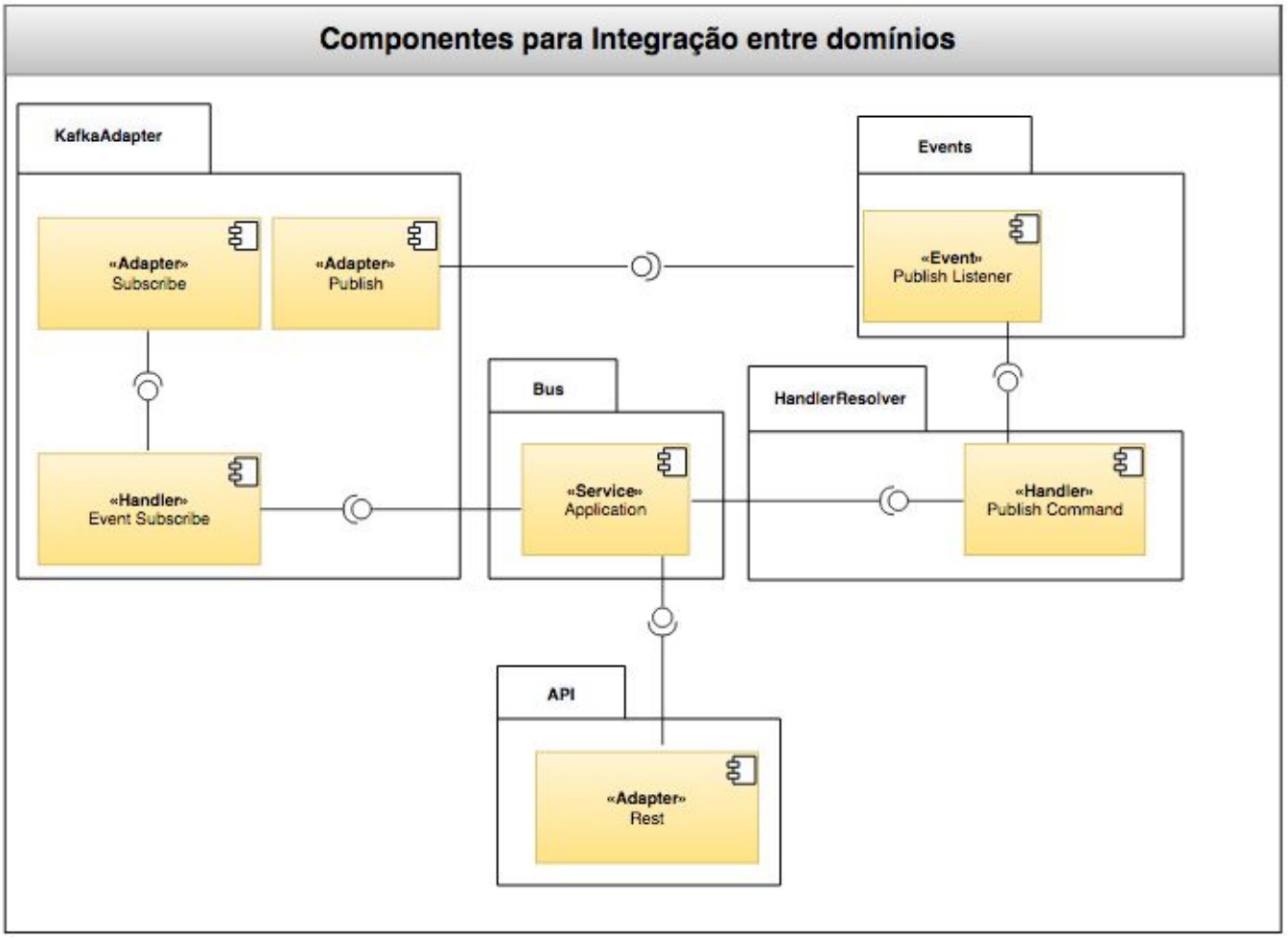
  Mecanismos de Implementação: O MongoDB por utilizar estrutura de documentos semelhantes a JSON utilizaremos para persistir todos os dados do Easy Quality.

4. Modelagem e projeto arquitetural

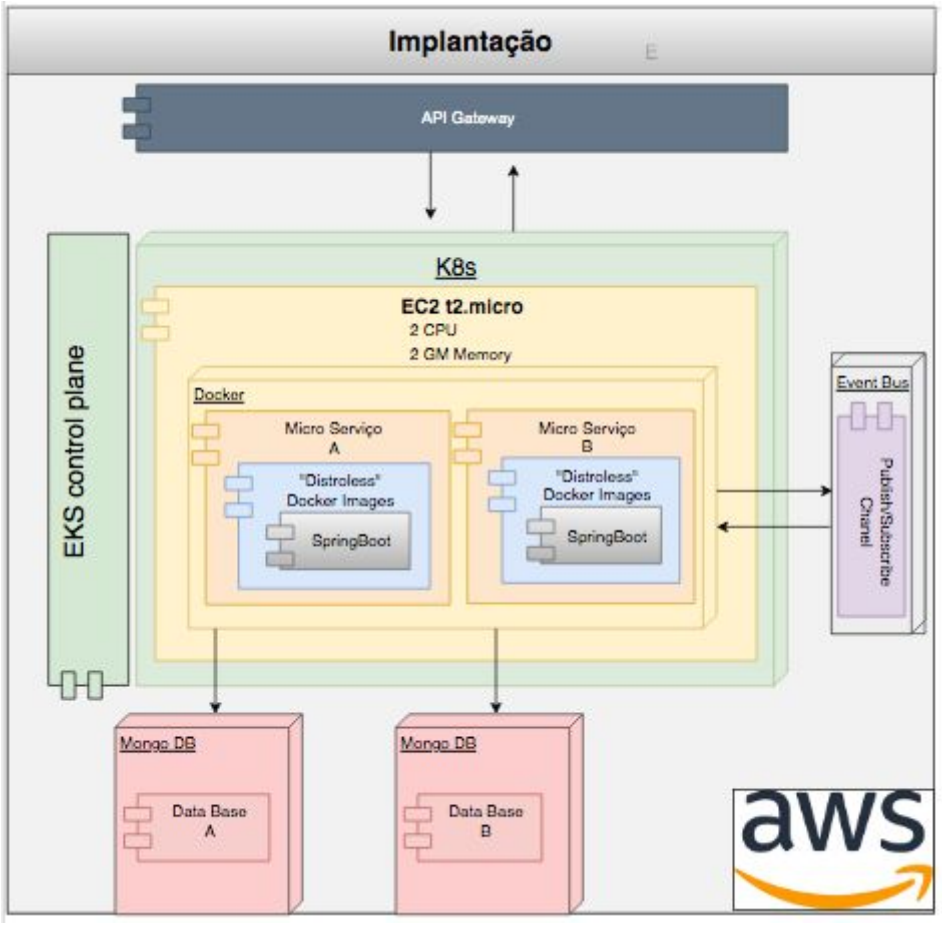
Na sequência serão apresentados os diagramas que permitem entender a arquitetura da aplicação, detalhando-a suficientemente para viabilizar sua implementação.

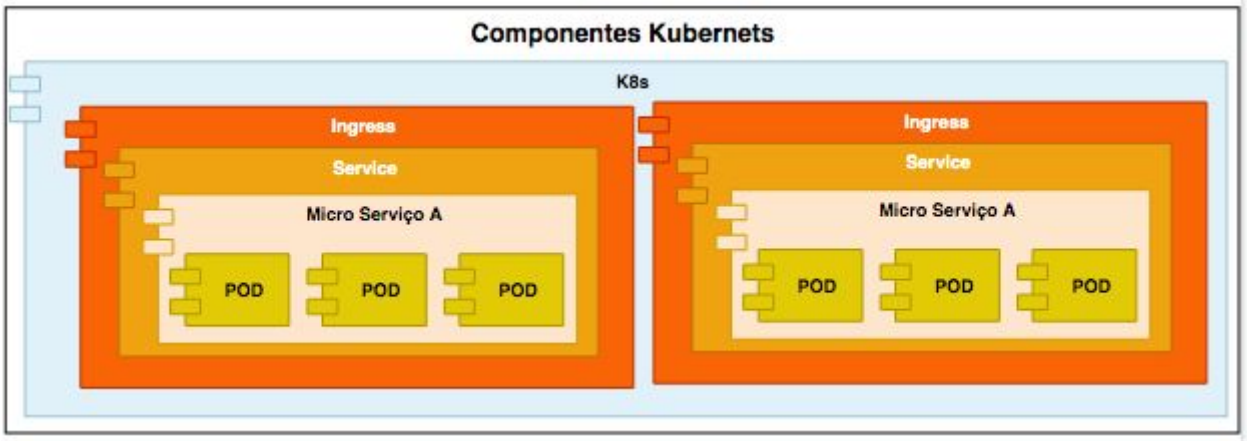
4.1. Modelo de componentes



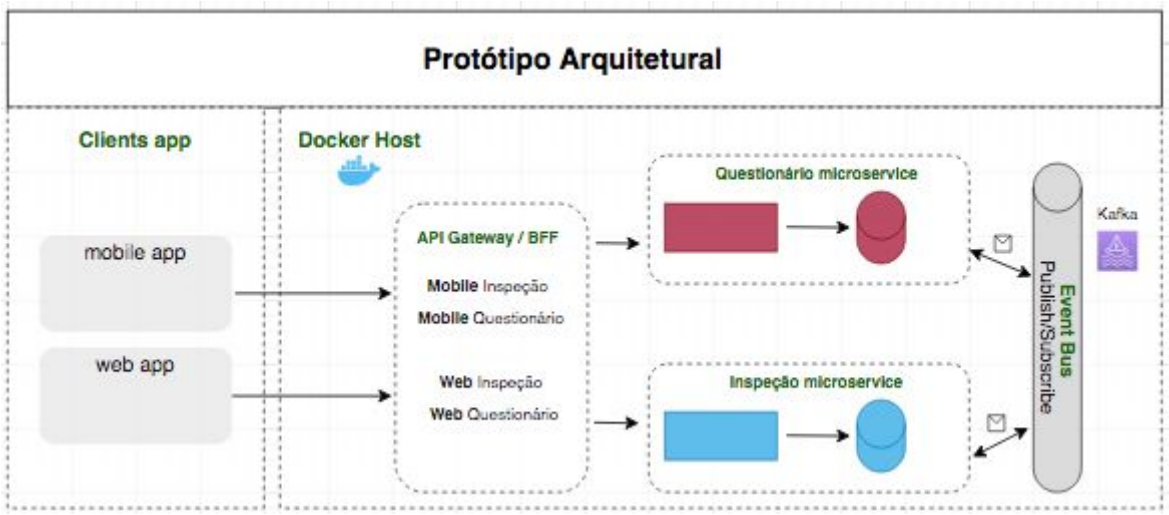

Componentes para Integração entre domínios
KafkaAdapter
«Adapter»
Subscribe
«Adapter»
Publish
Events
«Event»
Publish Listener
Bus
HandlerResolver
«Handler»
Event Subscribe
«Service»
Application
«Handler»
Publish Command
API
«Adapter»
Rest

4.2. Modelo de implantação





5. Prova de Conceito (POC) / protótipo arquitetural



5.1. Implementação e Implantação

Para implementação e implantação da POC destacamos as seguintes características:

- Tecnologias utilizadas: SpringBoot, Docker, Kubernets, Apache Kafka, EkS, API Gateway e Mongo DB.
- Casos de Uso **Manter Conformidades com os seguintes requisitos** Criar questionário, Criar Inspeção e Listar Inspeção
- Requisitos não funcionais avaliados.

    - **Comunicação entre os micro serviços deverá ser assíncrona.**
      Critério de aceite: Os micros serviços deverão fazer a troca de mensagem a partir do modelo publish/subscribe intermediado pelo componente Event Bus e **apache kafka.**

    - **Armazenar o estado dos dados da aplicação a partir de eventos.**
      Critério de aceite: Cada operação de leitura ou escrita deverá criar uma evento para publicação no apache kafka**.**

- **Os micro serviços deverão ser implementados separando componentes de leitura e escrita de dados, assim possibilitando utilizar recursos físicos separados como base de dados e ambientes de execução.**
  Critério de aceite: A camada de leitura e escrita dos micro serviços deverão ser apresentadas em módulos spring separados.

6. Avaliação da Arquitetura

A arquitetura proposta mostrou-se aderente às necessidades e estratégias de mercado a curto, médio e longo prazo. Por apresentar componentes de negócios pequenos, independentes e com um mecanismo de comunicação baseado na assinatura de eventos por interesse negocial, proporcionando maior agilidade e mais inteligência na interação entre serviços de negócio totalmente desacoplados. Impulsionando a interoperabilidade entre produtos que precisam atender um time-to-marketing agressivo e com riscos reduzidos. O ganho também passa a refletir na gestão dos times de desenvolvimentos, isso porque a arquitetura impulsiona a formação de equipes baseadas em contextos de negócio bem definidos de acordo com estratégias de marketing. Podemos destacar também maior eficiência no fluxo de valor incorporado a esteira devops que tenderá a ter lead times reduzidos e aprendizado contínuo.

6.1. Análise das abordagens arquiteturais

Integração entre serviços é uma realidade discutida há algum tempo nas disciplinas que estão por trás do desenvolvimento de software. A natureza do Event-driven assumirá uma arquitetura distribuída interagindo por eventos e possibilitando reações a eventos ocorridos os quais serão orquestrados por uma plataforma de mensagens, facilitando a extensão de um ecossistema sem comprometer as implementações existentes. Os micro serviços naturalmente ajudam a "estrangular" contextos de negócio auxiliando as equipes direcionar suas entregas, consequentemente alimentando um ecossistema onde será composto por várias peças as quais são entregues por vez e com características auto contidas como observabilidade, flexibilidade de infra estrutura, provisionamento de recurso computacional on de mand entre outros. A segregação da aplicação baseada no padrão CQRS vai possibilitar escrevermos códigos com responsabilidade de escrita diferente da leitura dentro de contextos de negócios, os quais são classificados como Bounded Context quando falamos de Domain Diven Design. O pattern Command Query Responsibility apresentou-se muito interessante pois a segregação não foi aplicada apenas de forma lógica mas também possibilitando a criação de módulos separados completamente

especializados no que diz respeito à leitura e escrita. Diante dos aspectos positivos apresentados, o que mais apresentou e agregou a arquitetura quando falamos de CQRS foi a possibilidade de levarmos modelos de dados desnormalizados para escrita e "normalizados" aplicados à leitura os quais serão preparados por equipes de dados utilizando as mais diversificadas ferramentas de transformação.

6.2. Cenários

- **Performance para requisições HTTP**

Qualitativo	
Tipo de Ação	**Transação por segundo**
Leitura	200
Gravação	100

- **Custo AWS**

Quantitativo				
EKS Nodes	Family Instance EC2	Números de Instancias	Quantidade de Dias	Custo
2	t3.medium	2	5	$ 25
1	t3.small	1	5	$ 10

6.4. Resultado

Minha proposta eleva a capacidade de entrega dos times com software prontos para produção e com o menor risco de afetar produtos já consumidos pelos clientes, também minimizando o problema da invasão de contextos de negócio mostrado na implementação do Event Driven. Por último e não menos importante, conseguimos organizar nosso código com base na arquitetura hexagonal que propõe a separação de componentes intercambiáveis organizados por camadas lógicas bem definidas. O que é interno ao aplicativo, o que é externo e o que é usado para conectar o código interno e externo são características importantes no hexagonal. Não abordamos neste trabalho assuntos de extrema importância que são: **Solução** de **transformação dos dados capturados pelos adaptadores Kafka quando são assinantes de tópicos e implementação de Dead Letter Queues(DLQ) para tratamento de erros,** complementos que deverão ser tratados para concluirmos nossa proposta arquitetural.

7. Conclusão

A arquitetura aqui exposta apresenta maior complexidade do que o habitual mas não tenho dúvidas que favorece ao desenvolvimento adaptável à transformação digital a qual vivemos. Cada dia área de negócio demanda soluções resilientes ao mercado, onde temos que apresentar soluções rápidas e com riscos totalmente minimizados. Não apresentei apenas tecnologias hoje tidas como novas ou "moda" mas sim uma arquitetura que possibilita sobreviver às mudanças de negócios sem termos que reescrever o passado. A mudança de mindset dos times de tecnologia poderá ser o maior "sabotador" da arquitetura proposta, pois nem sempre conseguimos engajamento total entre os times para que seja viável desenvolvermos alicerces de software para impulsionar produtos.

APÊNDICE A - URLs

Micro Serviço Questionário: http://3.137.98.24:8082/conformidade/v1/swagger-ui.html

Micro Serviço Inspeção: http://3.137.98.24:8081/conformidade/v1/swagger-ui.html

Kafka Web UI :

http://3.137.98.24:9000/

Código fonte:

https://github.com/rianmachado/easy-quality

Apresentação:

https://www.screencast.com/t/W3iOOSK82eU

https://www.screencast.com/t/sm6uHSk9e

https://www.screencast.com/t/GNkaN8TTORs

https://www.screencast.com/t/B3URwUNOCbsT

https://www.screencast.com/t/qxtC9nDoKhw

https://www.screencast.com/t/e3pLLn3S

Postman collection: Os payloads estarão disponíveis através da ferramenta POSTMAN. Após importar o arquivo JSON disponível abaixo troque as URLs **localhost** por **3.137.98.24**

https://github.com/rianmachado/easy-quality/blob/master/dominio-conformidades/collections/conformidades.postman_collection.json