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 RESPONSIBILITY SEGREGATION

Recife
2021

RIAN DE VASCONCELOS MACHADO

EVENT DRIVEN E MICROSERVICE USANDO COMMAND QUERY RESPONSIBILITY 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 e Júlia Oliveira de Vasconcelos. A Ana Adélia Oliveira e minha mãe Maria Nair de Vasconcelos.

RESUMO

O Easy Quality através de sua suíte de ferramentas e dados integrados permite às empresas aplicarem a gestão da qualidade de acordo com sua estratégia de negócio. Os módulos são desenhados no formato de escada de produtos e permitem incrementar ou remover módulos de software, levando aos clientes melhor experiência na sua jornada que precisa estar muito bem alinhada com a transformação digital e o time to marketing.

Diante disso não podemos adotar uma arquitetura na qual grandes blocos de softwares são desenvolvidos acoplando uma série de funcionalidades e as expondo em um servidor de aplicação "parrudo" e quando escalado, levar junto um conjunto de recursos computacionais que nem sempre é necessário para atender o aumento na demanda de determinada funcionalidade.

Precisamos trabalhar o auto scaling de recursos de acordo com a granularidade dos serviços de negócios, que muito provavelmente serão desenvolvidos por times distintos e com linguagens de programação que podem ser escolhidas com base nas características do problema a ser resolvido. A estruturação das camadas lógicas da aplicação onde os dados de leitura e escrita são tratados de forma unificada para atender necessidades de criação, consulta, atualização e remoção deverão ser repensadas.

O principal problema a ser resolvido é a comunicação bloqueante entre serviços, que em muitos cenários acabam tendo uma dependência muito grande um do outro. Diante do cenário apresentado, abordaremos a arquitetura de micro serviços impulsionada pela a orquestração de containers, a segregação de responsabilidades com CQRS e a comunicação assíncrona desenhada a partir da arquitetura event driven.

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**, detalhando como as arquiteturas propostas, padrões arquiteturais, infra estrutura, requisitos funcionais e não funcionais juntos se tornam o alicerce do produto. Os objetivos específicos são:

- 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**. A arquitetura proposta 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 negociais 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

Sistemas monolíticos tendem a elevar o grau de acoplamento, seja nas tecnologias subjacentes ou na escalabilidade do negócio. Fica ainda mais complexo falar de esteiras DevOps com lead time reduzido e produzindo uma peça por vez para atender determinados fluxos de valor. Outra questão problemática é a formação de equipes heterogêneas focadas e especializadas em contextos de negócio, onde cada time implementa seus aplicativos sem impactos nas funcionalidades dos demais. Diante desse cenário surge um novo problema que é a forma de comunicação entre esses aplicativos que passam a ser ainda mais distribuídos. Vejo também a necessidade de termos provisionamentos mais granulares e dados desenhados para leitura diferente da escrita.

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 ofertados. Os principais pilares são 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 será apresentado: requisitos funcionais, não funcionais, restrições e mecanismos arquiteturais considerados.

3.1. Requisitos Funcionais

- RF001 - Cadastrar questionário modelo

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

- RF002 - Listar questionários modelos

  Descrição: Permite listar todos os questionários cadastrados

- RF003 - Consultar questionário modelo

  Descrição: Permite consultar através de um identificador válido.

- RF004 - Criar inspeção

  Descrição: Permite criar uma inspeção de conformidade a partir de um questionário previamente cadastrado.

- RF005 - Listar inspeção

  Descrição: Permite listar todas as inspeções cadastradas

3.2 Requisitos Não-Funcionais

Estímulo	Nova POD é iniciada
Fonte do Estímulo	O Kubernets inicia a nova POD
Ambiente	O Micro Serviço opera com alta carga de requisição.
Artefato	Nova POD
Resposta	O Micro Serviço opera com carga distribuída entre as PODs
Medida da Resposta	A nova POD estará disponível em até 3 segundos após ser solicitada pelo gerenciador de métricas do Kubernets.

Estímulo	O Spring Boot recebe a requisição.
Fonte do Estímulo	Requisição REST a um micro serviço.
Ambiente	O Spring Boot opera em condições normais
Artefato	Spring Boot
Resposta	O Spring Boot processa várias requisições concorrentes.
Medida da Resposta	Uma requisição HTTP é processada em até três segundos.

Estímulo	Um API Gateway de borda recebe a requisição
Fonte do Estímulo	Requisição REST a um micro serviço.
Ambiente	Um API Gateway está disponível na DMZ
Artefato	API Gateway
Resposta	API Gateway repassa as requisições aos componentes internos
Medida da Resposta	O API Gateway deverá adicionar no máximo 300 milissegundos a latência total da requisição

Estímulo	O componente interno Eventbus recebe a requisição
Fonte do Estímulo	Requisição REST a um micro serviço.
Ambiente	O micro serviço está em condições normais para receber requisições.
Artefato	Eventbus
Resposta	O Eventbus cria eventos
Medida da Resposta	O Eventbus envia à plataforma de mensagem no mínimo uma mensagem para cada evento processado.

Estímulo	O componente interno adaptador do Kafka recebe a mensagem.
Fonte do Estímulo	Mensagem postada pelo Eventbus.
Ambiente	O Cluster Kafka está disponível.
Artefato	Cluster Kafka
Resposta	O Kafka recebe a mensagem e armazena em um tópico
Medida da Resposta	Pelo menos um nó do cluster Kafka está disponível para o processamento de mensagens.

3.3. Restrições Arquiteturais

- A aplicação deverá ser implantada em tecnologias de containers. É necessário realizar o empacotamento da aplicação através de imagens docker as quais ficarão armazenadas no Amazon Elastic Container Registry.
- Todo deploy deve ser realizado a partir de um modelo de implantação para gerenciar as PODS. É necessário declarar a implantação através de um arquivo YAML contendo no mínimo duas PODs.
- Toda comunicação entre micro serviços que compõem domínios de negócio deverá ser assíncrona. É necessário que o Kafka seja utilizado como plataforma de mensagem para intermediar a comunicação.
- Cada requisição deverá ser tratada internamente como Query ou Command a partir de adaptadores REST, que enviará os dados para o barramento de eventos.
- Repositórios de dados são particulares aos seus micro serviços. É necessário utilizar banco de dados orientado a documentos para gravação.

3.4. Mecanismos Arquiteturais

- **Gestão de Eventos**

  Descrição: Fornece um meio centralizado que captura input de dados permitindo que eventos sejam criados e lançados com características de leitura ou escrita.

  Mecanismo de Análise: Condições dos dados são observadas.

  Mecanismo de Design: O pattern CRQS é escolhido para ser implementado junto com o Event Bus.

  Mecanismos de Implementação: Converter inputs de consultas em objetos que implementam a interface **Query** e inputs de escritas em objetos que implementam a interface **Command**. Para processamento de um comandos ou consultas sempre invocar o componente **ServiceBus.**

- **Processamento de Eventos**

  Descrição: Permitir o processamento de eventos assíncronos possibilitando a publicação de mensagens.

  Mecanismo de Análise: Processamento de eventos criados pela aplicação.

  Mecanismo de Design: Padrão para troca de mensagem (publish/subscribe) é escolhido.

  Mecanismos de Implementação: Adaptadores Kafka deverão ser implementados a partir do módulo Spring Kafka. Os Eventos deverão ser capturados de forma assíncrona utilizando Spring Api Async ou CompletableFuture da linguagem Java.

- **Escalabilidade**

  Descrição: Permitir o auto scaling de aplicação mantendo o tempo de resposta mesmo com o aumento de usuários concorrentes.

  Mecanismo de Análise: Monitoramento de aplicação.

  Mecanismo de Design: Desenvolvimento de micro serviços utilizando container.

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

- **Persistência de dados**

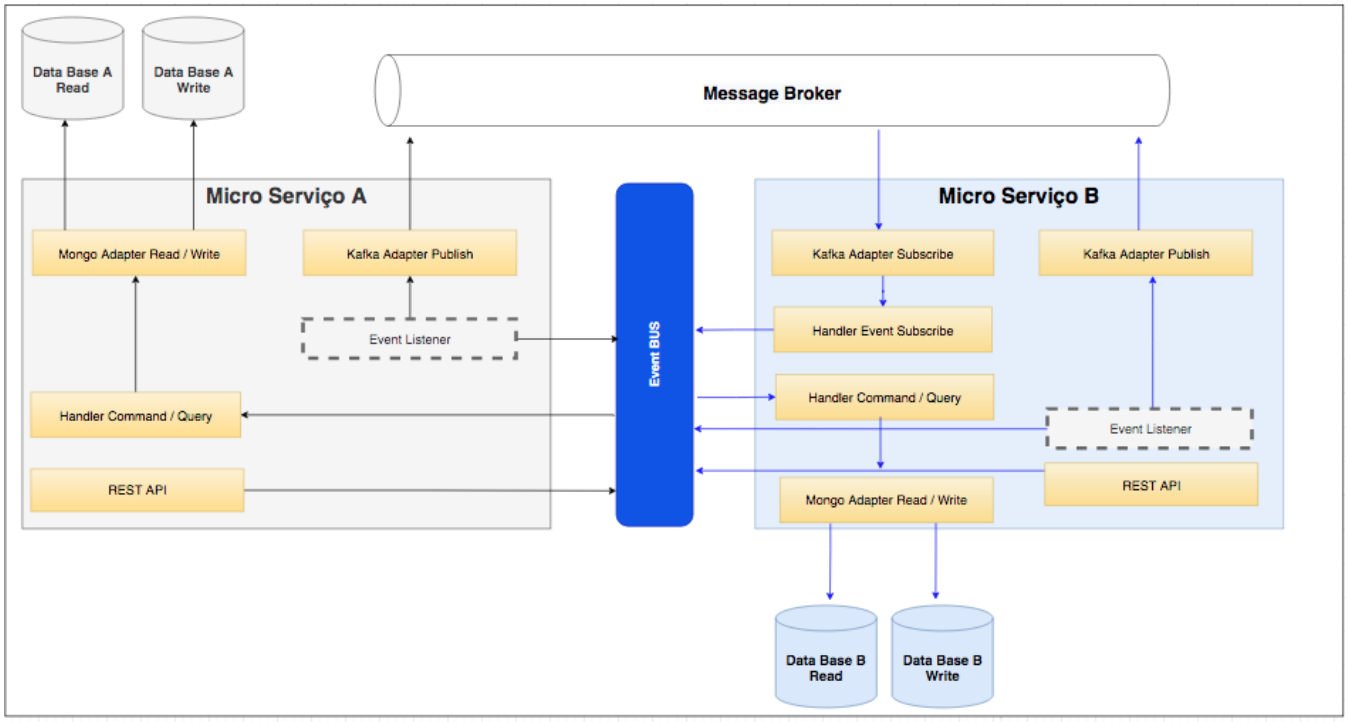
  Descrição: Serviços para lidar com a leitura e gravação de dados em base NoSQL

  Mecanismo de Análise: Persistência de dados.

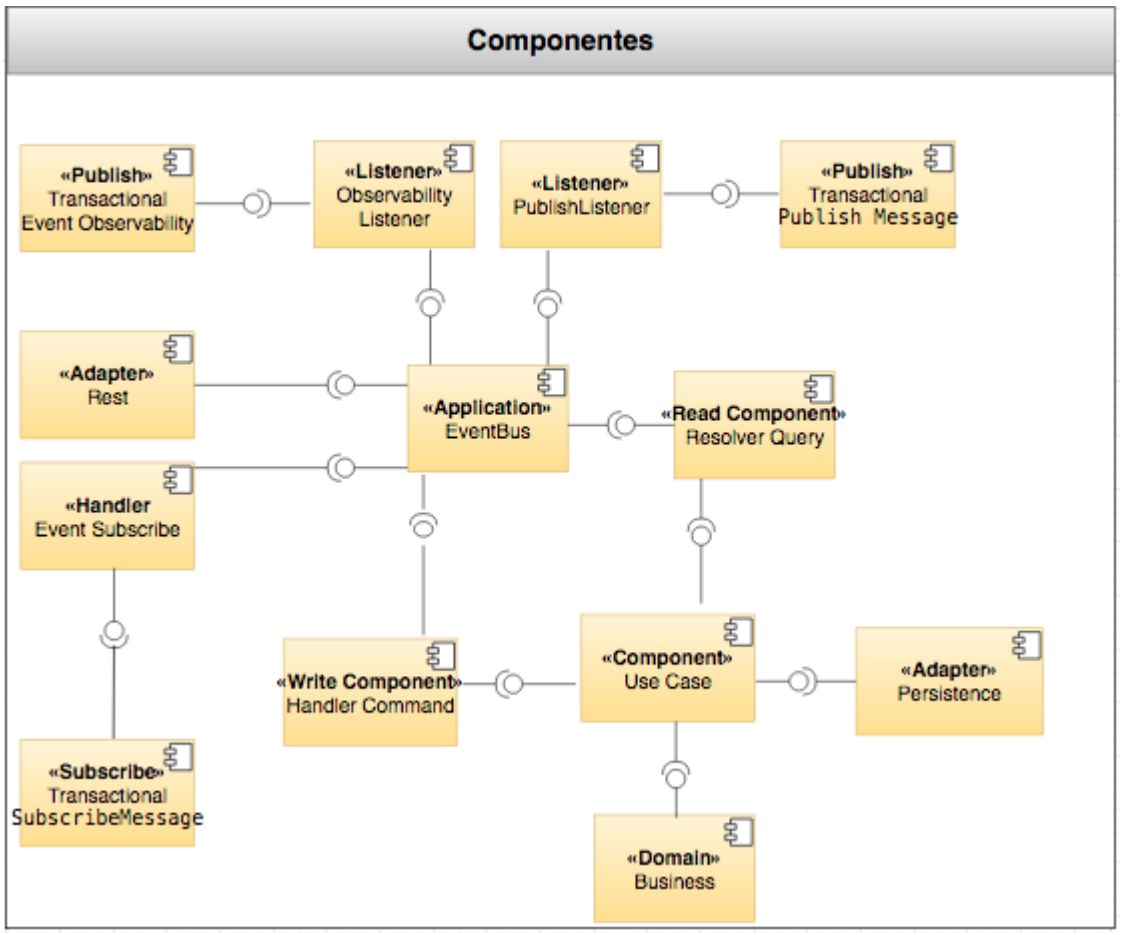
  Mecanismo de Design: Armazenamento de dados baseados em bancos orientados a documentos.

  Mecanismos de Implementação: O MongoDB deverá ser utilizado para armazenar dados no formato JSON. Toda camada de persistência da aplicação estende o componente MongoRepository disponível no Spring Data.

4. Modelagem e projeto arquitetural



4.1 Modelo de componentes



Transactional Event Observability: Responsável por publicar no Kafka mensagens criadas a partir dos eventos referentes à observabilidade de dados.

Observability Listener: Captura eventos referentes a observabilidade de dados.

Publish Listener: Captura eventos de negócio emitidos pelos micro serviços.

Transaction Publish Message: Responsável por publicar mensagens criadas a partir de eventos de negócio.

Transaction Subscribe Message: Responsável por receber mensagens publicadas em canais de comunicação conhecidos como tópicos.

Handler Event subscribe: Responsável por manipular dados recebidos pelos assinantes de mensagens.

Event Bus: Centraliza inputs de dados para aplicar a gestão de eventos nos dados de entrada e saída. Responsável também por rotear consultas e comandos para seus respectivos componentes.

Adapter Rest: Expõe a camada interna da aplicação através de APIs REST.

Handler Command: Manipula todos os comandos de escrita antes de repassar os dados para camadas mais internas dos micro serviços.

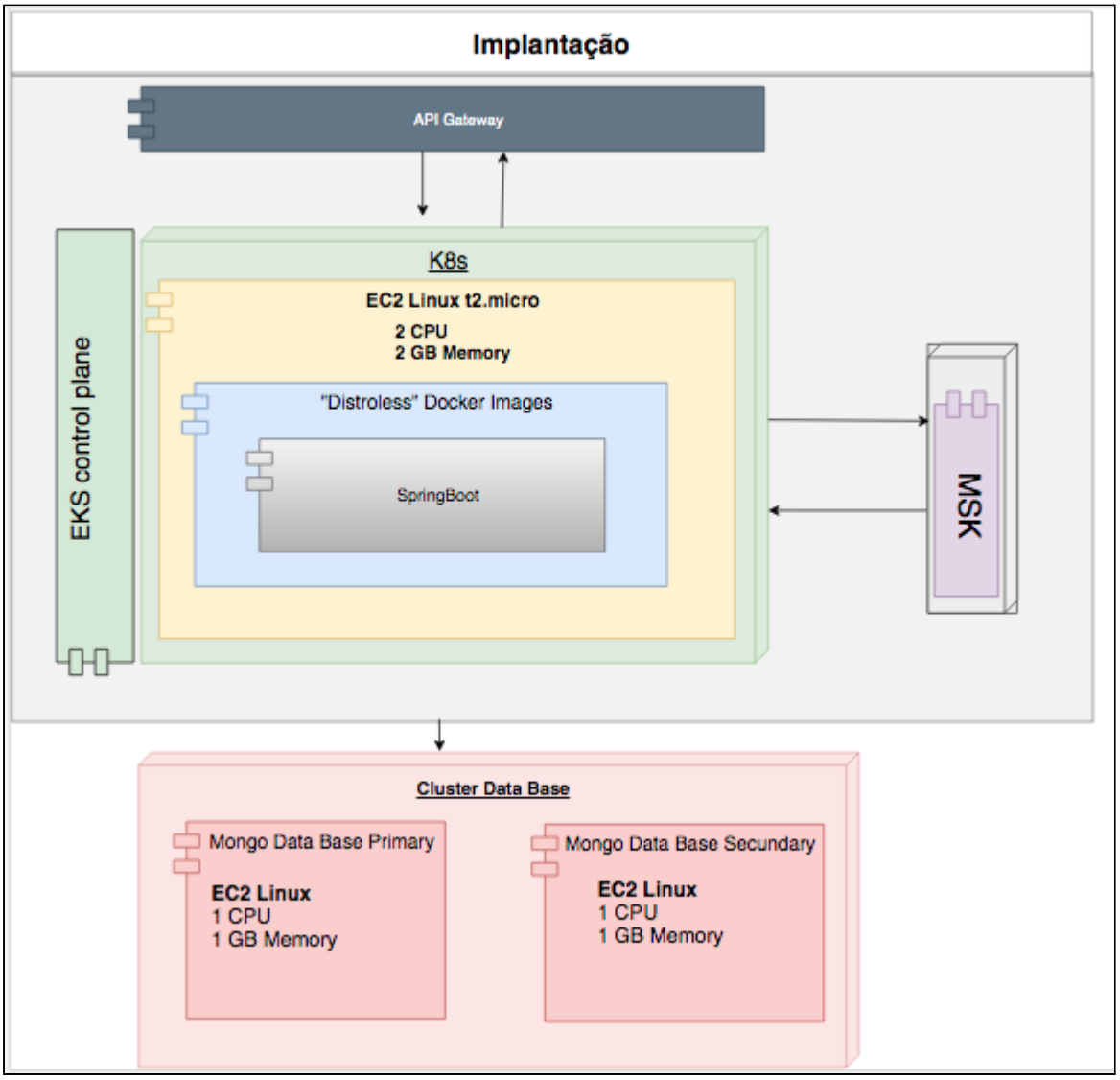
Resolver Query: Manipula todos os comandos de leitura antes de repassar os dados para camadas mais internas dos micro serviços.

Use Case: Componente responsável por abrigar a implementação dos cenários de cada use case.

Domain Business: Componente que implementa comportamentos de negócios e que compõem os domínios.

Adapter Persistence: Implementa operações como leitura, escrita, consulta e remoção de dados no Mongo DB

4.2. Modelo de implantação



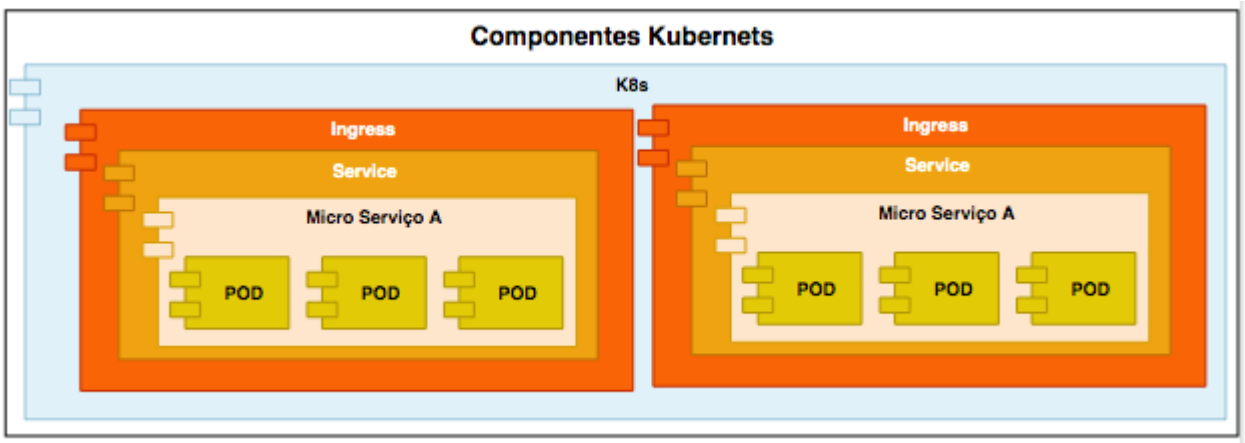
Amazon API Gateway (Nuvem)

- Exposição de serviços HTTP na DMZ
- Prover autenticação
- Criação de APIs RESTful
- Implantações de versão Canary

Amazon EKS (Nuvem)

- Gerenciar execução do Kubernetes
- Executa e dimensiona o plano de controle do Kubernetes em várias zonas de disponibilidade da AWS para garantir alta disponibilidade.

- Escala automaticamente as instâncias do plano de controle com base na carga, detecta e substitui instâncias não íntegras.
- Está integrado a vários serviços da AWS para fornecer escalabilidade e segurança aos aplicativos, incluindo recursos como: **Amazon ECR para imagens de contêiner, Elastic Load Balancing para distribuição de carga,**



IAM para autenticação e Amazon VPC para isolamento.

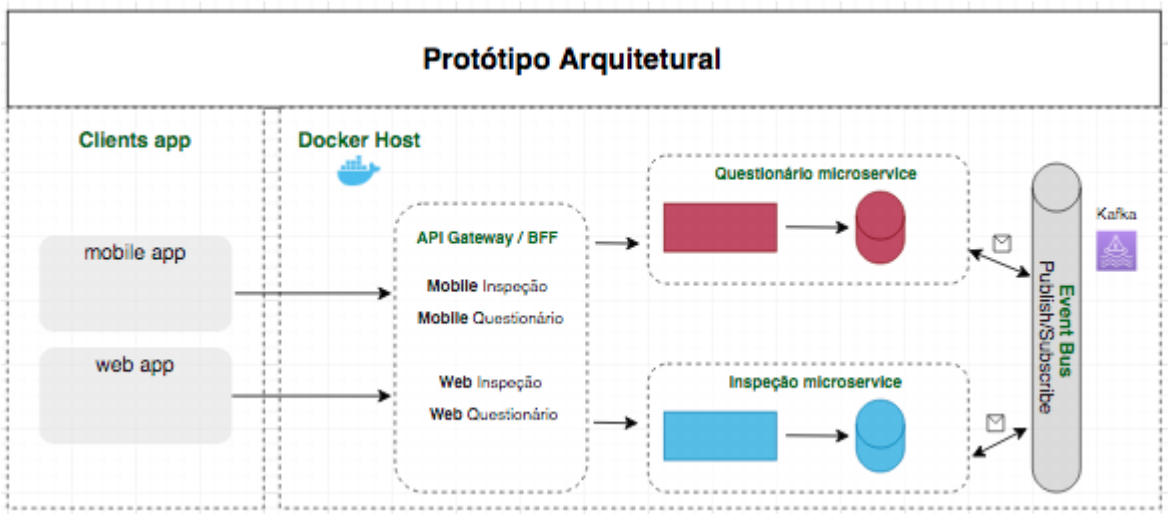
Amazon EC2 (Nuvem)

- Ambientes de computação virtual, conhecidos como instâncias.
- Os modelos pré-configurados para suas instâncias, conhecidos como Imagens de máquina da Amazon (AMIs), que empacotam os bits de que você precisa para seu servidor (incluindo o sistema operacional e software adicional).
- Tipos de instância para configurações de capacidade de CPU, memória, armazenamento e rede.
- Volumes de armazenamento persistentes para dados usando o **Amazon Elastic Block Store**.

Amazon MSK (Nuvem)

- Gerenciamento de Streaming para apache Kafka

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 questionário: O usuário cria questionários modelos para aplicar em pesquisas de conformidade. O sistema permite a exibição e edição dos modelos cadastrados.

  Manter inspeção: O usuário realiza pesquisas a partir de modelos de questionários pré definidos. O Sistema permite a exibição de todos os questionários respondidos.

  Plano de ação: O Sistema avalia os questionários respondidos listando para o usuário as respostas que não atingiram o percentual mínimo de conformidade. O usuário cria checklists para acompanhamento de questionários em não conformidades.

A solução proposta não traz um sistema com telas pré definidas. O comportamento da aplicação é exposto através de uma linguagem de descrição de interface utilizada para descrever APIs RESTful expressas usando JSON. O **Swagger-UI** foi utilizado como ferramenta para levar as informações necessárias aos desenvolvedores front-end e com isso possibilitar a criação de qualquer tipo de tela. Para maiores detalhes consulte a seção apêndice.

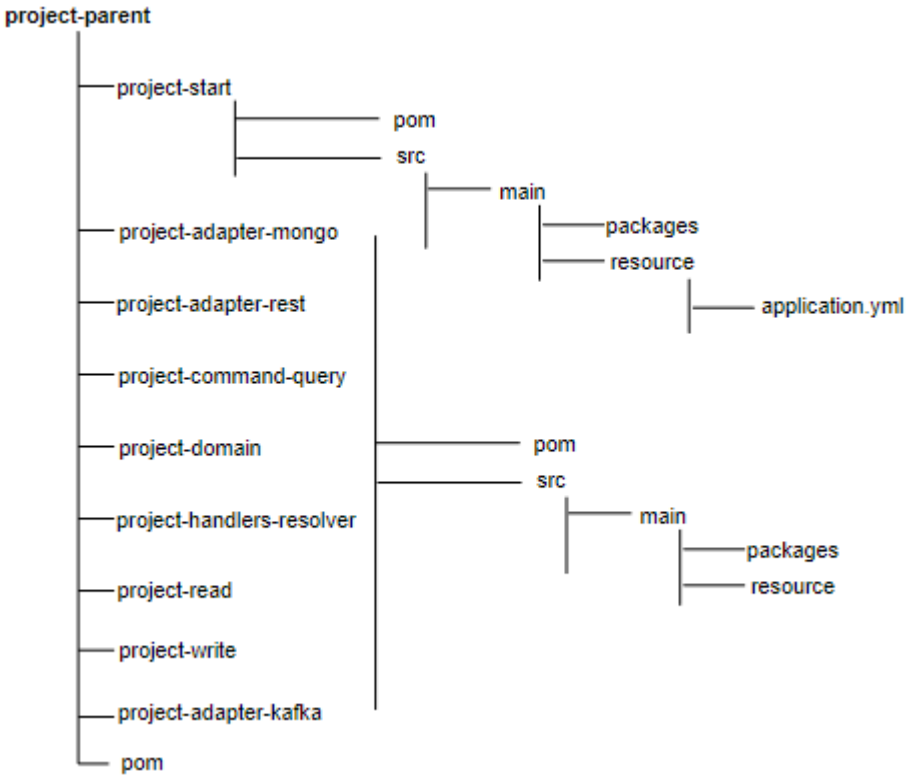
- Requisitos não funcionais avaliados.

    - **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.

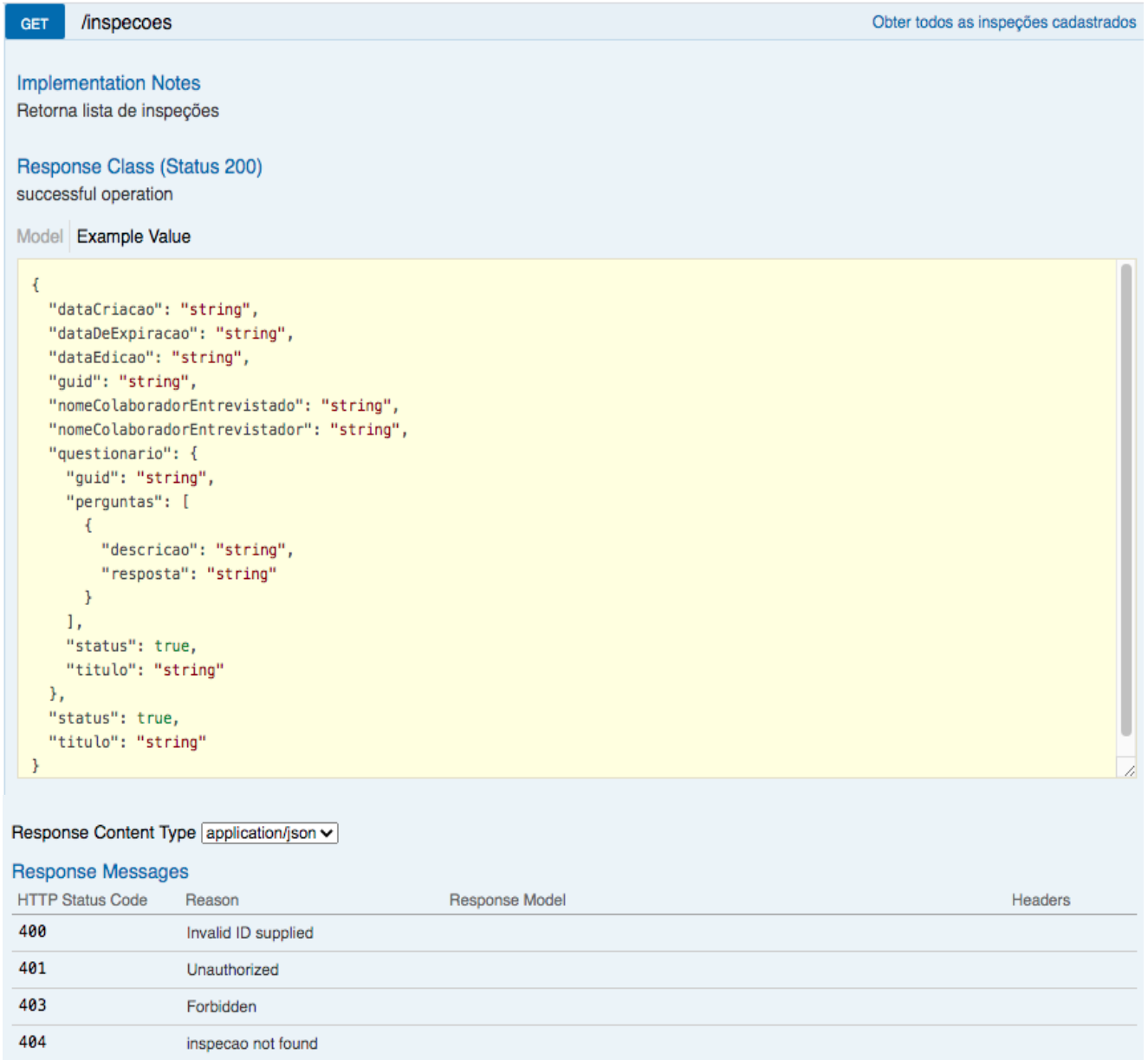
5.2 Código



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

5.3. Interfaces/ APIs

- **API inspeção**



GET /inspecoes — Obter todos as inspeções cadastrados

Implementation Notes

Retorna lista de inspeções

Response Class (Status 200)

successful operation

Model | Example Value


```
{
  "dataCriacao": "string",
  "dataDeExpiracao": "string",
  "dataEdicao": "string",
  "guid": "string",
  "nomeColaboradorEntrevistado": "string",
  "nomeColaboradorEntrevistador": "string",
  "questionario": {
    "guid": "string",
    "perguntas": [
      {
        "descricao": "string",
        "resposta": "string"
      }
    ],
    "status": true,
    "titulo": "string"
  },
  "status": true,
  "titulo": "string"
}
```


Response Content Type application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	Invalid ID supplied		
401	Unauthorized		
403	Forbidden		
404	inspecao not found		

POST /inspecoes — Criar Inspecao

Parameters

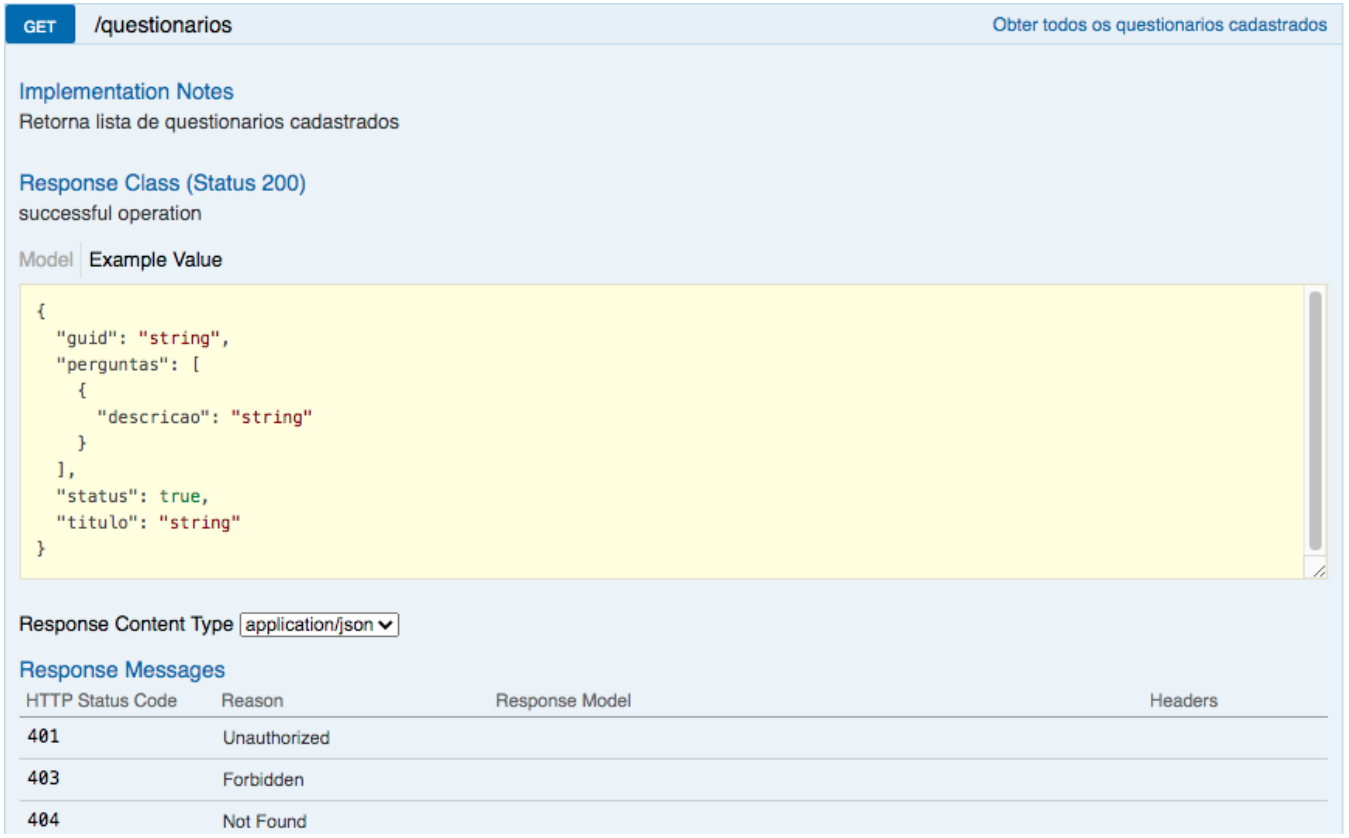
Parameter	Value	Description	Parameter Type
body	`{ "dataCriacao": "string", "dataDeExpiracao": "string", "dataEdicao": "string", "guid": "string", "nomeColaboradorEntrevistado": "string", "nomeColaboradorEntrevistador": "string", "questionario": { "guid": "string", "perguntas": [{ "descricao": "string", "resposta": "string" }], "status": true, "titulo": "string" }, "status": true, "titulo": "string" }`	**Objeto utilizado para adicionar nova inspecao**	body

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		
405	Invalid input		

- **API Questionário**



POST /questionarios — Criar questionario

Parameters

Parameter	Value	Description	Parameter Type

body	<pre>{
	"status":true,
	"titulo" : "string",
	"perguntas" : [
	{
	"descricao" : "string"
	}
	]
	
	}</pre>
		Objeto utilizado para adicionar novo questionario	body

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	OK		
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		
405	Invalid input		

GET /questionarios/{questionarioId} Obter questionario por ID

Implementation Notes
Retorna uma questionario simples

Response Class (Status 200)
successful operation

Model | Example Value


```
{
  "guid": "string",
  "perguntas": [
    {
      "descricao": "string"
    }
  ],
  "status": true,
  "titulo": "string"
}
```


Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
questionarioId	(required)	**ID da questionario para retorno**	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	Invalid ID supplied		
401	Unauthorized		
403	Forbidden		
404	Questionario not found		

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, diminuindo a complexidade na comunicação entre micro serviços.

A implementação da arquitetura Event Driven apresentada possibilitou o detalhamento de componentes internos importantes que sustentam o paradigma da arquitetura de software que promove a produção, detecção, consumo e reação a eventos minimizando cenários onde encontramos comunicação síncrona com fluxos complexos que envolvem vários micro serviços. Outro aspecto positivo observado na arquitetura baseada em eventos foi a resiliência encontrada nos micro serviços pois não precisaram implementar lógicas de comunicação, geralmente presentes quando temos comunicação síncrona utilizando APIs REST. A comunicação abordada neste tipo de arquitetura é abstraída por plataformas de mensagem, nesse caso o kafka. Isso possibilitou implementarmos a observabilidade nos dados a partir de eventos os quais representam as mudanças no estado do aplicativo.

Podemos não apenas consultar esses eventos, mas também usar o log de eventos para reconstruir estados anteriores e isso foi possível com a implementação do pattern Event Source o que agregou muito na proposta arquitetural. Os microsserviços trouxeram uma abordagem arquitetônica e organizacional ao desenvolvimento no qual conseguimos elaborar pequenas peças de software independentes que expõem suas informações usando APIs bem definidas. Na minha avaliação as maiores vantagens que os microsserviços ofertaram ao desenho de arquitetura proposto foram: possibilitar a entrega de peças independentes facilitando a implementação de uma esteira mais eficiente e também organizar times independentes especialistas.

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 esses eventos ocorridos, os quais serão orquestrados por uma plataforma de mensagens, facilitando a extensão de um ecossistema sem comprometer implementações existentes.

Os micro serviços naturalmente ajudam a "estrangular" contextos de negócio, auxiliando as equipes direcionarem suas entregas alimentando um ecossistema composto por várias peças, as quais são entregues uma por vez e com características auto contidas como observabilidade, flexibilidade de infra estrutura, provisionamento de recurso computacional on demand 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 Driven Design.

O pattern CQRS apresentou-se muito interessante pois a segregação não foi aplicada apenas de forma lógica mas também possibilitou 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

Disponibilidade
Os serviços REST estarão disponíveis pelo menos 99.98% do tempo de qualquer mês do ano.

Desempenho
O tempo de resposta dos serviços é sempre inferior a 1.4 segundos para consultas individuais, incluíndo a latência da internet.

Escalabilidade
Cada micro serviço deverá ser escalado com base no servidor de métricas, recurso disponível no Kubernetes. As pods deverão ser dimensionadas a partir da utilização de CPU e memória observada.

Manutenibilidade
Alterações nos micro serviços não deverão causar impactos a qualquer domínios de negócio em produção.

Confiabilidade
Comunicações baseadas em publish/subscribe deverão tratar erros em DLQs, possibilitando o reprocessamento sem bloqueio.
O Retry deverá ser aplicado com limites de 10 tentativas, com isso ter a capacidade de simplesmente aceitar a falha ou tentar uma alternativa para atender determinada requisição.

Portabilidade
Os serviços deverão rodar em qualquer provedor de nuvem que disponha de serviços para orquestração de container, desde que seja observada a menor latência de rede.
O Retry deverá ser aplicado com limites de 10 tentativas, com isso ter a capacidade de simplesmente aceitar a falha ou tentar uma alternativa para atender determinada requisição.

6.4. Resultado

Minha proposta é elevar a capacidade de entrega dos times com softwares prontos para produção e com o menor risco de afetar produtos já consumidos pelos clientes, também minimizando problemas como a invasão de contextos de negócio através da 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 encontradas na arquitetura hexagonal. Apesar de não termos apresentado implementações referentes a **Dead Letter Queues(DLQ)** para tratamento de erros, o desenho arquitetural proposto e as tecnologias escolhidas permitirão sem grandes esforços implementarmos micro serviços resilientes.

7. Conclusão

A arquitetura aqui apresentada, demonstra maior complexidade do que o habitual mas não tenho dúvidas que favorece ao desenvolvimento adaptável à transformação digital a qual vivemos. Cada dia uma área de negócio demanda soluções resilientes ao mercado, onde temos que apresentar soluções rápidas e com riscos totalmente minimizados. A proposta não traz apenas tecnologias aceitas tendências ou "modas" 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 o engajamento total entre os times para que seja viável desenvolvermos alicerces de software para impulsionar produtos.

APÊNDICE A - URLs

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

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

Kafka Web UI:

http://3.137.98.24:9000/

Apresentação:

https://www.youtube.com/watch?v=EOih99Djk6Q&t=2s

https://www.youtube.com/watch?v=4FhqNN9ySgc&t=1s

https://www.youtube.com/watch?v=K6jJU_gYuVo&t=7s

https://www.youtube.com/watch?v=lO2y-Ed4cac&t=10s

https://www.youtube.com/watch?v=6pdefbT4Zg8

https://www.youtube.com/watch?v=uC7mLAcSFfA