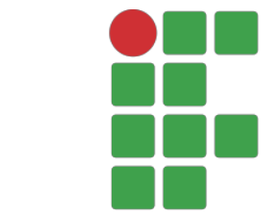


INSTITUTO FEDERAL
Paraíba

Campus Cajazeiras

PROGRAMAÇÃO P/ WEB 2

3

5. IMPLEMENTANDO CONSULTAS NUMA ARQUITETURA DE MICROSSERVIÇOS

PROF. DIEGO PESSOA

DIEGO.PESSOA@IFPB.EDU.BR

@DIEGOEP

CST em Análise e Desenvolvimento de Sistemas

ROTEIRO

- Desafios para consultar dados numa arquitetura de microsserviços
- Quando e como implementar consultas usando o padrão API composition
- Quando e como implementar consultas usando o padrão Command Query Responsibility Segregation (CQRS)

INTRODUÇÃO

- O problema das transações de escrita distribuídas pode ser resolvido com o uso de sagas
- No entanto, resta uma lacuna que são as consultas distribuídas: como recuperar dados que estão espalhados por múltiplos serviços / bancos de dados?
- Não é possível usar os mecanismos de consulta distribuídas , pois violam o encapsulamento e são tecnicamente inviáveis

INTRODUÇÃO

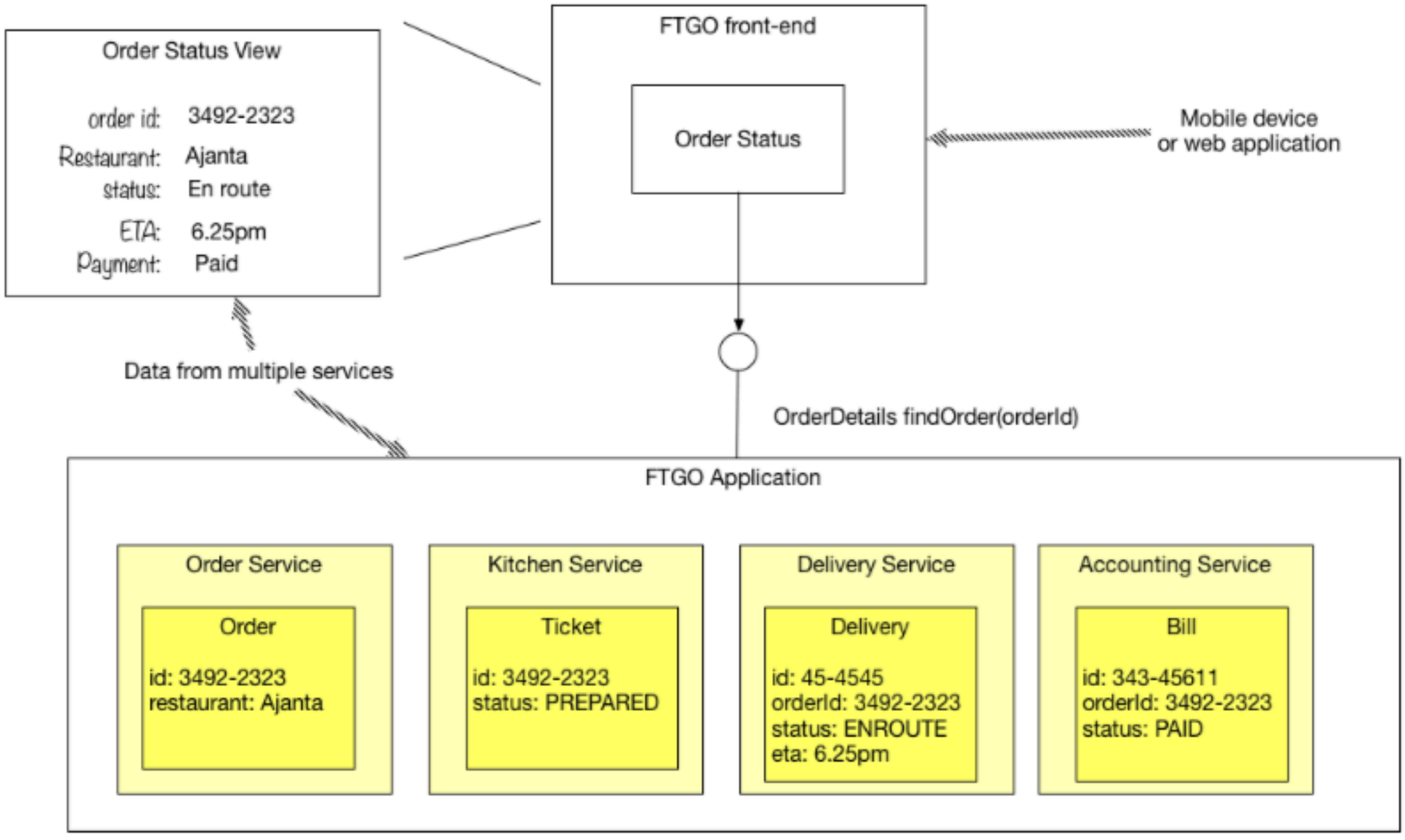
- Algumas operações de consulta (ex.: recuperarPedido(), recuperarHistoricoDePedido()) retornam dados de múltiplos serviços
- Implementar essas operações de consulta não é simples. Alguns padrões podem ser usados:
  - API Composition: abordagem mais simples, realiza chama aos serviços responsáveis por cada dado e combina os resultados
  - Comand Query Responsibility Segregation (CQRS): mais poderoso e mais complexo, mantém uma ou mais visões só para suportar consultas.

1. FAZENDO CONSULTAS USANDO API-COMPOSITION

FAZENDO CONSULTAS USANDO O PADRÃO API COMPOSITION

- Uma aplicação geralmente implemente várias operações de consulta. Muitas deles somente recuperam dados de um único serviço
- No entanto, há alguns casos em que é necessário recuperar dados de múltiplos serviços.

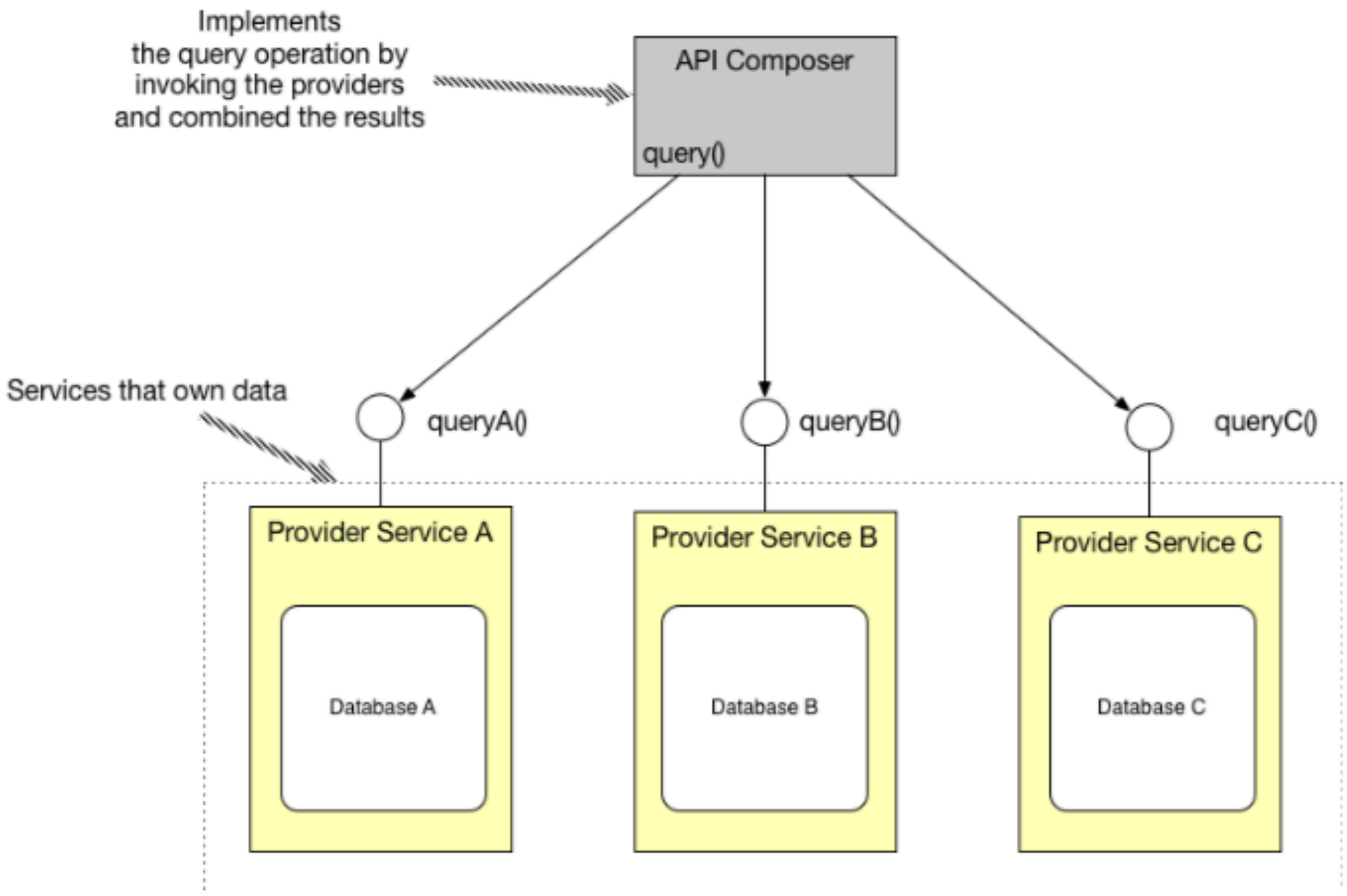
EXEMPLO DE CONSULTA USANDO O PADRÃO API COMPOSITION



EXEMPLO DE CONSULTA USANDO O PADRÃO API COMPOSITION

- Um pedido pode ter informações advindas de vários serviços (pagamento, restaurante, entrega)
- Caso fosse monolítico e residisse em um único banco de dados, seria fácil fazer uma consulta. Bastaria dar um SELECT com os joins em várias tabela.
- Por outro lado, usando microsserviços, os dados estão espalhados em múltiplos serviços, então teríamos algo como para um serviço de pedidos:
  - Order Service - informação básica sobre o pedido, incluindo detalhes e status
  - Kitchen Service - estado do pedido na perspectiva do restaurante
  - Delivery Service - data estimada de entrega
  - Account Service - estado do pagamento do pedido
- Qualquer cliente que precisar todos os detalhes de um pedido, terá de consultar todos esses serviços

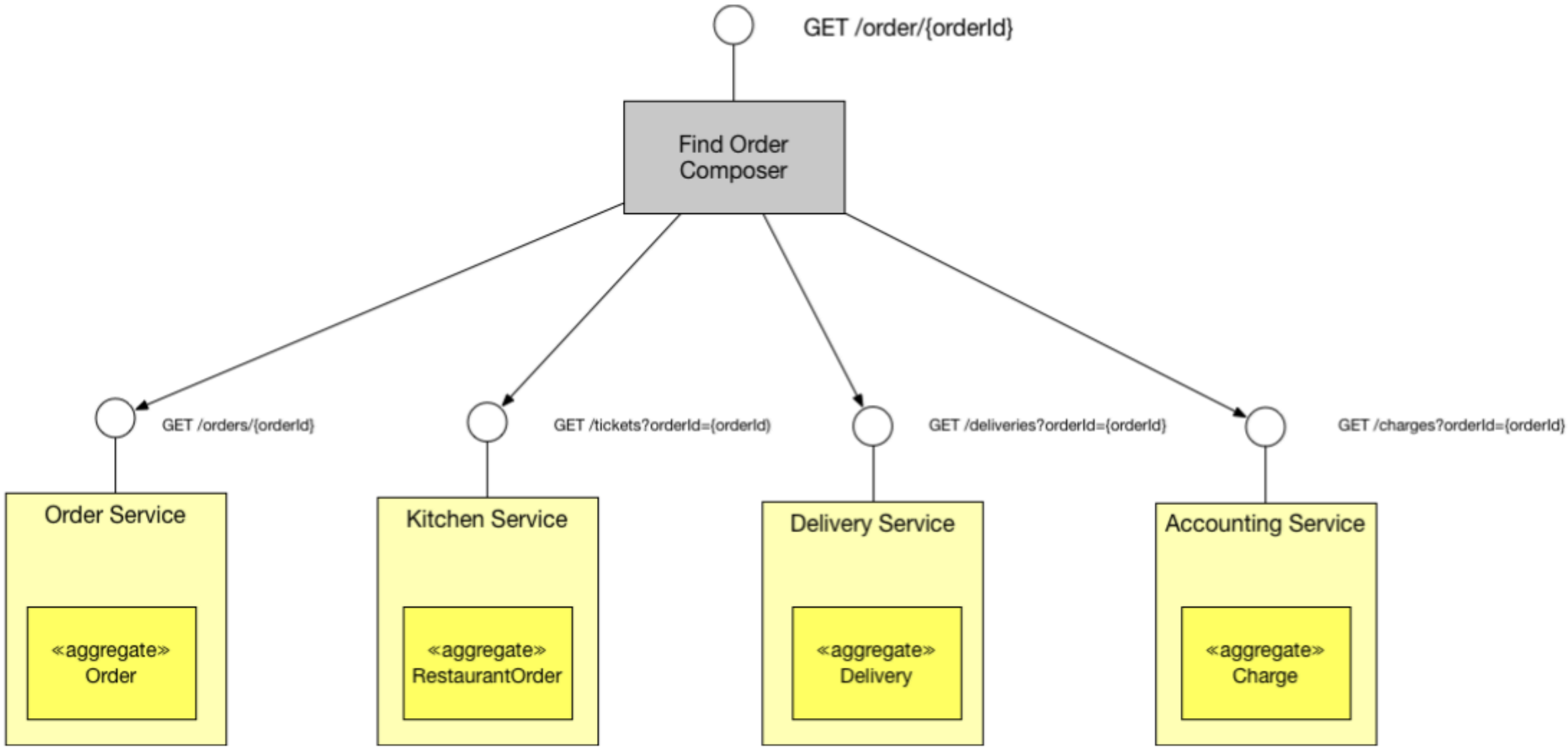
PADRÃO API COMPOSITION



IMPLEMENTA UMA CONSULTA QUE PRECISA RECUPERAR DADOS DE VÁRIOS SERVIÇOS RECUPERANDO DADOS DE CADA API E COMBINANDO OS RESULTADOS

Padrão API Composition

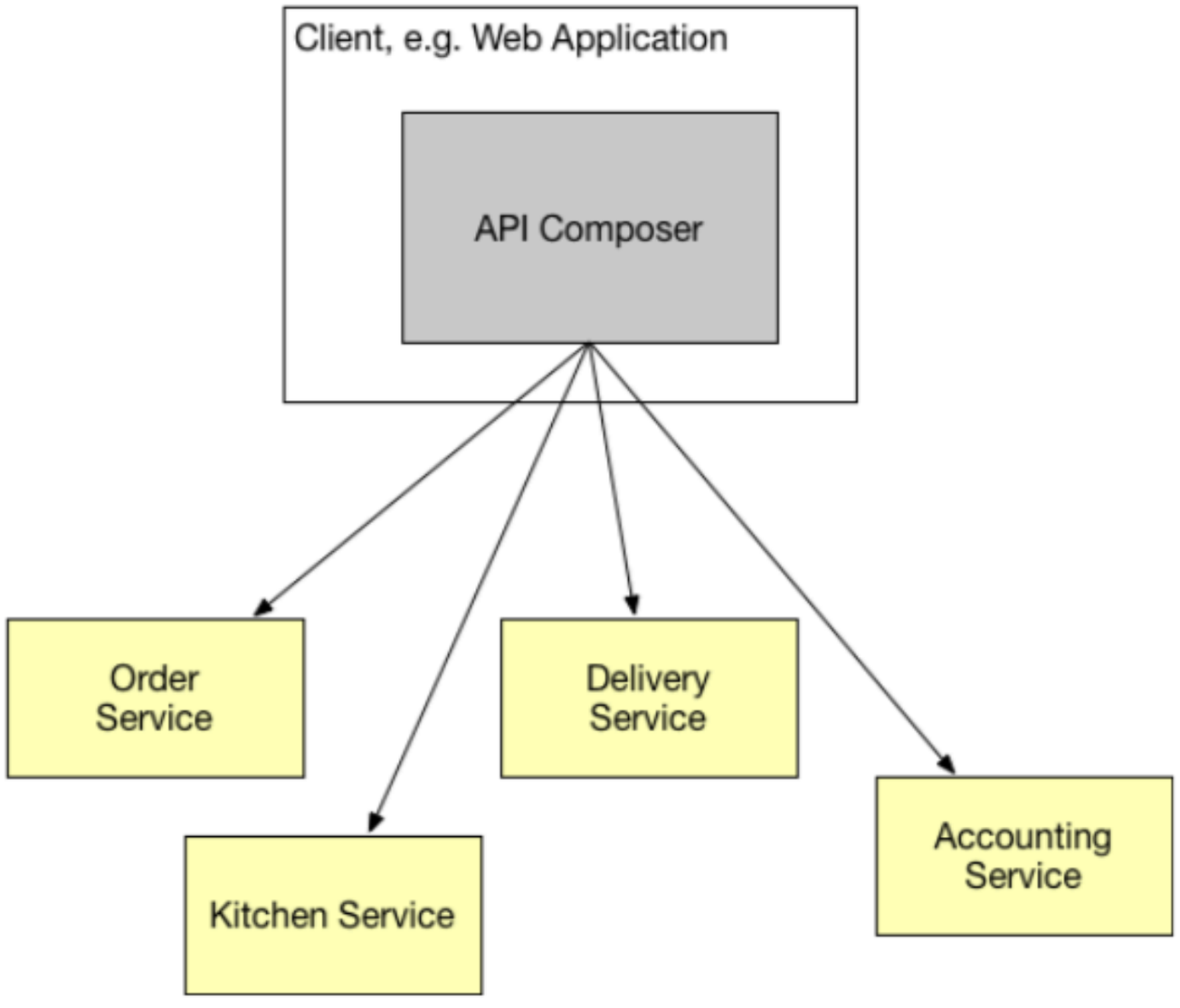
IMPLEMENTANDO O PADRÃO API COMPOSITION



API COMPOSITION – QUESTÕES DE DESIGN

- Ao utilizar este padrão, há uma série de questões que devem ser tratadas:
  - Decidir qual componente na arquitetura fará o papel de compositor de APIs
  - Como escrever lógica de agregação eficiente

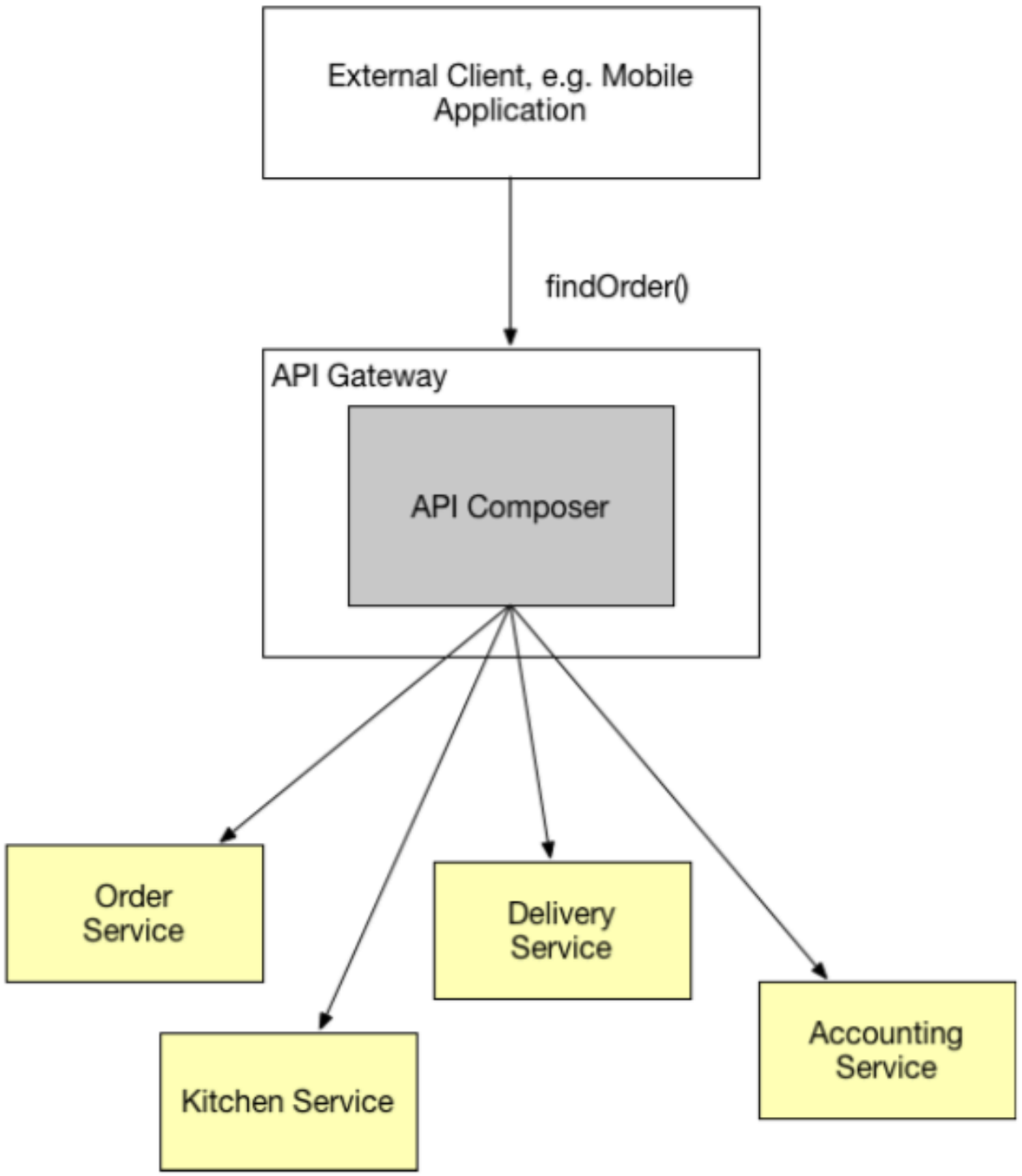
QUEM FAZ O PAPEL DE "API COMPOSER"?



QUEM FAZ O PAPEL DE "API COMPOSER"?

- Um cliente front-end que implementa o acesso aos serviços e roda na rede local, poderia recuperar os detalhes usando este padrão. No entanto, para clientes que estão fora da rede local (e do controle do firewall), esse acesso se torna um problema.
- A segunda opção, é utilizar um API Gateway, padrão que implementa uma API externa da aplicação, para fazer o papel de API Composer

QUEM FAZ O PAPEL DE "API COMPOSER"?



- Esta abordagem permite que um cliente, seja mobile ou web, que esteja rodando fora daquela rede, recupere de maneira eficiente dados de vários serviços com uma única chamada a API

API COMPOSERS DEVEM UTILIZAR O MODELO REATIVO DE PROGRAMAÇÃO

- Ao desenvolver um sistema distribuído, uma preocupação constante é minimizar latência
- Sempre que possível, um API composer deve chamar seus serviços provedores em paralelo, visando minimizar o tempo de resposta para uma consulta.
- Em alguns casos, no entanto, o API composer precisa do resultado fornecido por um serviço. Neste caso, é necessário invocar em sequência.
- A lógica para executar de maneira eficiente uma mistura de chamadas sequenciais e paralelas pode ser complexa. Para fazer um API composer ser tanto fácil de manter, como performático, como escalável, deve ser usado um design reativo baseado em Java CompletableFutures, RxJava, etc.
-

BENEFÍCIOS E DESVANTAGENS

- Benefícios: forma simples e intuitiva de implementar operações de consulta numa arquitetura de microsserviços
- Desvantagens:
  - Aumento do overhead
  - Risco de disponibilidade reduzida ()
  - Falta de consistência de dados transacional

AUMENTO DO OVERHEAD

- Invocar múltiplos serviços significa um aumento no número de requisições realizadas
- Mais recursos computacionais e de rede são requeridos, o que aumento o custo de manter a aplicação rodando

RISCO DE DISPONIBILIDADE REDUZIDA

- A disponibilidade do sistema passa a depender da disponibilidade dos serviços que são chamados
- Há algumas estratégias que podem ser usadas para lidar com o problema da indisponibilidade:
  - O API composer retornar dados em uma cache quando um serviço fornecedor esteja indisponível
  - Retornar dados incompletos, retirando os dados provenientes dos serviços que estejam indisponível

FALTA DE CONSISTÊNCIA TRANSACIONAL

- A falta de transações ACID deixa a aplicação suscetível a retornar dados inconsistentes, já que não é possível garantir o isolamento dos dados recuperados.

- Por exemplo: um pedido pode ter sido cancelado e o seu ticket no serviço de restaurante ainda não, o que faria com que esses dados fossem retornados mesmo após o cancelamento do pedido.

API COMPOSITION – CONSIDERAÇÕES

- Apesar das desvantagens, o padrão API Composition é extremamente útil e ele pode ser utilizado para implementar várias operações de consulta.
- Há, no entanto, operações que não podem ser implementadas de maneira eficiente usando esse padrão.
- Para os casos de consultas que, por exemplo, requeiram a junção de vários datasets em larga escala, geralmente é melhor implementar essas consultas usando CQRS.

2. FAZENDO CONSULTAS USANDO O PADRÃO CQRS

USANDO O PADRÃO CQRS

- Apesar das desvantagens, o padrão API Composition é extremamente útil e ele pode ser utilizado para implementar várias operações de consulta.
- Há, no entanto, operações que não podem ser implementadas de maneira eficiente usando esse padrão.
- Para os casos de consultas que, por exemplo, requeiram a junção de vários datasets em larga escala, geralmente é melhor implementar essas consultas usando CQRS.

O PADRÃO CQRS

- Muitas aplicações corporativas usam um banco relacional como sistema transacional para gravar dados um banco de dados de texto (NoSQL) realizar buscas, como o ElasticSearch ou Solr.
  - Algumas aplicações mantém as bases de dados sincronizadas, escrevendo em ambas de maneira simultânea.
- Outras periodicamente copiam os dados do banco relacional para o mecanismo de busca.
- Aplicações com esta arquitetura se aproveitam das forças de múltiplos bancos de dados: as propriedades transacionais do banco relacional e o poder das consultas em bancos de dados textuais.

IMPLEMENTA UMA CONSULTA QUE DEMANDA DADOS DE VÁRIOS SERVIÇOS USANDO EVENTOS PARA MANTER UMA VISÃO SOMENTE-LEITURA QUE REPLICA OS DADOS DOS SERVIÇOS.

Padrão: Command Query Responsibility Segregation (CQRS)

O PADRÃO CQRS

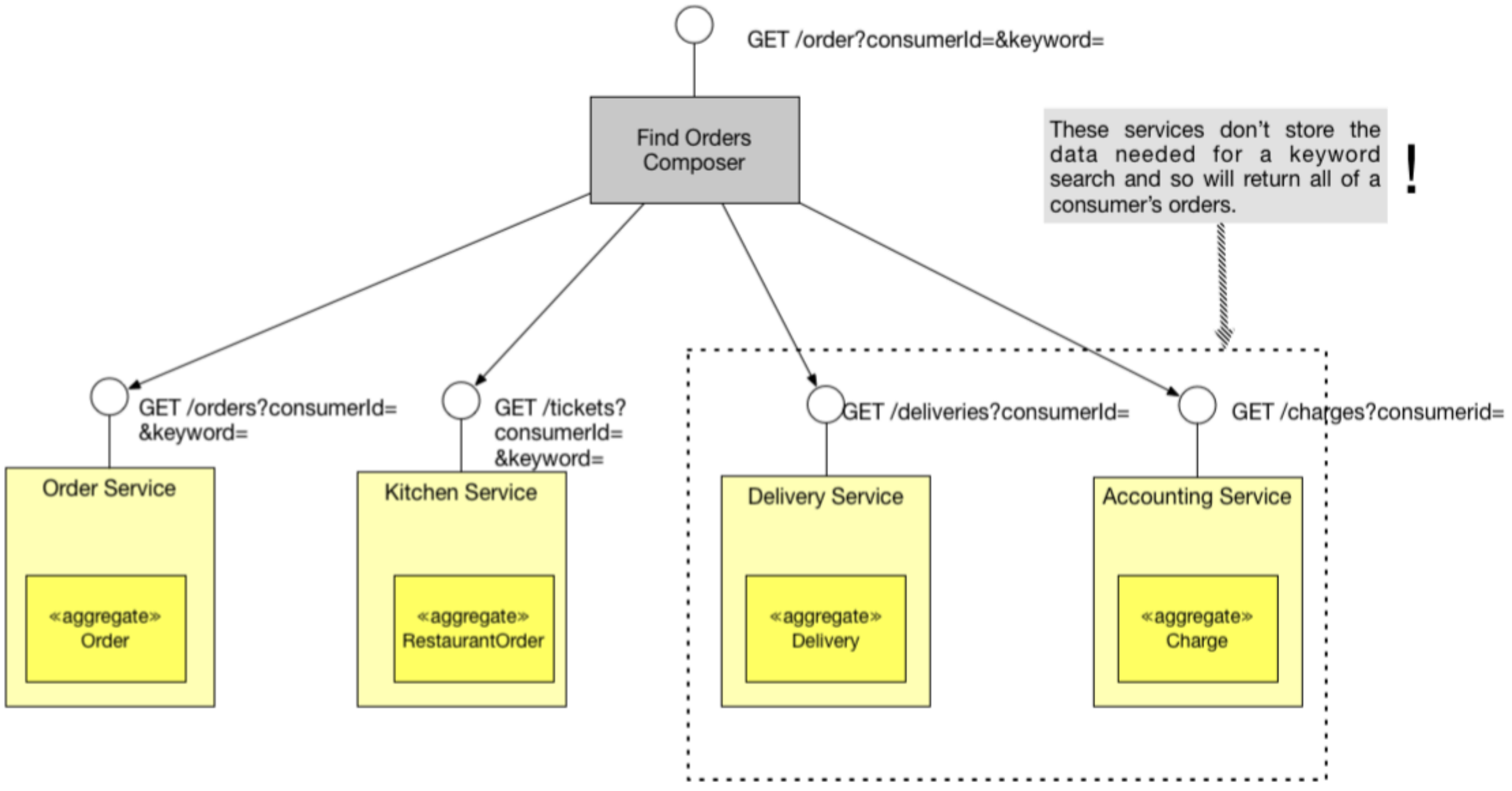
- CQRS é uma generalização deste tipo de arquitetura.
- A ideia é manter um ou mais bancos de dados de visões - não somente os de busca textual - que implementa um ou mais consultas da aplicação

POR QUE USAR CQRS?

- Há situações em que o padrão API Composition não consegue implementar de maneira eficiente a consulta distribuída em vários serviços
- Exemplo:
- 1- Consulta paginada que agrega dados de vários serviços.
- 2- Filtro por um determinado tipo de dado que está em um serviço diferente (ex.: filtrar dados de uma entidade por um atributo que não está em todos os serviços)
- Realizar a paginação ou o filtro em memória de grandes volumes de dados seria impraticável.

POR QUE USAR CQRS?

- Situações em que o padrão API Composition não seria adequado:



POR QUE USAR CQRS?

- Até mesmo em consultas que consideram um único serviço podem ser difícil de implementar, visto que pode acontecer de o serviço não suportar o tipo de consulta requerida.
- Exemplo: recuperar dados de restaurante pela geolocalização.
- Isto pode demandar a replicação de dados em uma estrutura que suporte consultas geoespaciais (ex.: Geospatial Indexing Library for DynamoDB).
- O desafio em usar réplicas é mantê-las atualizadas sempre que houver alguma mudança nos dados originais
- Uma das premissas do CQRS é resolver o problema de sincronização de réplicas.

POR QUE USAR CQRS?

- A demanda por separação de responsabilidades:
  - É preciso levar em consideração que a responsabilidade primária de um serviço é manter os seus dados (ex.: Restaurant Service deverá manter dados sobre restaurantes).
  - Implementar nos serviços consultas com lógicas mais elaboradas para permitir, por exemplo, filtros avançados que considerem dados de outros serviços, seria ultrapassar as responsabilidades do serviço.

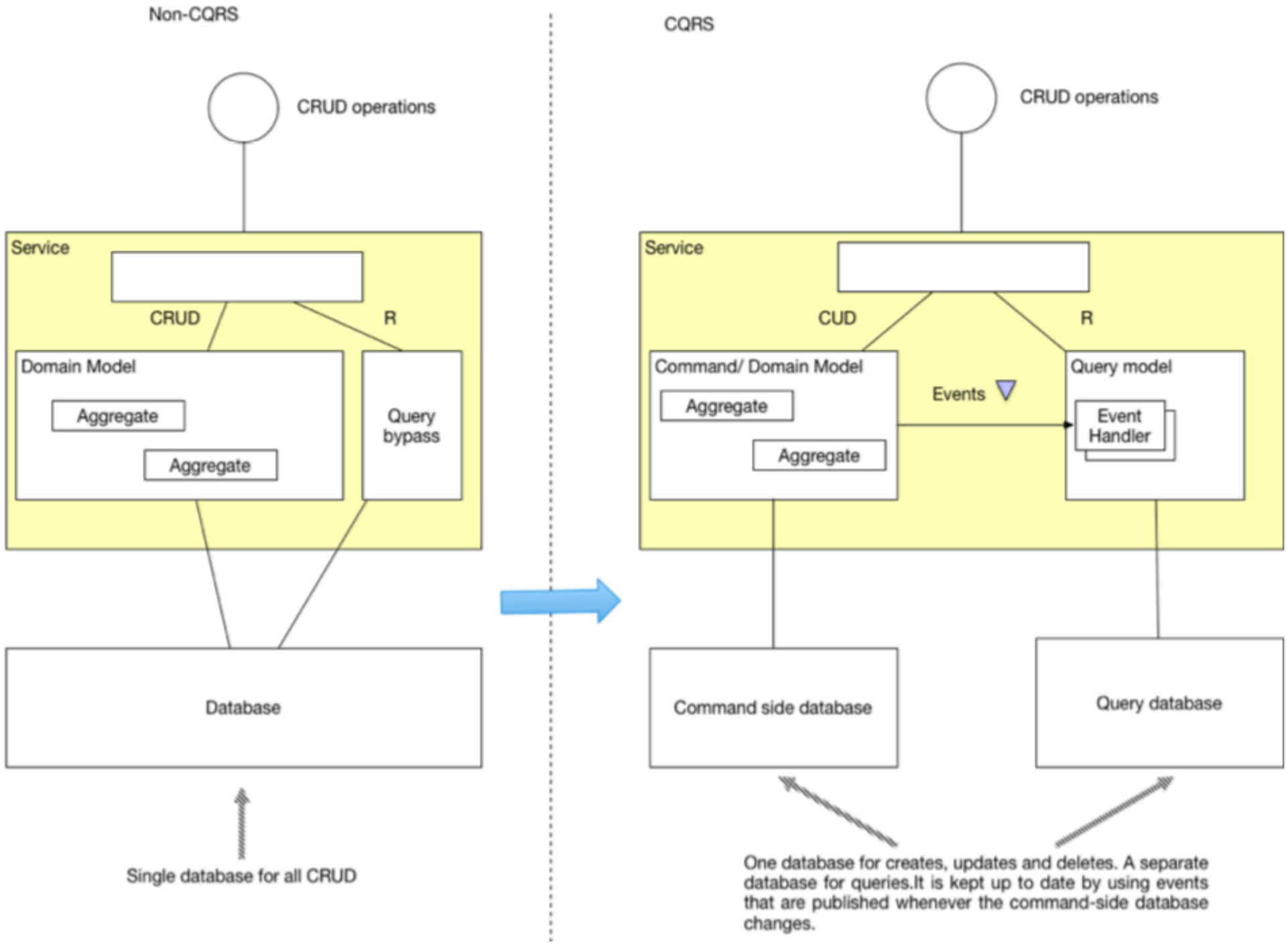
VISÃO GERAL DO CQRS

- Alguns problemas podem ser destacados quando trata-se de realizar consultas numa arquitetura de microsserviços:
  - 1. Usar o padrão API composition para recuperar dados espalhados por múltiplos serviços pode resultar em ineficientes e custosos joins em memória.
  - 2. O serviço que armazena os dados, pode guardá-los de forma ou em um banco de dados que não suporte eficientemente a consulta requerida
  - 3. A necessidade por separar as responsabilidades significa que cada serviço que deve controlar seus dados e não implementar as consultas de operação
- Esses três problemas são resolvidos com o uso de CQRS

CQRS SEPARA COMANDOS DE CONSULTAS

- Como o nome sugere, o padrão diz respeito à separação de responsabilidades.
- A ideia é separar o modelo de dados persistente em duas partes: *command-side* e *query-side*.
  - O módulo de *command-side* implementa as operações de criação, atualização e remoção (ex.: POSTs, PUTs, DELETEs)
  - O módulo de query-side implementa as operações de leitura (ex.: GETs)
- O módulo de query-side se mantém sincronizado com o *command-side* através da subscrita aos eventos gerados pelo *command-side*.

CQRS SEPARA COMANDOS DE CONSULTAS

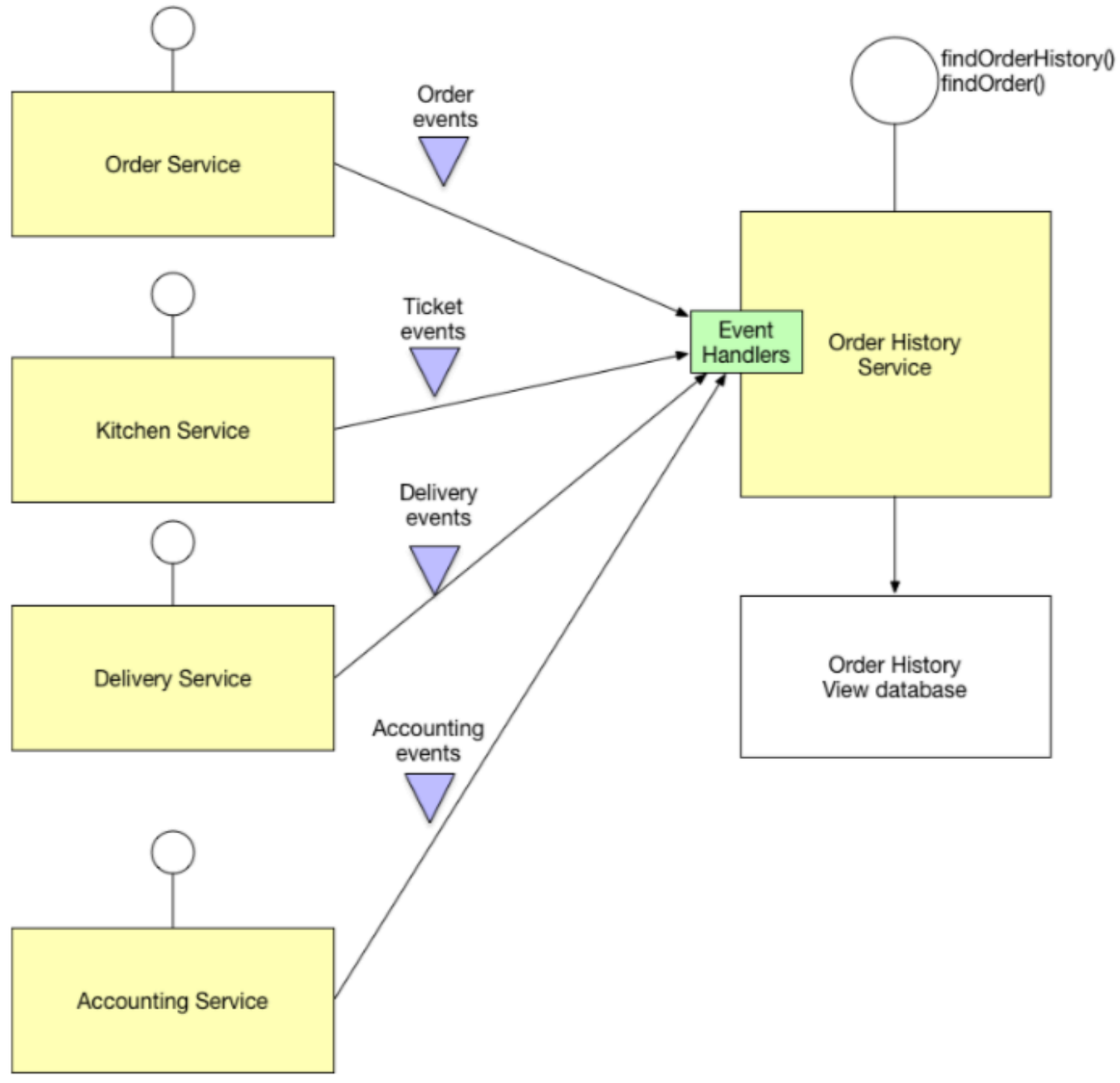


CQRS SEPARA COMANDOS DE CONSULTAS

- Em serviços baseados em CQRS, o command-side realiza as operações de mudanças em entidades e dispara eventos, que podem ser implementados através do uso de um framework (ex.: eventuate tram) ou event-sourcing.
- O query-side consome esses comandos e mantém uma visão atualizada que pode ser consultada de maneira a agregar dados de vários serviços. Podem inclusive ser mantidos múltiplos modelos de consulta (ex.: espaciais, texto, relacional).

CQRS E SERVIÇOS SÓ DE CONSULTA (QUERY-SERVICES)

- É possível modelar um serviço que seja responsável só por consumir os eventos do command-side e fornecer acesso às consultas.



BENEFÍCIOS E DESVANTAGENS DO CQRS

- Benefícios:
  - Permite a implementação eficiente de consultas numa arquitetura de microsserviços
  - Permite a implementação eficiente de vários tipos de consulta
  - Torna possível realizar consultas numa aplicação baseada em event-sourcing
  - Implementa a separação de responsabilidades dos serviços

BENEFÍCIOS E DESVANTAGENS DO CQRS

- Desvantagens:
  - Arquitetura mais complexa
  - Necessidade de lidar com o lag de replicação (o dado atualizado leva um tempo para estar disponível na tabela da visão)
    - Uma possível solução seria prover ao cliente um código de versão, o que indicaria que a versão lida imediatamente estaria desatualizada. Forçando-o a consultar novamente.

CQRS – RESUMO

- CQRS possui várias vantagens e desvantagens.
- CQRS não deve ser usado para todas as consultas na aplicação. Sempre que possível, deve-se adotar o uso de API composition, usando CQRS apenas quando realmente for necessário.

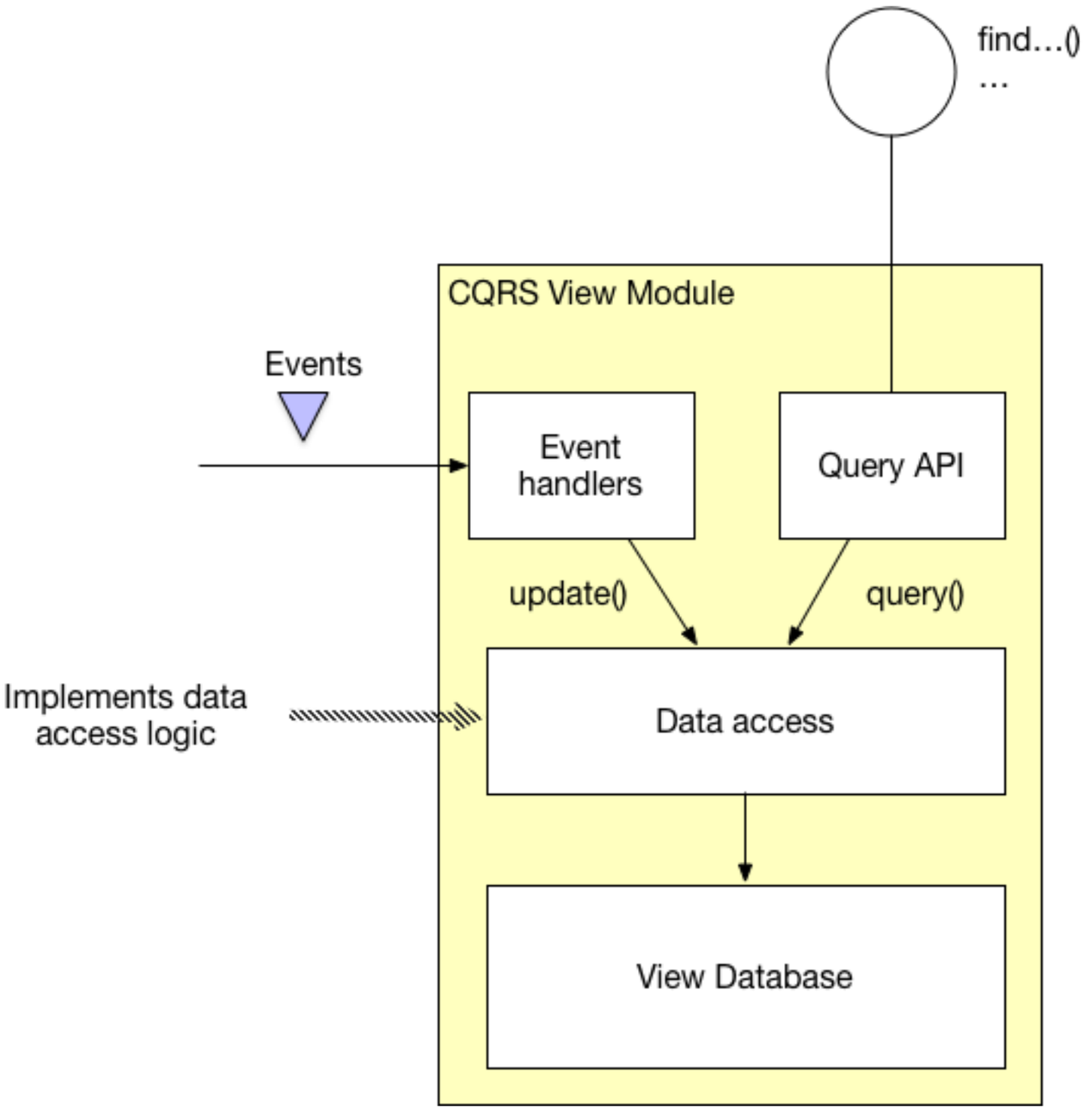
3. MODELANDO VISÕES CQRS

PROJETANDO VISÕES CQRS

- Uma visão CQRS possui uma API que consiste de uma ou mais operações de consulta
- Essas operações são implementadas através da consulta a bancos de dados que são mantidos através da inscrição a eventos publicados por um ou mais serviços.

PROJETANDO VISÕES CQRS

- O modelo de um módulo de visão CQRS.
- Tratados de eventos atualizam o banco da visão, que é consultado pelo módulo Query API



PROJETANDO VISÕES CQRS

- As seguintes decisões devem ser tomadas ao projetar um módulo de visão:
  - Escolher um banco de dados e definir o modelo de dados
  - Ao projetar o módulo "DataAccess", devem ser tratados algumas questões como: garantir que a atualização seja idempotente e o tratamento de atualizações concorrentes
  - Ao implementar uma nova visão numa aplicação existente ou mudar o esquema, deve ser implementado um mecanismo para reconstruir a view de maneira eficiente.
  - Decidir como permitir ao cliente que a visão lide com o lag de replicação

ESCOLHENDO UM BANCO DE DADOS

- O propósito primordial deste banco e do modelo de dados é implementar de maneira eficiente as operações de consulta.
- As características das consultas que serão executadas devem ser levadas em consideração ao selecionar um banco de dados
- O banco deve também implementar de maneira eficiente a atualização das operações realizadas pelos tratadores de eventos

SQL VS NOSQL

- Até um tempo atrás, os bancos relacionais eram a única opção para todos os casos. Com o crescimento da Web, viu-se que os relacionais não satisfazem a demanda no nível da Web. O que levou a criação dos conhecidos bancos NoSQL.
- Um banco NoSQL geralmente tem uma forma limitada de transações e menos funcionalidades de consulta. Para alguns casos, esses bancos têm vantagens em relação so bancos SQL, incluindo um modelo de dados mais flexível e um melhor desempenho e escalabilidade

SQL VS NOSQL

- Um banco NoSQL é geralmente uma boa escolha para uma visão CQRS, o que leva à aproveitar seus lados positivos e ignorar as desvantagens.

- Uma visão CQRS se beneficia do modelo mais rico e do desempenho de um banco NoSql. Da mesma forma, não é afetado pelas limitações de um banco NoSQL, visto que ele usa transações simples e executa um conjunto fixo de consultas.

SQL VS NOSQL

- No entanto, algumas vezes faz sentido implementar uma visão CQRS usando um banco SQL. Um banco relacional moderno rodando em bom hardware possui excelente desempenho. Desenvolvedores, DBAs e operadores de TI são, geralmente, mais familiarizados com bancos SQL do que com bancos NoSQL.

- Além disso, bancos SQL geralmente possui extensões para recursos não-relacionais, como o suporte a consultas e tipos geoespaciais ou colunas em JSON. Uma visão CQRS também pode precisar usar um banco SQL para dar suporte a uma ferramenta de geração de relatórios.

ESCOLHENDO UM BANCO DE DADOS

Se você precisa…	então use…	por exemplo…
Recuperação de objetos JSON com base em chave primárias	Um BD baseado em documentos, como MongoDB/DynamoDB, ou até mesmo um em chaves-valor como Redis.	Implementar histórico de pedidos mantendo um Document no Mongodb contendo a informação por cliente
Recuperação de objetos JSON com base em consultas	Um BD baseado em documentos, como MongoDB ou DynamoDB	Implementar uma visão de "clientes" usando MongoDB ou DynamoDB
Consultas baseadas em texto	Um mecanismo de buscas textuais como Elastic Search	Implementar busca baseada em textos para pedidos através da manutenção de um documento Elastic Search por pedido
Consultas em grafos	Um banco de dados de grafos, como o Neo4j	Implementar detecção de fraude através da manutenção de um grafo de pedidos, clientes e outros dados
Relatórios SQL Tradicionais / Business Intelligence	Um banco de dados relacional	Relatórios e análises padrões de negócio

SUPORTE A OPERAÇÕES DE ATUALIZAÇÃO

- Geralmente as operações são realizadas a partir da chave-primária do objeto. No entanto, algumas vezes é demandada a atualização pela 'chave estrangeira' (Ex.: atualizar a entrega correspondente a um pedido).
- Alguns tipos de bancos de dados possuem suporte à operações de atualização baseado em foreign-key. No caso de bancos relacionais ou MongoDB, basta criar um índice nas colunas necessárias. No entanto, atualização que não usam chave primária pode ser mais complexo em outros tipos de bancos NoSQL.
- Nestes casos, aplicação terá de manter algum tipo de mapeamento específico da chave estrangeira para determinar qual registro atualizar.

MODELANDO O MÓDULO “DATA ACCESS”

- Os tratadores de eventos e a o módulo da API de consultas não acessa o banco de daods diretamente
- Eles usam o módulo “Data Access”, que consiste de um DAO e suas respectivas classes utilitárias
- O DAO neste caso tem várias responsabilidades: implementa operações de atualização chamadas pelos tratadores de eventos e as operações de consulta chamadas pelo módulo de consultas.
- O DAO faz o mapeamento entre os tipos de dados usados no código de alto nível e a API do banco de dados.
- Ele também pode tratar as atualizações concorrentes, garantindo que as atualizações seja idempotentes.

LIDANDO COM A CONCORRÊNCIA

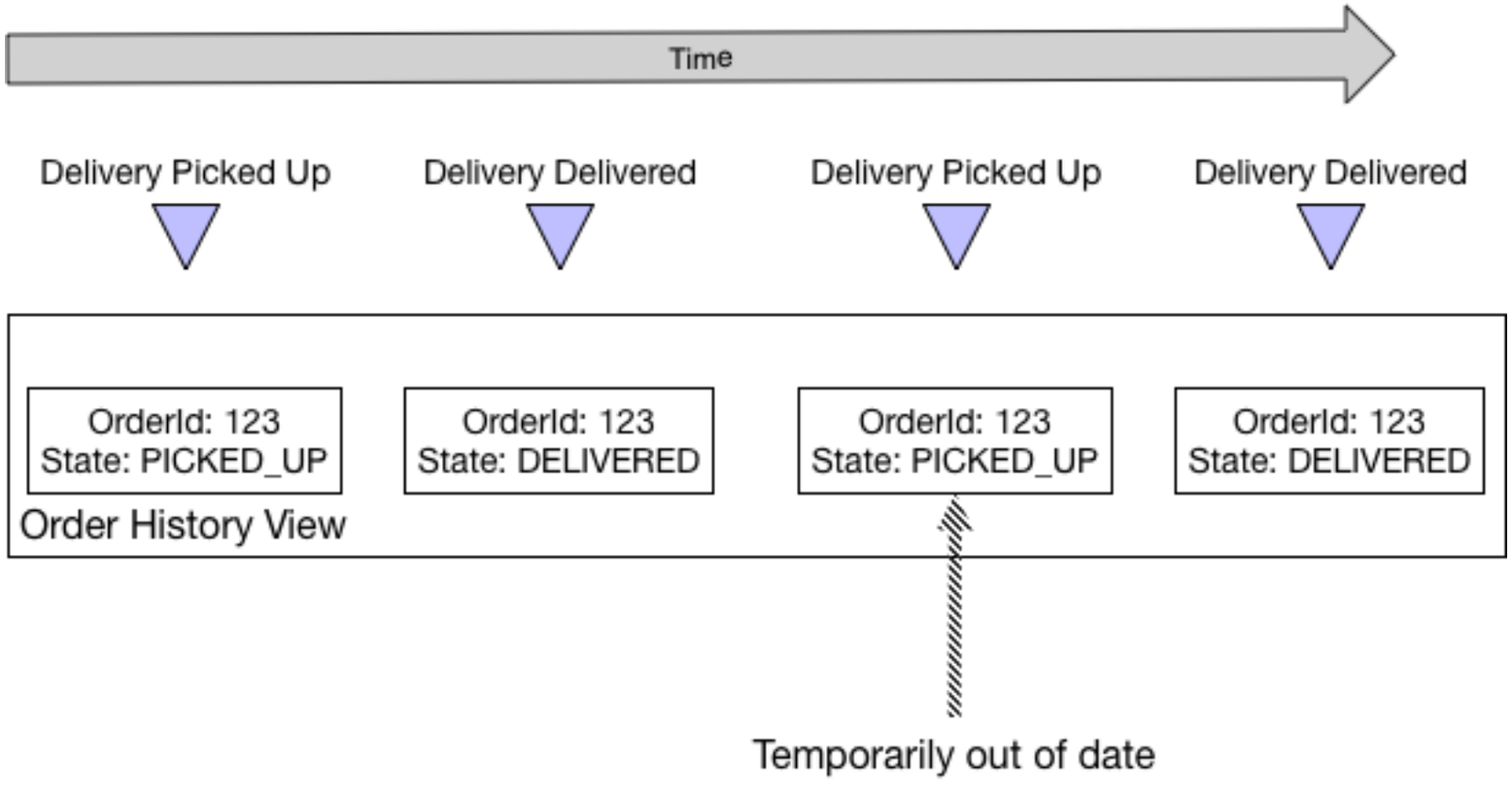
- Eventualmente um DAO pode lidar com atualizações concorrentes de um mesmo registro no banco.
- Se uma visão se inscreve a eventos publicados por um único aggregate, não deve haver qualquer concorrência, visto que os eventos publicados por uma aggregate particular são processados sequencialmente. No entanto, uma visão que se inscreve a eventos publicados por múltiplas aggregates corre o risco de múltiplos tratadores de eventos atualizar o mesmo registro de maneira simultânea.
- Exemplo, se um tratador de evento para “Pedido" for chamado ao mesmo tempo que um tratador de evento para “Entrega”. Ambos os tratadores irão chamar o DAO para atualizar o registro no banco para “Pedido”.
- O DAO precisa garantir que uma atualização não sobrescreva a outra. Se um DAO implementa atualizações através da leitura de um registro para então escrever a atualização, ele pode usar locking pessimista ou otimista.

TRATADORES DE EVENTOS IDEMPOTENTES

- Um tratador de evento pode ser invocado mais de uma vez para o mesmo evento.
- Isto geralmente não é um problema se o tratador de eventos do query-side for idempotente, isto é, se mesmo que haja duplicação de eventos, a saída será sempre correta.

TRATADORES DE EVENTOS IDEMPOTENTES

- Exemplo do caso de evento duplicado do estado “PICKED_UP” que é executado após o pedido ter passado pro estado “DELIVERED".



TRATADORES DE EVENTOS IDEMPOTENTES

- Para resolver esse problema, o tratador de eventos devem gravar o id do evento e atualizar o banco de maneira atômica (ou seja, se o evento não foi concluído, então não grave o ID do evento como recebido).
- Se implementado como um banco NoSQL, o tratador de eventos pode simplesmente inserir os eventos processados numa tabela PROCESSED_EVENTS como parte da atualização que atualiza a view.
- Se for utilizado um banco Nosql, que possui um modelo de transações limitados, o tratador deverá salvar o evento dentro do registro (ex.: MongoDB document) que o banco atualiza.

TRATADORES DE EVENTOS IDEMPOTENTES

- É importante reforçar que os tratadores de eventos não precisam necessariamente armazenar os IDs de todos os eventos gerados.
- Pode-se incrementar o ID de cada registro a cada inserção e manter apenas a referência do último id gerado max(eventId) que é recebido de uma instância de uma aggregate.
- Exemplo (no DynamoDB) para guardar a referência do último evento executado para uma aggregate de determinado tipo:

Tipo da Aggregate

ID da Aggregate

ID do evento

```
{…
   "Order3949384394-039434903" : "0000015e0c6fc18f-0242ac1100e50002",
   "Delivery3949384394-039434903" : "0000015e0c6fc264-0242ac1100e50002",
}
```

ADICIONANDO E ATUALIZANDO CQRS VIEWS

- Visões CQRS serão adicionadas e atualizadas durante todo o tempo de vida da aplicação.
- Algumas vezes é necessário adicionar uma nova visão para suportar uma nova consulta.
- Outras vezes é preciso recriar uma visão por causa de mudanças no esquema do banco ou para corrigir algum bug no código que atualiza a visão.
- Para criar uma nova visão, basta fazer as alterações no módulo query-side, ajustar o banco de dados e implantar o serviço
  - Os tratadores de eventos do módulo query-side processam todos os eventos e a visão estará atualizada.
- Da mesma forma, atualizar uma visão é simples: basta ajustar os tratadores de eventos e reconstruir os dados da visão desde o início.
- No entanto, essa abordagem possui alguns desafios, descritos a seguir.

CONSTRUINDO VISÕES USANDO EVENTOS ARQUIVADOS

- Um problema é que os message brokers não guardam mensagens indefinidamente.
- MB tradicionais como o RabbitMQ deletam as mensagens após o consumo.
- Mesmo o Kafka, que guarda as mensagens, mantém elas por apenas um período.
- Neste caso, a aplicação deverá ler eventos que foram arquivados em outra fonte (ex.: AWS S3), usando tecnologias de processamento escaláveis, para Big Data, como a tecnologia Apache Spark.

CONSTRUINDO VISÕES CQRS DE MANEIRA INCREMENTAL

- Outro problema com a criação da visão é que o tempo e os recursos requeridos para processar todos os eventos continua crescendo ao longo do tempo.
- Eventualmente, criar uma visão se tornará lento e custoso.
- A solução seria usar um algoritmo incremental em duas etapas:
  - Na primeira, periodicamente (ex.: diariamente) seria computado um snapshot de cada instância de aggregate baseado os snapshots anteriores e eventos que ocorreram desde que o último snapshot foi criado
  - Na segunda etapa, a visão seria criada usando os snapshots e quaisquer eventos subsequentes a ela.

RESUMO

- Implementar consultas que recuperam de múltiplos serviços é desafiador, já que o dado está restrito a cada serviço.
- Foram apresentadas duas maneiras de implementar esse tipo de consulta: o padrão API Composition e o padrão Command Query Responsibility Segregation (CQRS).
- O padrão API Composition, que combina dados de múltiplos serviços, é a forma mais simples de implementar consultas e deve ser usada sempre que possível.
- Uma limitação do padrão API composition é que consultas mais complexas requerem ineficientes in-memory joins em datasets de larga-escala.
- O padrão CQRS, que implementa consultas usando bancos de dados de visões, é mais poderoso, porém é mais complexo para implementar
- Um módulo de visão CQRS deve tratar atualizações concorrentes, como também detectar e descartar eventos duplicados
- CQRS melhor a separação de responsabilidades, permitindo que um serviço implemente uma consulta que retorna dados pertencentes a um serviço diferente.
- Aplicações clientes precisam tratar a eventual consistência das visões CQRS.