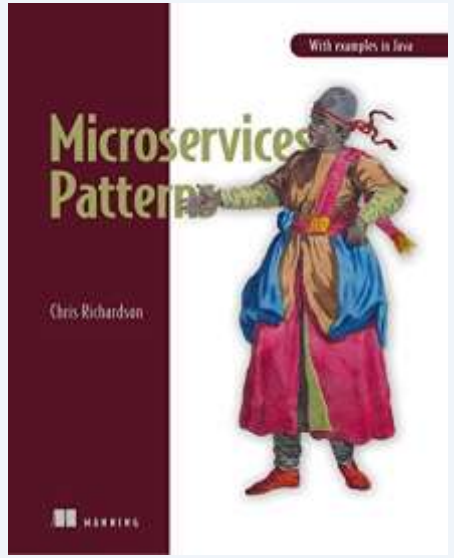
API Queries

The API Composition Pattern and CQRS

Objectives

- The need for an API query strategy
- Context & Problem
- Solution 1: The API Composition Pattern
- Solution 2: The CQRS Pattern



API Queries

- This is another problem arising from the microservices architecture pattern!

- Most applications will enable a set of queries through their API
  - getFarmerProfile()
  - findOrder()
  - findOrderHistory()

- In a monolithic architecture, these are easy to accommodate with appropriately written SQL SELECT statements to the shared relational DB.

Types of queries

- Single service queries
  - These are easy – just a SELECT statement from the service DB
  - E.g. getFarmerProfile()

- Cross service queries
  - These are harder – the information is not stored in any one database
  - Even if it could be implemented, a distributed query would violate encapsulation.
  - E.g. findOrder(), findOrderHistory()

Context & Problem

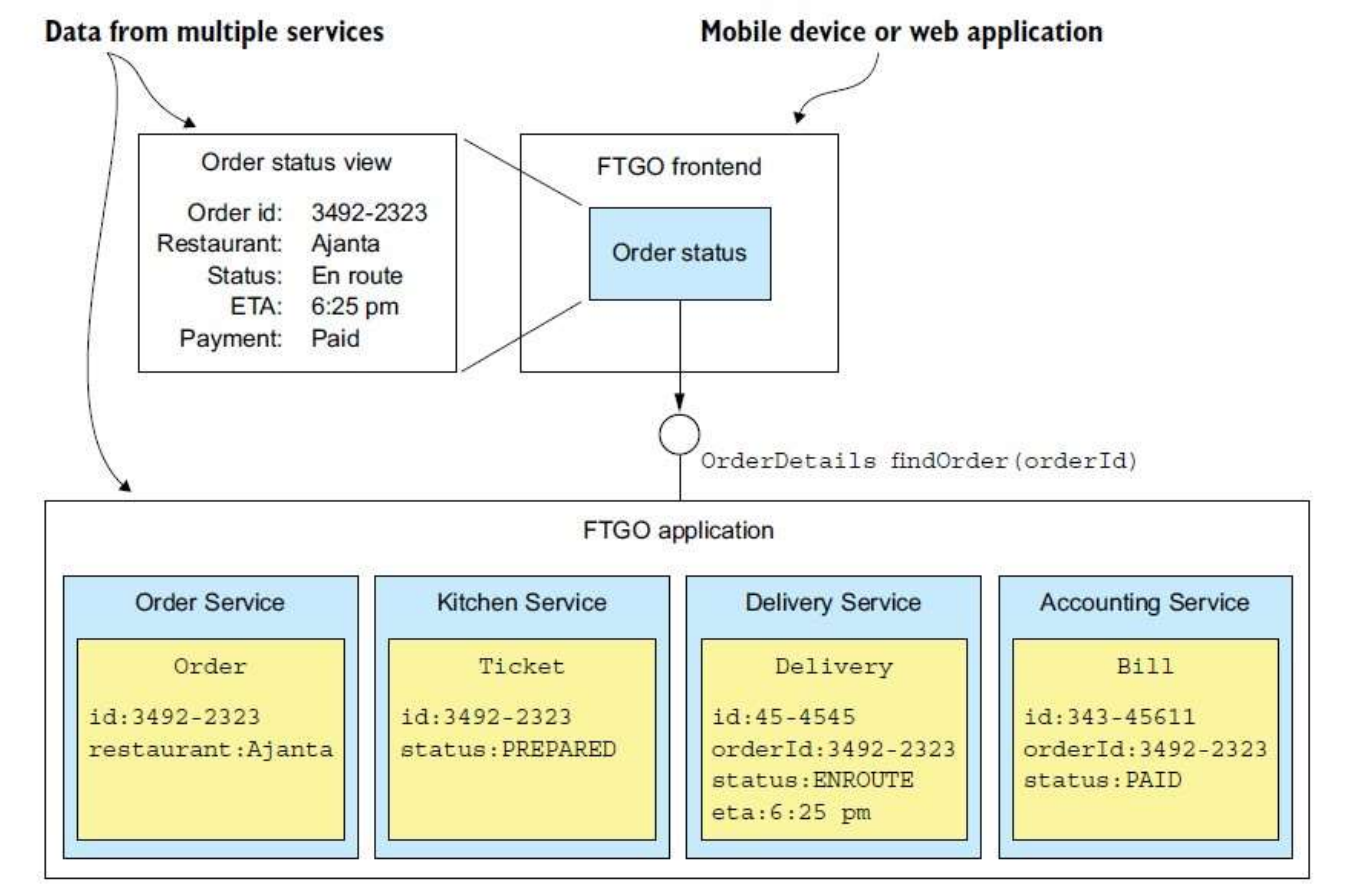
- Your service implements the Microservices Architecture Pattern
- You have also enforced the Database per Service Pattern

- The problem is then – How to implement queries?

- The only real force involved here is the need to have a solution for implementing queries.

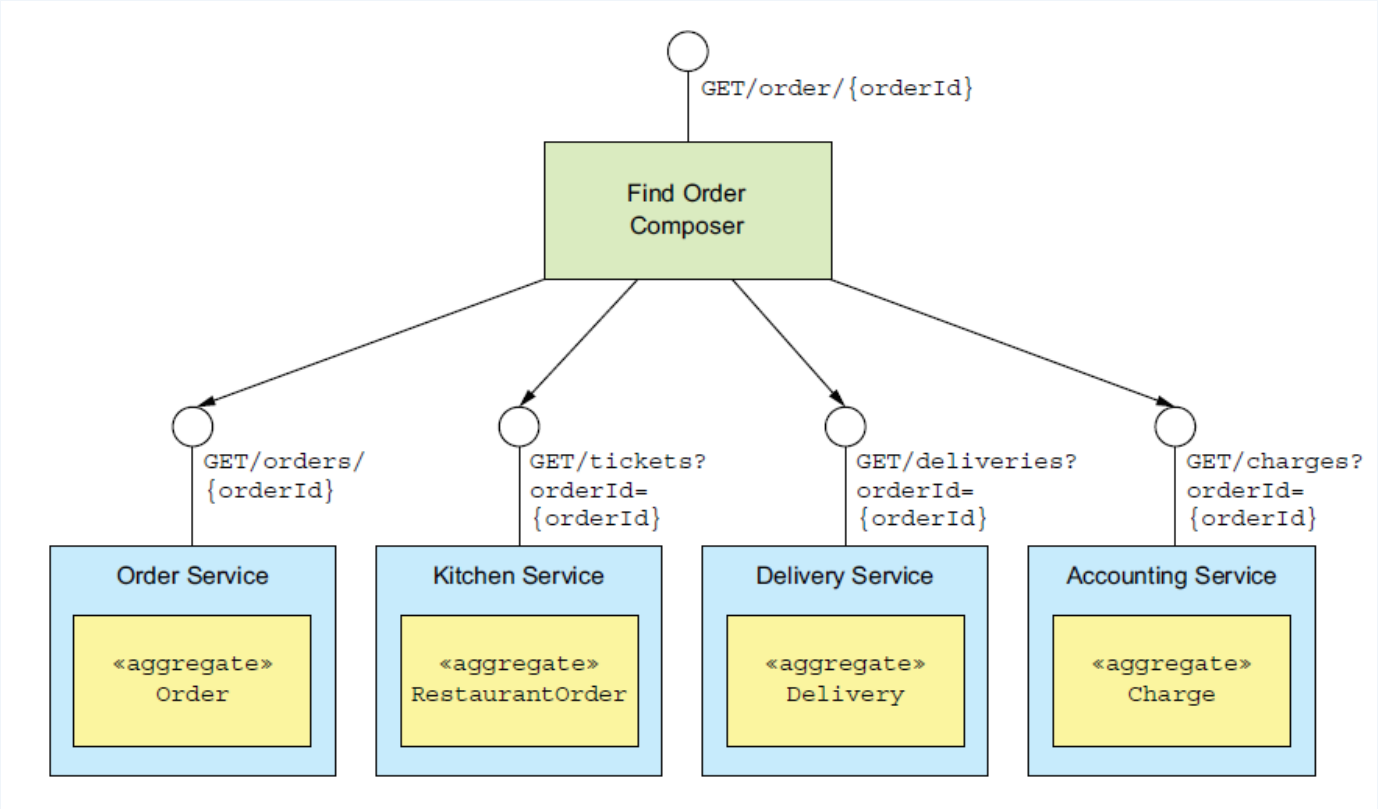
Context and Problem

- The client requires data from multiple services to complete the query



Solution 1: The API Composition Pattern

- A query is implemented by defining an API Composer
  - This invokes the services owning the necessary data and performs and in-memory join of the result



Example: Include with API Gateway

- An application most often has a single API Gateway service to route requests to the application services.
  - This often includes API composition.

Resulting Context

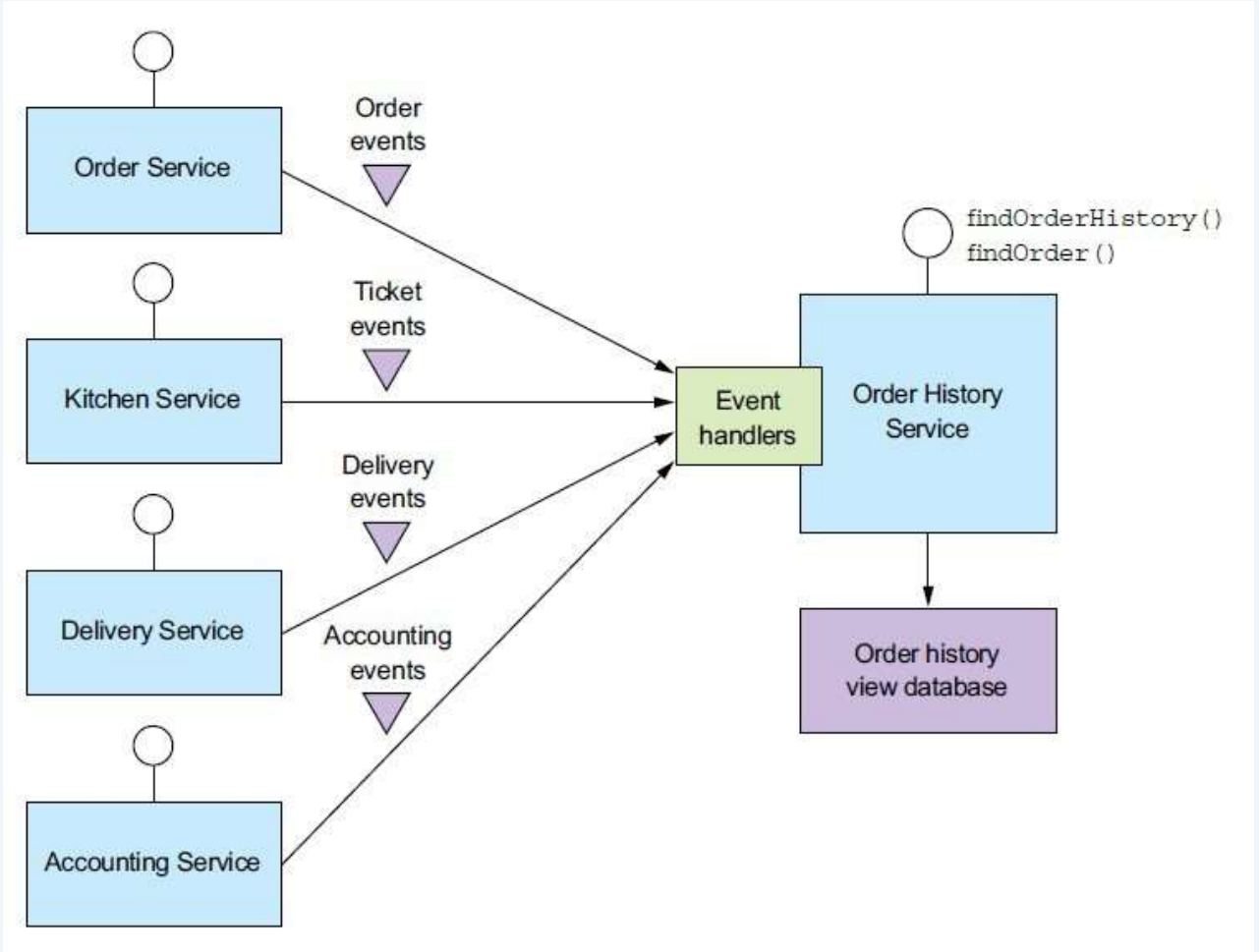
- Benefits
  - This is a very simple way to query data in a microservices architecture
- Drawbacks
  - Increased overhead – multiple services and databases queried.
  - In some cases in-memory joins of large datasets are required, which would be inefficient to implement this way → risk of reduced availability.
  - Lack of transactional data consistency
    - Because of multiple queries, there's a risk the data returned may not be consistent.
- The rule of thumb – use an API composer unless you really can't.

Solution 2: The CQRS Pattern

- Command-Query-Responsibility-Segregation
- Commands (create, update, delete) and queries (retrieve) are treated differently because the way they affect the application is different.
- The application keeps a view database, which is a read-only replica designed to support the query.
  - The replica is kept up to date by subscribing to domain events published by the service(s) that own the data.

CQRS Implementation

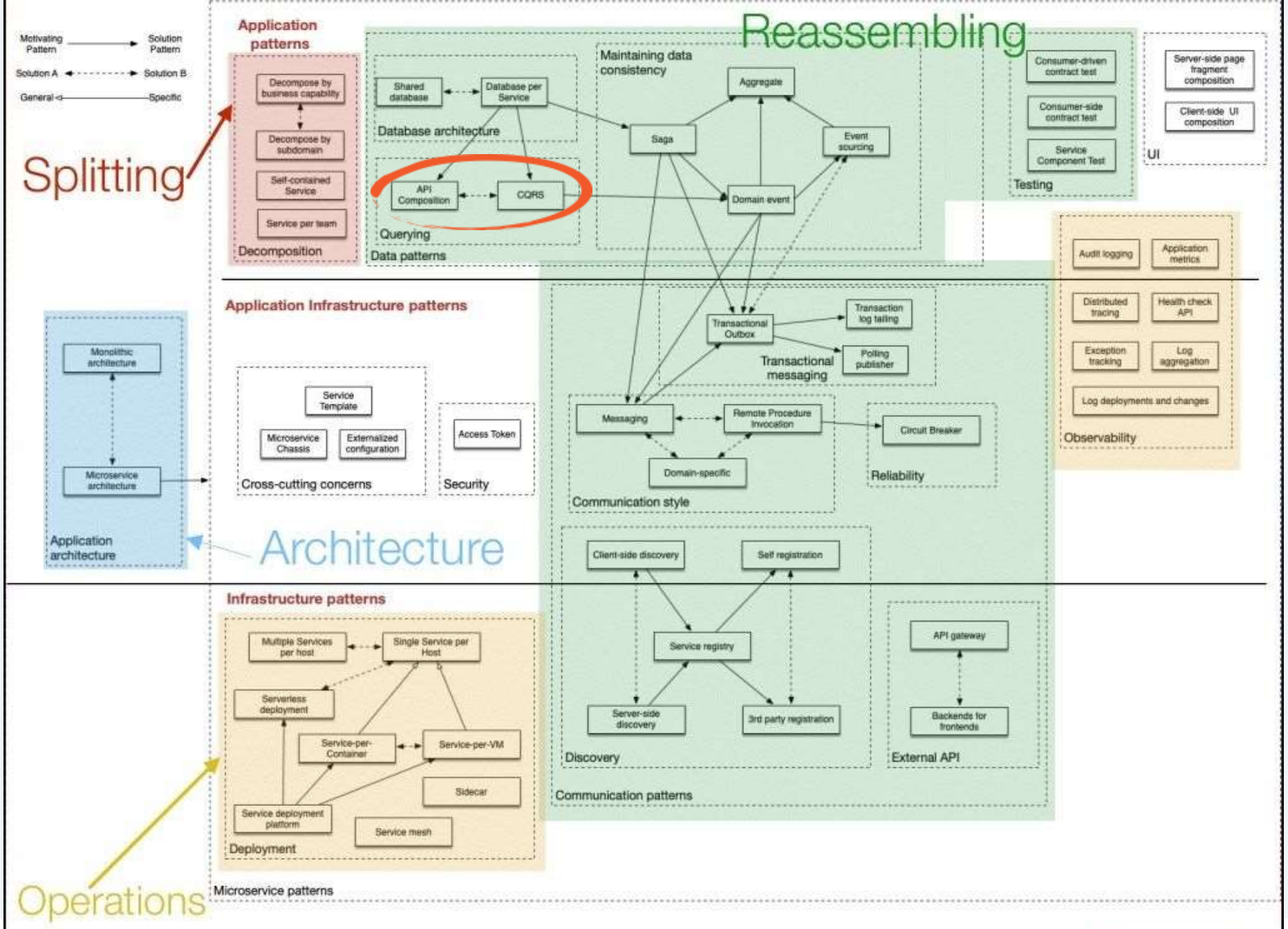
- This is all about separation of concerns.
- The view database doesn't belong to any particular service, so it makes sense to implement it as a standalone service.
- Replicates data that exists in other service(s), but is optimized for the particular queries allowed.



Resulting Context

- Benefits
  - Enables efficient implementation of queries
  - Enables efficient implementation of **diverse** queries
  - Makes querying possible in an event sourcing based application
  - Improves the separation of concerns

- Drawbacks
  - More complex architecture
  - Must deal with replication lag
    - There is a natural lag between the command-side and query-side views.
    - How does the client know if the data is out of date?
    - One solution – supply version info to the client, they can poll for updated information.



Learn-Build-Assess Microservices http://adopt.microservices.io

Summary

- The need for an API query strategy
- Context & Problem
- Solution 1: The API Composition Pattern
- Solution 2: The CQRS Pattern

Questions or Comments?