Microservice Architecture (/index.html)

Supported by Kong (https://github.com/Mashape/kong)

Pattern: Command Query Responsibility Segregation (CQRS)

Context

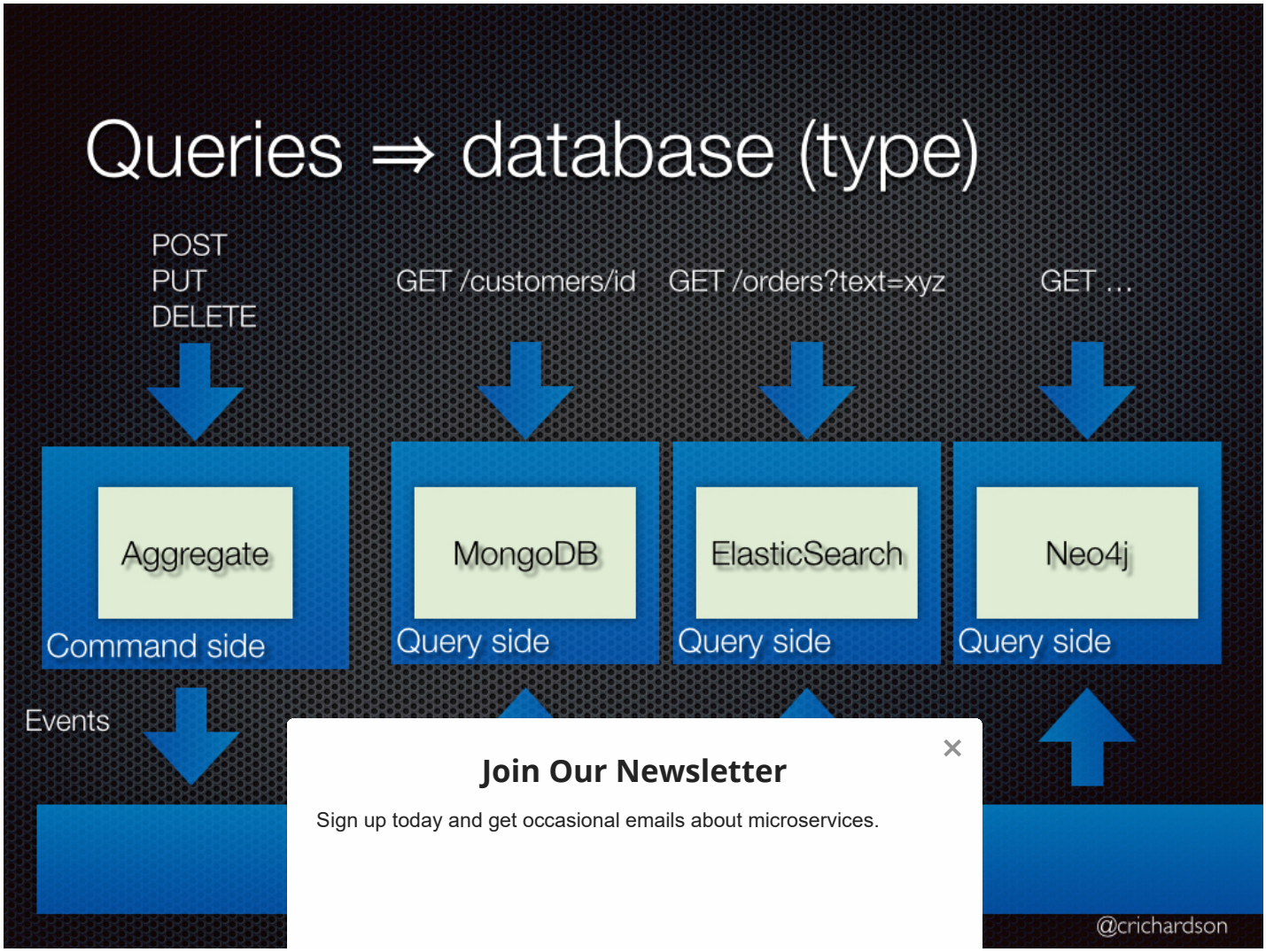
You have applied the Microservices architecture pattern (../microservices.html) and the Database per service pattern (database-per-service.html). As a result, it is no longer straightforward to implement queries that join data from multiple services. Also, if you have applied the Event sourcing pattern (event-sourcing.html) then the data is no longer easily queried.

Problem

How to implement queries in a microservice architecture?

Solution

Split the application into tw de handles create, update, and delete reques queries by executing them against one or more stream of events emitted when data changes.



Examples

Customers and Orders (https://github.com/eventuate-examples/eventuate-examples-java-customers-and-orders) is an example of an application that is built using Event Sourcing and CQRS (cqrs.html). The application is written in Java, and uses Spring Boot. It is built using Eventuate (http://eventuate.io), which is an application platform based on event sourcing and CQRS.

This application maintains a CQRS view in MongoDB. Each document contains information about a customer and their orders. The view is updated by subscribing to customer and order events:

```
@EventSubscriber(id = "orderHistoryWorkflow")
public class OrderHistoryViewWorkflow {

  private OrderHistoryViewService orderHistoryViewService;

  @Autowired
  public OrderHistoryViewWorkflow(OrderHistoryViewService orderHistoryViewService) {
    this.orderHistoryViewService = orderHistoryViewService;
  }

  @EventHandlerMethod
  public void createCustomer(DispatchedEvent<CustomerCreatedEvent> de) {
    String customerId = de.getEntityId();
    orderHistoryViewService.createCustomer(customerId, de.getEvent().getName(),
            de.getEvent().getCreditLimit());
  }

  @EventHandlerMethod
  public void createOrder(DispatchedEvent<OrderCreatedEvent> de) {
    String customerId = de.getEvent().getCustomerId();
    String orderId = de.getEntityId();
    Money orderTotal = de.getEvent().getOrderTotal();
    orderHistoryViewSe
  }

  @EventHandlerMethod
  public void orderAp
    String customerId
    String orderId = (
    orderHistoryViewSe

  @EventHandlerMethod
  public void orderRejected(DispatchedEvent<OrderRejectedEvent> de) {
    String customerId = de.getEvent().getCustomerId();
    String orderId = de.getEntityId();
    orderHistoryViewService.rejectOrder(customerId, orderId);
  }
```


`OrderHistoryViewService` uses Spring Data for MongoDB to update MongoDB.

There are several example applications (http://eventuate.io/exampleapps.html) that illustrate how to use event sourcing.

Resulting context

This pattern has the following benefits:

- Necessary in an event sourced architecture
- Improved separation of concerns = simpler command and query models
- Supports multiple denormalized views that are scalable and performant

This pattern has the following drawbacks:

- Increased complexity

- Potential code duplication
- Replication lag/eventually consistent views

Related patterns

- The Database per Service pattern (database-per-service.html) creates the need for this pattern
- The API Composition pattern (api-composition.html) is an alternative solution
- The Saga pattern (/patterns/data/saga.html) pattern generates the event stream
- The Event sourcing (event-sourcing.html) is often used with CQRS

See also

- Eventuate (http://eventuate.io), which is a platform for developing applications with Event Sourcing and CQRS
- Articles about event sourcing and CQRS (http://eventuate.io/articles.html)