CSC12108

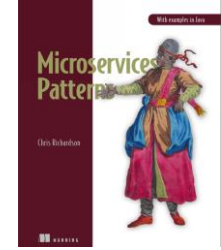
Distributed Application

TOPIC

Implementing queries in a microservice architecture

Instructor – Msc. PHAM MINH TU

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN



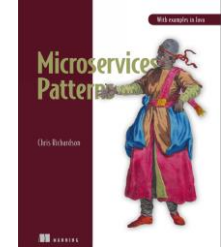
Outline

- Querying using the API composition pattern
- Using the CQRS pattern
- Designing CQRS views
- Implementing a CQRS view with AWS DynamoDB

pmtu@fit.hcmus.edu.vn

Outline

- Querying using the API composition pattern
- Using the CQRS pattern
- Designing CQRS views
- Implementing a CQRS view with AWS DynamoDB

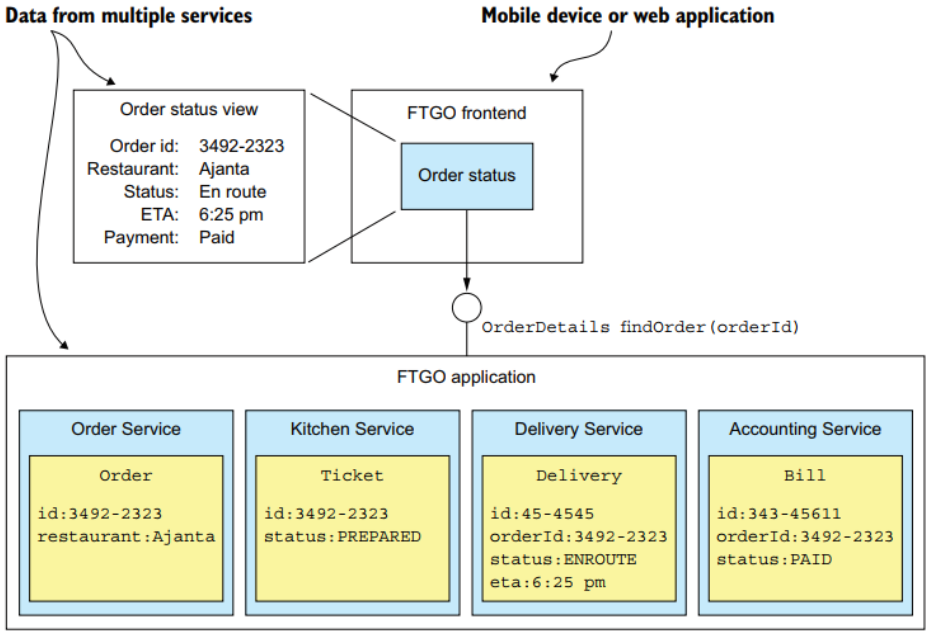


pmtu@fit.hcmus.edu.vn

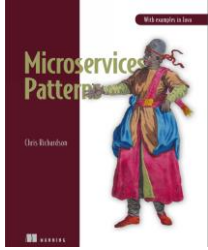
Querying using the API composition pattern

The findOrder() query operation

- The **monolithic** FTGO application can easily retrieve the order details by executing a single **SELECT** statement that **joins** the various tables
- The **microservices-based** version of the FTGO application, the data is scattered around the following services:
  - **Order Service**—Basic order information, including the details and status
  - **Kitchen Service**—Status of the order from the restaurant's perspective and the estimated time it will be ready for pickup
  - **Delivery Service**—The order's delivery status, estimated delivery information, and its current location
  - **Accounting Service**—The order's payment status



pntu@fit.hcmus.edu.vn



Querying using the API composition pattern

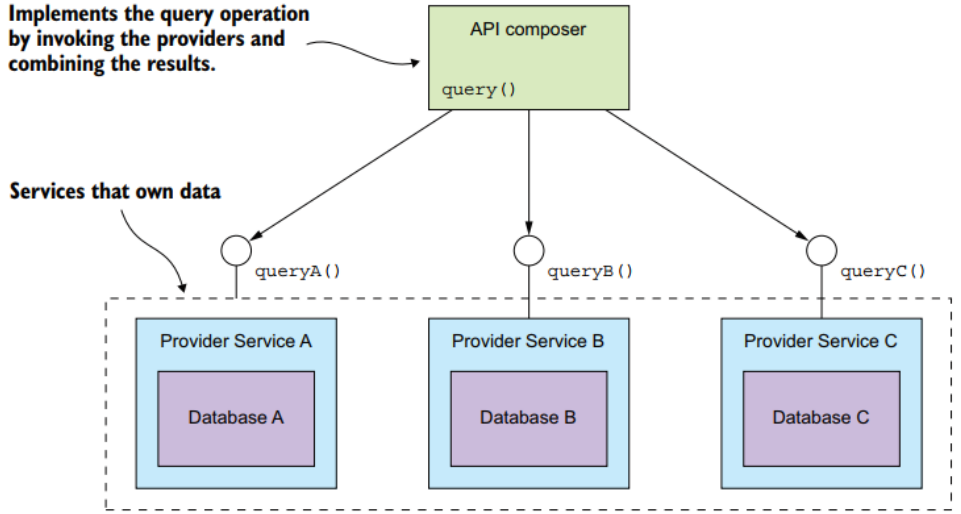
Overview of the API composition pattern

- One way to implement query operations that retrieve data owned by multiple services is to use the **API composition pattern.**
- Two types of participants
  - **API composer**—This implements the query operation by querying the provider services.
  - **Provider service**—This is a service that owns some of the data that the query returns.

pmtu@fit.hcmus.edu.vn

Querying using the API composition pattern

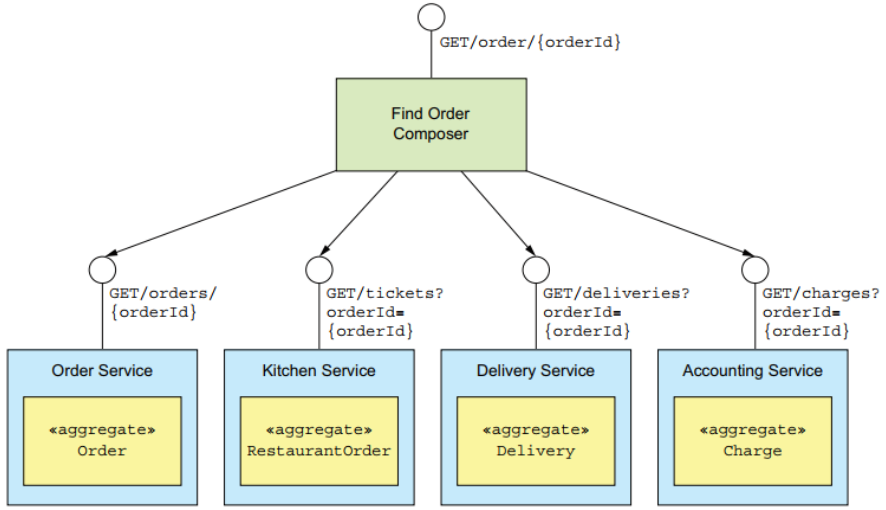
- Overview of the API composition pattern
  - The API composer implements the query by retrieving data from the provider services and combining the results
  - An API composer might be a client, such as a web application
  - An API composer also might be a service, such as an API gateway.



pntu@fit.hcmus.edu.vn

Querying using the API composition pattern

- Implementing the findOrder() query operation using the API composition pattern
  - The API composer is a service that exposes the **query as a REST endpoint**
  - The **Provider services** also implement **REST APIs**
  - The **Find Order Composer** implements a REST endpoint GET /order/{orderId}. It **invokes the four services** and **joins** the responses using the **orderId**
  - Each **Provider service** implements a **REST endpoint** that returns a response corresponding to a single aggregate
  - The **OrderService** retrieves its version of an **Order** by **primary key** and the other services use the **orderId** as a **foreign key** to retrieve their aggregates



pntu@fit.hcmus.edu.vn

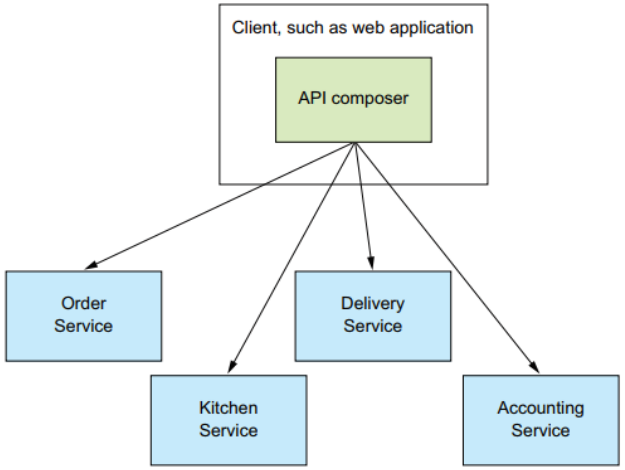
Querying using the API composition pattern

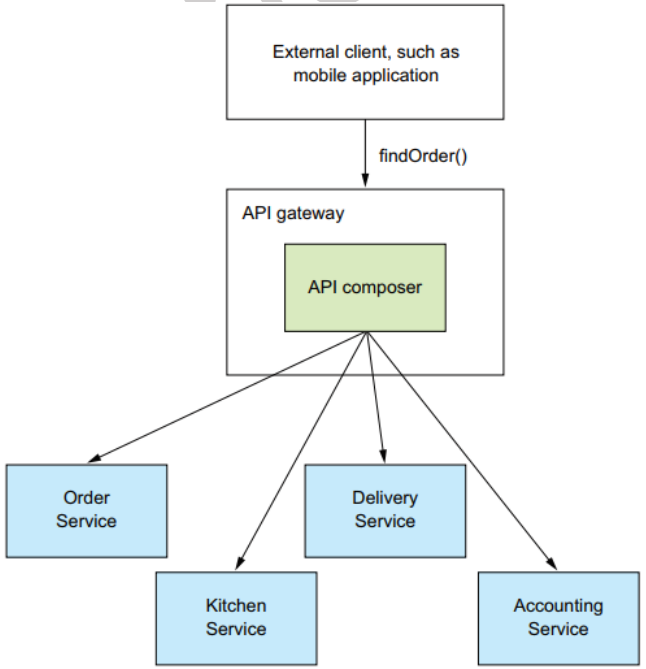
- API composition design issues
  - Who plays the role of the api composer?

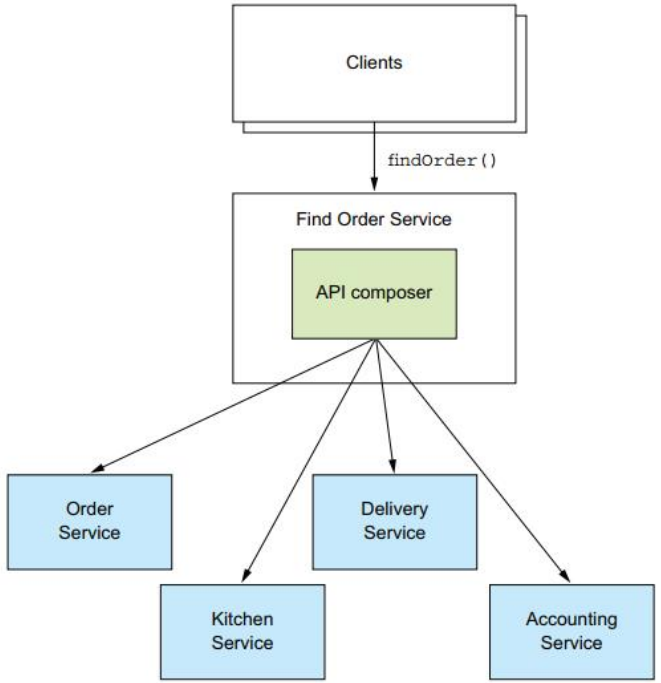
pmtu@fit.hcmus.edu.vn

Querying using the API composition pattern

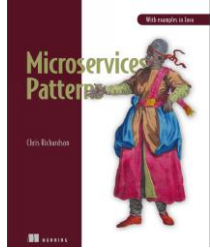
- API composition design issues
  - Who plays the role of the api composer?







hcmus.edu.vn



Querying using the API composition pattern

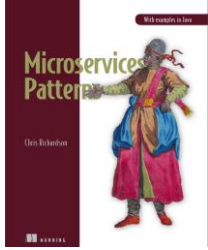
❑API composition design issues

❑Api composers should use a reactive programming model

- ❑Whenever possible, an API composer should call provider services in **parallel** in order to **minimize** the **response time** for a query operation
- ❑Sometimes, though, an API composer needs the result of one Provider service in order to invoke another service

➡ The logic to efficiently execute a mixture of **sequential** and **parallel** service invocations can be complex - **CompletableFuture's, RxJava**

pmtu@fit.hcmus.edu.vn



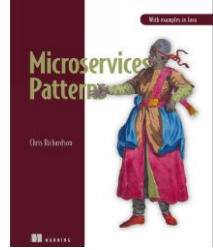
Querying using the API composition pattern

- Benefits and drawbacks
  - This pattern is a simple and intuitive way to implement query operations in a microservice architecture.
  - Drawbacks
    - Increased overhead
    - Risk of reduced availability
    - Lack of transactional data consistency

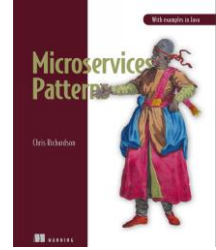
pmtu@fit.hcmus.edu.vn

Outline

- Querying using the API composition pattern
- **Using the CQRS pattern**
- Designing CQRS views
- Implementing a CQRS view with AWS DynamoDB

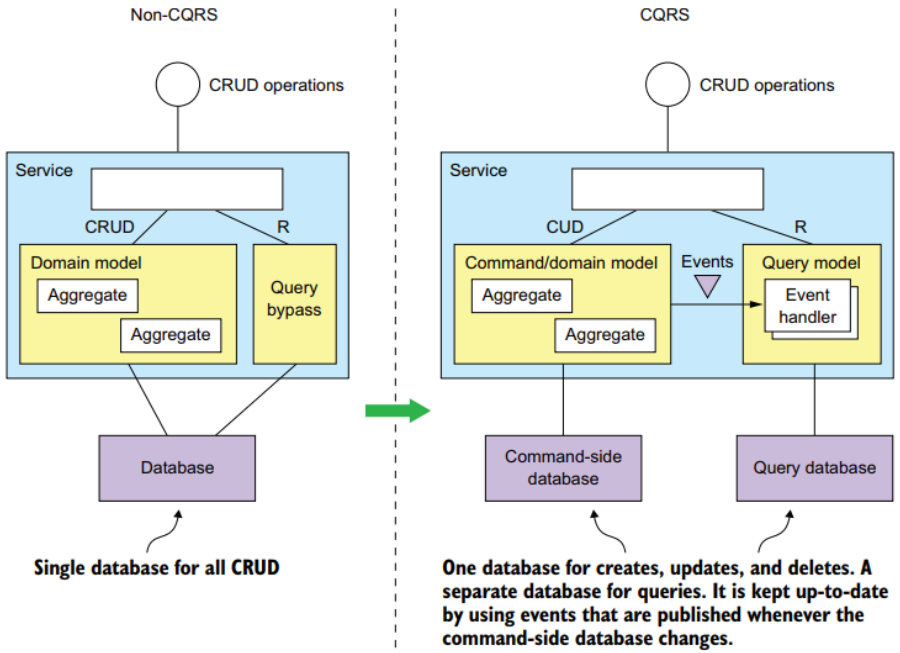


pmtu@fit.hcmus.edu.vn

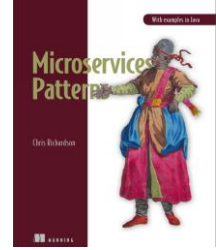


Using the CQRS pattern

- Command query responsibility segregation
  - Overview
    - It splits a persistent data model and the modules that use it into two parts: the command side and the query side.
    - The command side modules and data model implement create, update, and delete operations (HTTP POSTs, PUTs, and DELETEs)
    - The query-side modules and data model implement queries (HTTP GETs)

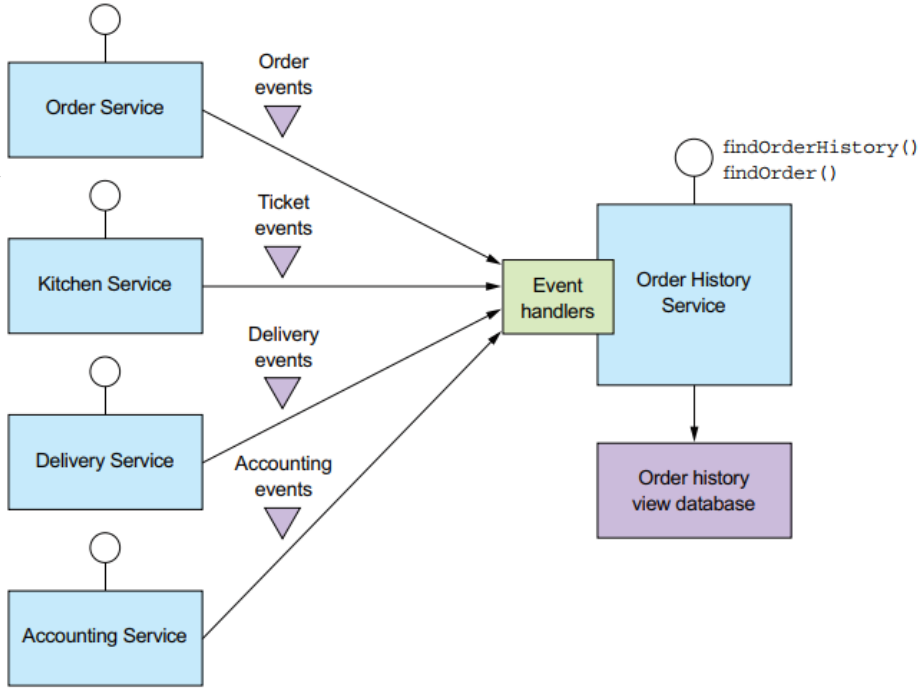


pmtu@fit.hcmus.edu.vn



Using the CQRS pattern

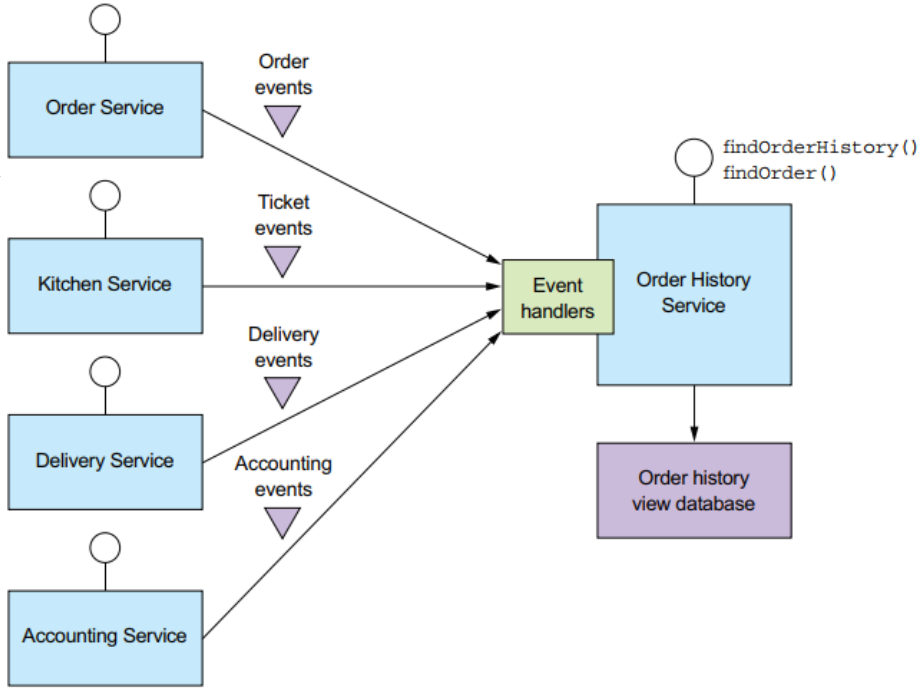
- Command query responsibility segregation
  - Overview
    - A service is **Order History Service**, which is a query service that implements the **findOrderHistory**() query operation. This service **subscribes** to events published by several services, including **Order Service, Delivery Service,…**
    - **Order History Service** has event handlers that **subscribe** to events published by several services and **update** the Order History View Database



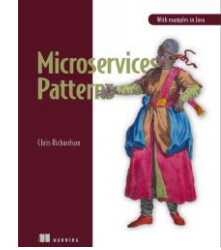
pmtu@fit.hcmus.edu.vn

Using the CQRS pattern

- Command query responsibility segregation
  - Overview
    - A service is **Order History Service**, which is a query service that implements the **findOrderHistory**() query operation. This service **subscribes** to events published by several services, including **Order Service, Delivery Service,…**
    - **Order History Service** has event handlers that **subscribe** to events published by several services and **update** the Order History View Database



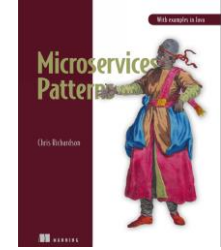
pmtu@fit.hcmus.edu.vn



Using the CQRS pattern

- Command query responsibility segregation
  - The benefits of CQRS
    - Enables the efficient implementation of queries in a microservice architecture
    - Enables the efficient implementation of diverse queries
    - Makes querying possible in an event sourcing-based application
    - Improves separation of concerns

pmtu@fit.hcmus.edu.vn



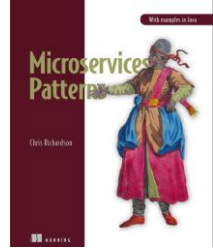
Using the CQRS pattern

- Command query responsibility segregation
  - The drawbacks of CQRS
    - More complex architecture
    - Dealing with the replication lag

pmtu@fit.hcmus.edu.vn

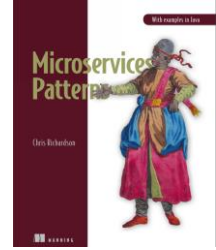
Outline

- Querying using the API composition pattern
- Using the CQRS pattern
- **Designing CQRS views**
- Implementing a CQRS view with AWS DynamoDB

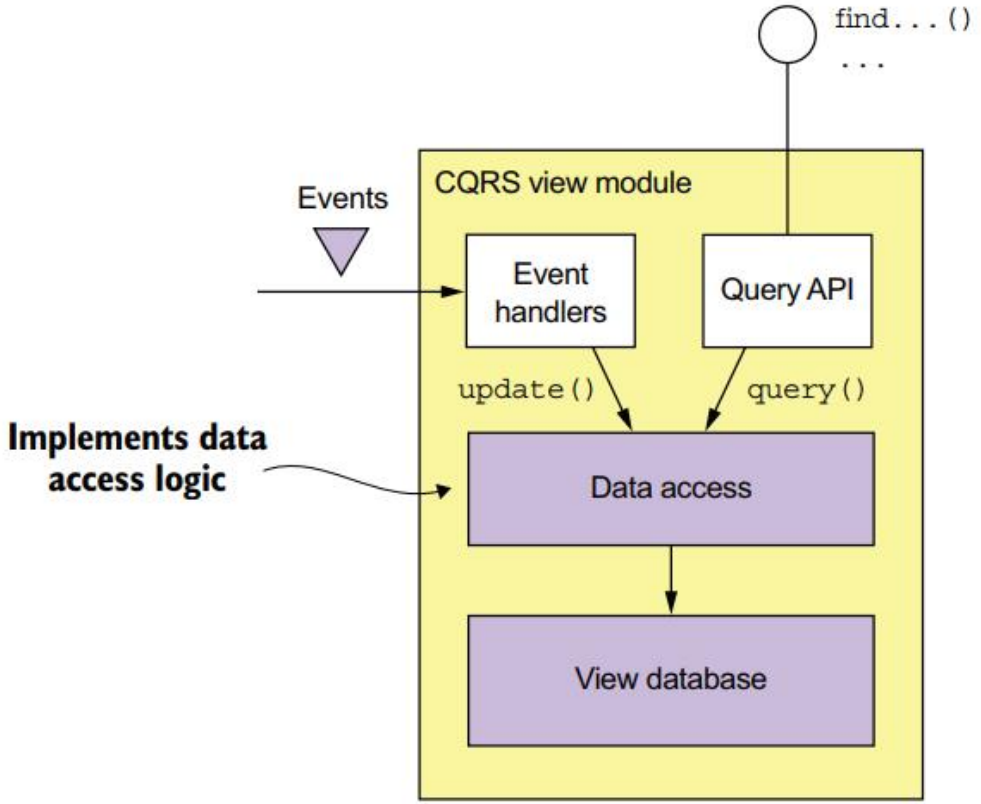


pmtu@fit.hcmus.edu.vn

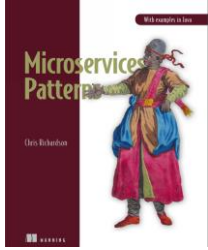
Designing CQRS views



- A CQRS view module has an API consisting of one more query operations. It implements these query operations by querying a database that it maintains by subscribing to events published by one or more services
- The data access module implements the database access logic
- The event handlers and query API modules use the data access module to update and query the database
- The event handlers module subscribes to events and updates the database



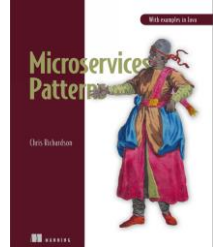
pmtu@fit.hcmu.edu.vn



Designing CQRS views

- **Some important design decisions when developing a view module**
  - To choose a database and design the schema.
  - When designing the data access module, must address various issues, including ensuring that updates are idempotent and handling concurrent updates.
  - When implementing a new view in an existing application or changing the schema of an existing application, must implement a mechanism to efficiently build or rebuild the view.
  - To decide how to enable a client of the view to cope with the replication lag

pmtu@fit.hcmus.edu.vn



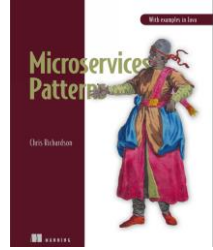
Designing CQRS views

Choosing a view datastore

- A key design decision is the choice of database and the design of the schema. The primary purpose of the database and the data model is to efficiently implement the view module's query operations.
- Besides efficiently implementing queries, the view data model must also efficiently implement the update operations executed by the event handlers.

If you need	Use	Example
PK-based lookup of JSON objects	A document store such as MongoDB or DynamoDB, or a key value store such as Redis	Implement order history by maintaining a MongoDB document containing the per-customer.
Query-based lookup of JSON objects	A document store such as MongoDB or DynamoDB	Implement customer view using MongoDB or DynamoDB.
Text queries	A text search engine such as Elasticsearch	Implement text search for orders by maintaining a per-order Elasticsearch document.
Graph queries	A graph database such as Neo4j	Implement fraud detection by maintaining a graph of customers, orders, and other data.
Traditional SQL reporting/BI	An RDBMS	Standard business reports and analytics.

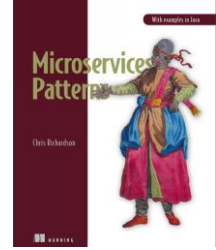
pmtu@fit.hcmus.edu.vn



Designing CQRS views

- Data access module design
  - Handling concurrency
    - The event handlers and the query API module don’t access the datastore directly. Instead they use the data access module, which consists of a data access object (DAO) and its helper classes.
    - It implements the update operations invoked by the event handlers and the query operations invoked by the query module.
    - The DAO maps between the data types used by the higher-level code and the database API.

pmtu@fit.hcmus.edu.vn

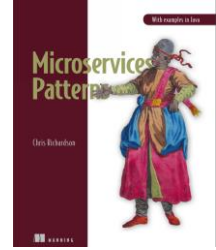


Designing CQRS views

- Data access module design
  - Handling concurrency
    - If a view subscribes to events published by a single aggregate type, there won't be any concurrency issues. That's because events published by a particular aggregate instance are processed sequentially
    - if a view subscribes to events published by multiple aggregate types, then it's possible that multiple events handlers update the same record simultaneously

An event handler for an Order* event might be invoked at the same time as an event handler for a Delivery* event for the same order. Both event handlers then simultaneously invoke the DAO to update the database record for that Order

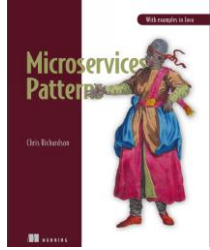
pmtu@fit.hcmus.edu.vn



Designing CQRS views

- Data access module design
  - Handling concurrency
    - A DAO must be written in a way that ensures that this situation is handled correctly. It must not allow one update to overwrite another.
    - If a DAO implements updates by reading a record and then writing the updated record, it must use either pessimistic or optimistic locking

pmtu@fit.hcmus.edu.vn



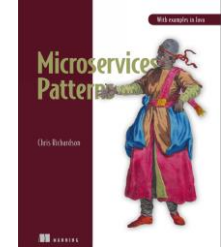
Designing CQRS views

- Data access module design
  - Enabling a client application to use an eventually consistent view
    - One issue with using CQRS is that a client that updates the command side and then immediately executes a query might not see its own update. The view is eventually consistent because of the unavoidable latency of the messaging infrastructure.
    - A command-side operation returns a token containing the ID of the published event to the client. The client then passes the token to a query operation, which returns an error if the view hasn't been updated by that event.

pmtu@fit.hcmus.edu.vn

Outline

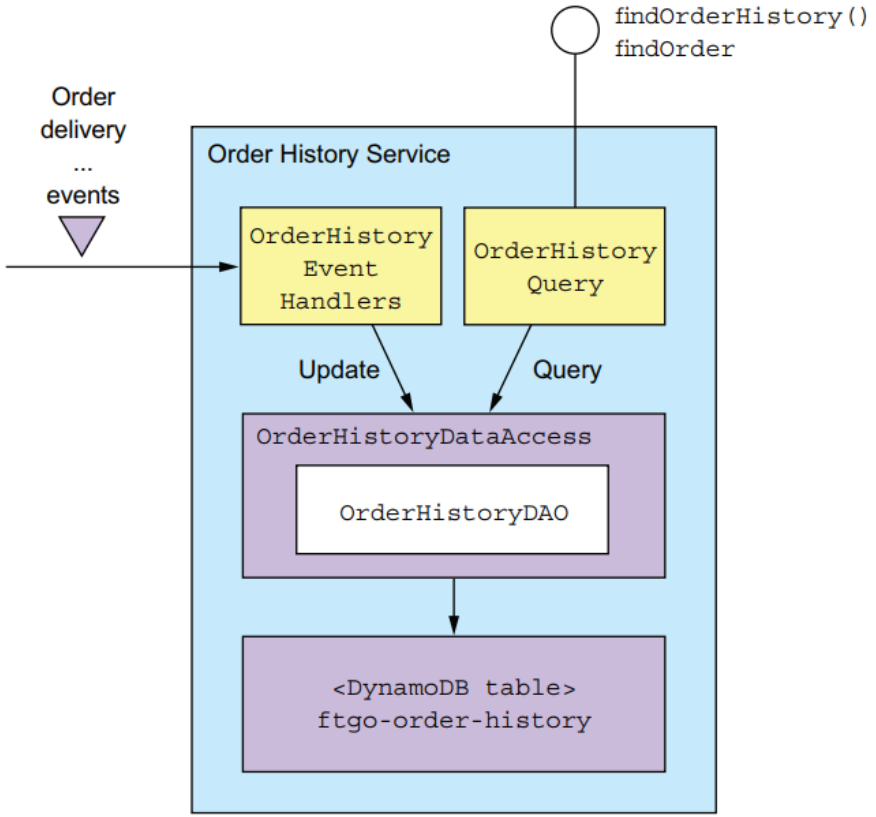
- Querying using the API composition pattern
- Using the CQRS pattern
- Designing CQRS views
- **Implementing a CQRS view with AWS DynamoDB**



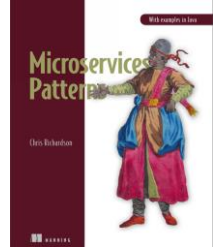
pmtu@fit.hcmus.edu.vn

Implementing a CQRS

- Example
  - OrderHistoryEventHandlers—Subscribes to events published by the various services and invokes the OrderHistoryDAO
  - OrderHistoryQuery APImodule—Implements the REST endpoints.
  - OrderHistoryDataAccess—Contains the OrderHistoryDAO, which defines the methods that update and query the ftgo-order-history DynamoDB table and its helper classes
  - ftgo-order-history DynamoDB table—The table that stores the orders



phuhv@fit.hcmus.edu.vn



Implementing a CQRS

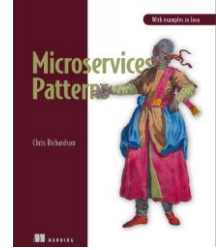
- The OrderHistoryEventHandlers module
  - The event handlers are simple methods. Each method is a one-liner that invokes an OrderHistoryDao method with arguments that are derived from the event

```
public class OrderHistoryEventHandlers {

  private OrderHistoryDao orderHistoryDao;

  public OrderHistoryEventHandlers(OrderHistoryDao orderHistoryDao) {
    this.orderHistoryDao = orderHistoryDao;
  }

  public void handleOrderCreated(DomainEventEnvelope<OrderCreated> dee) {
    orderHistoryDao.addOrder(makeOrder(dee.getAggregateId(), dee.getEvent()),
                             makeSourceEvent(dee));
  }

  private Order makeOrder(String orderId, OrderCreatedEvent event) {
    ...
  }

  public void handleDeliveryPickedUp(DomainEventEnvelope<DeliveryPickedUp>
                                                          dee) {
   orderHistoryDao.notePickedUp(dee.getEvent().getOrderId(),
           makeSourceEvent(dee));
  }

  ...
```

pmtu@fit.hcmus.edu.vn

Implementing a CQRS

- Data modeling and query design with DynamoDB
  - Designing the ftgo-order-history table
  - Defining an index for the findOrderHistory query
  - Implementing the findOrderHistory query
  - Paginating the query results
  - Updating orders
  - Detecting duplicate events



pntu@fit.hcmus.edu.vn

Implementing a CQRS

- Designing the ftgo-order-history table

ftgo-order-history table

Primary key

orderId	consumerId	orderCreationTime	status	lineItems	...
...	xyz-abc	22939283232	CREATED	[{...}. {...}, ...]	...
...	...	...	...	....	...

mus.edu.vn

Implementing a CQRS

- Designing the ftgo-order-history table

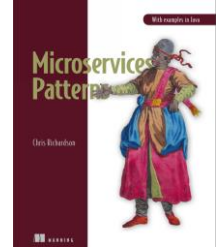
ftgo-order-history-by-consumer-id-and-creation-time global secondary index

Primary key

consumerId	orderCreationTime	orderId	status	...
xyz-abc	2293928323 2	cde-fgh	CREATED	...
...	...	...	...	...

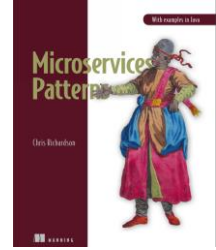
mus.edu.vn

Implementing a CQRS



- Implementing the findorderhistory query
  - A DynamoDB Query operation returns only those items that satisfy the filter expression. For example, to find Orders that are CANCELLED, the OrderHistoryDaoDynamoDb uses a query expression orderStatus = :orderStatus, where :orderStatus is a placeholder parameter.

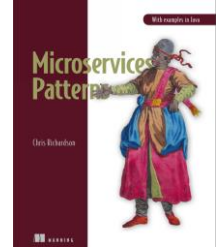
pmtu@fit.hcmus.edu.vn



Implementing a CQRS

- Paginating the query results
  - The DynamoDB Query operation has an operation pageSize parameter, which specifies the maximum number of items to return.
- Updating orders
  - DynamoDB supports two operations for adding and updating items: PutItem() and UpdateItem(). The PutItem() operation creates or replaces an entire item by its primary key.
  - With using PutItem() is ensuring that simultaneous updates to the same item are handled correctly.

pmtu@fit.hcmus.edu.vn

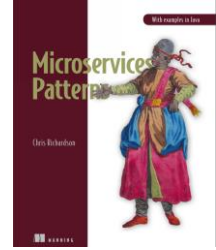


Implementing a CQRS

- Detecting duplicate events
  - The OrderHistoryDaoDynamoDb DAO can track events received from each aggregate instance using an attribute called «aggregateType»«aggregateId» whose value is the highest received event ID. An event is a duplicate if the attribute exists and its value is less than or equal to the event ID

```
attribute_not_exists(«aggregateType»«aggregateId»)
      OR «aggregateType»«aggregateId» < :eventId
```

pmt@fit.hcmus.edu.vn



Implementing a CQRS

- The OrderHistoryDaoDynamoDb class
  - The OrderHistoryDaoDynamoDb class implements methods that read and write items in the ftgo-order-history table. Its update methods are invoked by OrderHistoryEventHandlers, and its query methods are invoked by OrderHistoryQuery API

pmtu@fit.hcmus.edu.vn

Implementing a CQRS

- The OrderHistoryDaoDynamoDb class
  - The AddOrder() method
    - It's used to implement the conditional update

```
public class OrderHistoryDaoDynamoDb ...

@Override
public boolean addOrder(Order order, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
        .withPrimaryKey("orderId", order.getOrderId())
        .withUpdateExpression("SET orderStatus = :orderStatus, " +
                "creationDate = :cd, consumerId = :consumerId, lineItems =" +
                " :lineItems, keywords = :keywords, restaurantName = " +
                ":restaurantName")
        .withValueMap(new Maps()
                .add(":orderStatus", order.getStatus().toString())
                .add(":cd", order.getCreationDate().getMillis())
                .add(":consumerId", order.getConsumerId())
                .add(":lineItems", mapLineItems(order.getLineItems()))
                .add(":keywords", mapKeywords(order))
                .add(":restaurantName", order.getRestaurantName())
                .map())
        .withReturnValues(ReturnValue.NONE);
 return idempotentUpdate(spec, eventSource);
}
```

The primary key of the Order item to update

The update expression that updates the attributes

The values of the placeholders in the update expression

pmtu@fit.edu.vn

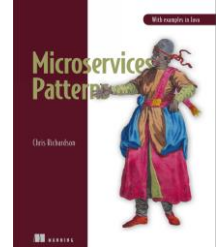
Implementing a CQRS

- The OrderHistoryDaoDynamoDb class
  - The NotePickedUp() method
    - It changes the deliveryStatus of the Order item to PICKED_UP

```
public class OrderHistoryDaoDynamoDb ...

@Override
public void notePickedUp(String orderId, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
         .withPrimaryKey("orderId", orderId)
         .withUpdateExpression("SET #deliveryStatus = :deliveryStatus")
         .withNameMap(Collections.singletonMap("#deliveryStatus",
                 DELIVERY_STATUS_FIELD))
         .withValueMap(Collections.singletonMap(":deliveryStatus",
                 DeliveryStatus.PICKED_UP.toString()))
         .withReturnValues(ReturnValue.NONE);
 idempotentUpdate(spec, eventSource);
}
```

pntu@fit.tmus.edu.vn

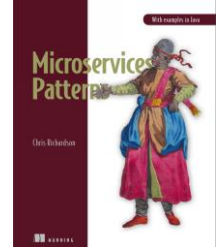
Implementing a CQRS



- The OrderHistoryDaoDynamoDb class
  - The IdempotentUpdate() method
    - It updates the item after possibly adding a condition expression to the UpdateItemSpec that guards against duplicate updates.

```
public class OrderHistoryDaoDynamoDb ...

private boolean idempotentUpdate(UpdateItemSpec spec, Optional<SourceEvent>
        eventSource) {
 try {
  table.updateItem(eventSource.map(es -> es.addDuplicateDetection(spec))
          .orElse(spec));
  return true;
 } catch (ConditionalCheckFailedException e) {
  // Do nothing
  return false;
 }
}
```

pmtu@fit.hcmus.edu.vn



Implementing a CQRS

- The OrderHistoryDaoDynamoDb class
  - the findorderhistory() method
    - It retrieves the consumer's orders by querying the ftgo-order-history table using the ftgo-order-history-by-consumerid-and-creation-time secondary index

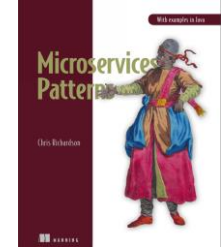
```
public class OrderHistoryDaoDynamoDb ...

@Override
public OrderHistory findOrderHistory(String consumerId, OrderHistoryFilter
        filter) {

QuerySpec spec = new QuerySpec()
        .withScanIndexForward(false)
         .withHashKey("consumerId", consumerId)
        .withRangeKeyCondition(new RangeKeyCondition("creationDate")
                                    .gt(filter.getSince().getMillis()));

filter.getStartKeyToken().ifPresent(token ->
      spec.withExclusiveStartKey(toStartingPrimaryKey(token)));

Map<String, Object> valuesMap = new HashMap<>();

String filterExpression = Expressions.and(
         keywordFilterExpression(valuesMap, filter.getKeywords()),
        statusFilterExpression(valuesMap, filter.getStatus()));

if (!valuesMap.isEmpty())
 spec.withValueMap(valuesMap);
```

Specifies that query must return the orders in order of increasing age

The maximum age of the orders to return

Construct a filter expression and placeholder value map from the OrderHistoryFilter.

pmtu@fit.edu.vn



Implementing a CQRS

- The OrderHistoryDaoDynamoDb class
  - the findorderhistory() method
    - It retrieves the consumer's orders by querying the ftgo-order-history table using the ftgo-order-history-by-consumerid-and-creation-time secondary index

```
if (!valuesMap.isEmpty())
 spec.withValueMap(valuesMap);

if (StringUtils.isNotBlank(filterExpression)) {
 spec.withFilterExpression(filterExpression);
}

filter.getPageSize().ifPresent(spec::withMaxResultSize);

ItemCollection<QueryOutcome> result = index.query(spec);

return new OrderHistory(
        StreamSupport.stream(result.spliterator(), false)
```

Construct a filter expression and placeholder value map from the OrderHistoryFilter.

Limit the number of results if the caller has specified a page size.

pmtu@fit.hcmus.edu.vn

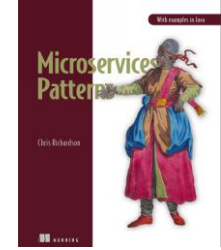
Implementing a CQRS

- The OrderHistoryDaoDynamoDb class
  - the findorderhistory() method
    - It retrieves the consumer's orders by querying the ftgo-order-history table using the ftgo-order-history-by-consumerid-and-creation-time secondary index

```
    .map(this::toOrder)
     .collect(toList()),
Optional.ofNullable(result
      .getLastLowLevelResult()
      .getQueryResult().getLastEvaluatedKey())
    .map(this::toStartKeyToken));
}
```

Create an Order from an item returned by the query.

pmtu@fit.hcmus.edu.vn



Summary

- Implementing queries that retrieve data from multiple services is challenging because each service's data is private.
- There are two ways to implement these kinds of query: the API composition pattern and the Command query responsibility segregation (CQRS) pattern.
- The API composition pattern, which gathers data from multiple services, is the simplest way to implement queries and should be used whenever possible.
- A limitation of the API composition pattern is that some complex queries require inefficient in-memory joins of large datasets.
- The CQRS pattern, which implements queries using view databases, is more powerful but more complex to implement.
- A CQRS view module must handle concurrent updates as well as detect and discard duplicate events.
- CQRS improves separation of concerns by enabling a service to implement a query that returns data owned by a different service.
- Clients must handle the eventual consistency of CQRS views.

pmtu@fit.hcmus.edu.vn