Shiju Varghese
2.4K Followers · About Follow

Building Microservices with Event Sourcing/CQRS in Go using gRPC, NATS Streaming and CockroachDB

Shiju Varghese Aug 14, 2018 · 11 min read

In this post, I will demonstrate a simple Event Sourcing/CQRS Example in Go to demonstrate how to solve the practical challenges of Microservices based distributed systems. Keep in mind that my objective of this post is not about introducing best practices for Event Sourcing and CQRS, but introduce these two architectural patterns by writing a simple example in Go, which provides a solution for the challenges in Microservices based distributed systems in order to dealing with transactions and data.

The Hardest Part of Microservices: Data and Transactions

People who have never worked on distributed systems, have been misinterpreting that Microservices are just about running services in Docker containers and orchestrating it with Kubernetes. In my country, India, a lot of people have been providing corporate training and guidance on Microservices with two misinterpretations: Microservices are just about orchestrating containers with Kubernetes, or just about using Spring Boot framework with Netflix OSS. But in practice, Microservices, is a distributed systems architecture pattern that is all about functional decomposition for building highly scalable and evolvable software systems. In a nutshell, microservices are small, autonomous services that work together.

Data is scattered in several databases owned by Microservices

A common practice to decomposes Microservices is to design each Microservice against a *bounded context,* which is a central pattern in Domain-Driven Design (DDD), which provides logical separation of business problem into various sub domains by dividing a large domain model into different *bounded contexts.* A single *bounded context* will have one or many *aggregate*, and you make transactions against aggregates. Because we typically build Microservices against each *bounded contexts,* we typically use individual databases for each Microservices where each Microservice looks like an independent system.

Because we broke up a monolithic system into several autonomous services, the data is scattered amongst several databases owned by individual Microservices. This makes lot of complexity to your applications and architecture. For example, a business transaction may span into multiple Microservices. Let's say you build an e-commerce system with microservices where placing an order would be initially handled by *OrderService* — a Microservice, then payment processing might be done by another service — a *PaymentService*, and so on. And another challenge is querying data from multiple databases. With a monolithic database, you can easily perform join queries from a single database. Because the monolithic database is moved into several databases as part of the decomposition of functional components, you can't simply execute join queries, thus you must get data from multiple databases. Here you don't have any centralized database.

Building Event-Driven Microservices with Event Sourcing and CQRS

In order to solve the practical challenges of Microservices, an event-driven reactive system on DDD aggregates would be a great approach. For this, I highly recommend to use Event Sourcing, which is an event-centric architecture to construct the state of an application by composing various events.

Event Sourcing: An event store of immutable log of events

Event Sourcing deals with an event store of immutable log of events, in which each log (a state change made to an object) represents an application state. An event store is like a version control system. In a microservices architecture, we can persist aggregates as a sequence of event. Events are facts, which represent some actions happened in the system. These are immutable, which can't be changed or be retracted. The example of

events in an e-Commerce system are *OrderCreated, PaymentDebited, OrderApproved, OrderRejected, OrderShipped, OrderDelivered, etc.*

In your Event Sourcing architecture, when you publish one event from one Microservice, other Microservices can be reactive to those events and publish another set of events. Sometimes, the sequence of events can be compared with Unix pipes. A single transaction in a Microservices based system may span into multiple Microservices where we can perform a transaction as a sequence of events by building reactive Microservices. Each state change of an aggregate can be treated as an event, which is an immutable fact about your system.

CQRS for building query model for the views of aggregates

When you make persistence as a sequence of events by using Event Sourcing, you may need an architecture approach to make queries for your Microservices because the write model (commands) are just an event store. An architecture pattern, Command Query Responsibility Segregation (CQRS) is an ideal pattern for implementing queries for your Microservices. As the name implies, CQRS segregates an application into two parts: *Commands* to perform an action to change the state of aggregates, and *Queries* to provide a query model for the views of aggregates. Although CQRS is an independent architectural pattern, we often used this with Event Sourcing as the model for commands. We may also use different databases for both write operations and query operations. This will also allows you to make highly performant query model by loading denormalised data sets into data stores of read models.

An Event Sourcing/CQRS Example in Go with gRPC, NATS Streaming and CockroachDB

The example demo is available here: https://github.com/shijuvar/go-distributed-sys

I've simplified the example for the sake of a conceptual demo in order to simply understand things, and get some insight on how to use technologies like gRPC, NATS, etc. for building distributed systems.

NATS Streaming for messaging

In Event Sourcing architecture, when you publish one event from one Microservice, other Microservices can be reactive to those events, and publish another set of events after doing its own local transactions. A single transaction in a Microservices based

system may span into multiple Microservices where we can perform a transaction as a sequence of events by building reactive Microservices. Each state change of an aggregate can be treated as an event, which is an immutable fact about your system. In order to publish events to let other Microservices know that something has happened in your system, we need to use a messaging system. In this example, we use NATS Steraming Server as the event streaming system to build event-driven Microservices. An event-driven, reactive architecture is a great choice of architecture approach for building massively scalable Microservices. If you're not familiar on NATS and NATS Streaming, check out my articles on basic NATS here and NATS Streaming here. I consider NATS as a nervous system for building distributed systems, which I've been working on Go ecosystem.

gRPC for building APIs

In this example, the Event Store provides an API to execute commands, which is an gRPC based API. gRPC is a high performance, open-source remote procedure call (RPC) framework that can run anywhere. It enables client and server applications to communicate transparently, and makes it easier to build connected systems. gRPC is widely known as a communication protocol in Microservices. If you make inter-process communication between Microservices over an API, gRPC is a better choice. If you're new gRPC, check out my article here.

Example demo

Here's the directory structure in the example:

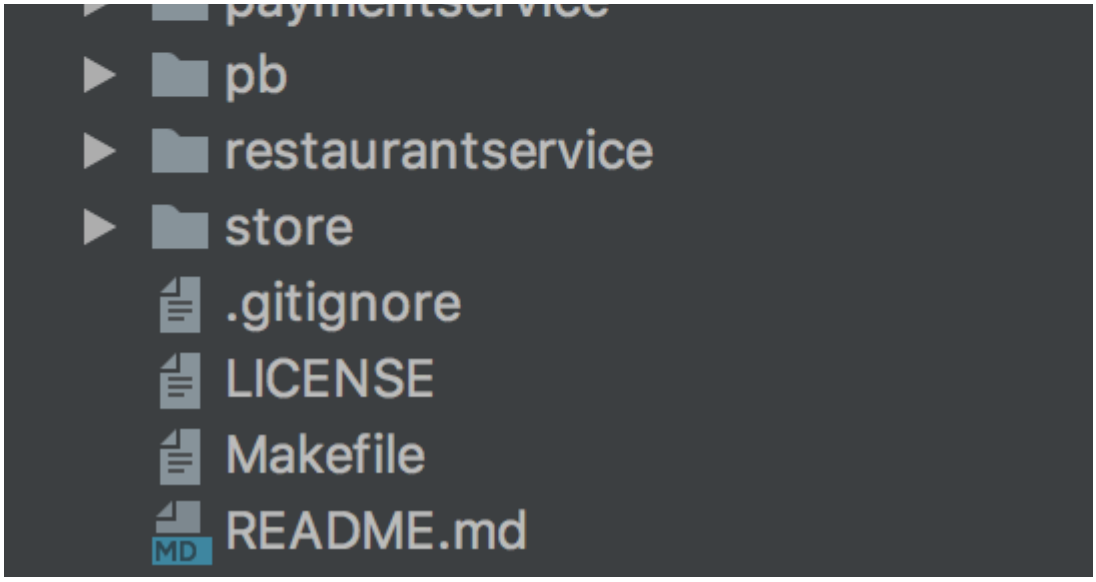

Figure 1. Directory structure of the example application

Here's the high level diagram of example demo:

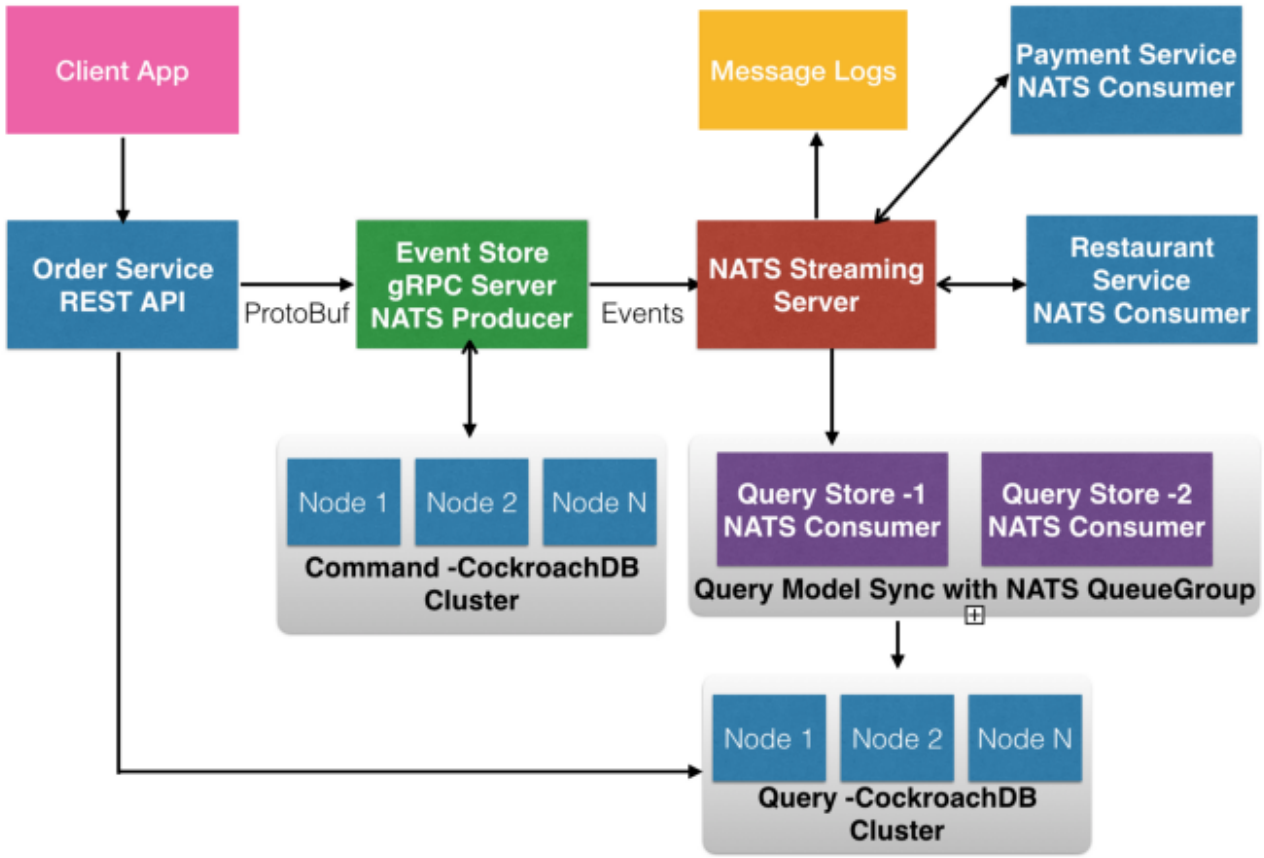

Figure 2. A high level diagram of the example application

Here's the basic workflow in the example:

1. A client app post an Order to an HTTP API.
2. An HTTP API (**orderservice**) receives the order, then executes a command onto Event Store, which is an immutable log of events, to create an event via its gRPC API (**eventstore**).
3. The Event Store API executes the command and then publishes an event "order-created" to NATS Streaming server to let other services know that an event is created.
4. The Payment service (**paymentservice**) subscribes the event "order-created", then make the payment, and then create an another event "order-payment-debited" via Event Store API.
5. The Query syncing workers (**orderquery-store1** and **orderquery-store2** as queue subscribers) are also subscribing the event "order-created" that synchronise the data models to provide state of the aggregates for query views.
6. The Event Store API executes a command onto Event Store to create an event "order-payment-debited" and publishes an event to NATS Streaming server to let other services know that the payment has been debited.
7. The restaurant service (**restaurantservice**) finally approves the order.
8. A Saga coordinator (Distributed Saga) manages the distributed transactions and makes void transactions on failures (to be implemented)

Event Store for Event Sourcing

Here's the structure of the message *Event*. Every state change in the example is treated as an event and execute a command into Event Store.

Listing 1. Structure of the message Event in protocol buffers

```
message Event {
    string event_id = 1;
    string event_type = 2;
    string aggregate_id = 3;
    string aggregate_type = 4;
    string event_data = 5;
```

```
    string channel = 6; // an optional field
}
```

The Event Store provides an gRPC API to persist events. Here's the basic code block in the Event Store implementation:

Listing 2. Basic implementation in gRPC server

```
// CreateEvent RPC creates a new Event into EventStore
// and publish an event to NATS Streaming
func (s *server) CreateEvent(ctx context.Context, in *pb.Event)
(*pb.Response, error) {
   // Persist data into EventStore database
   command := store.EventStore{}
   // Persist events as immutable logs into CockroachDB
   err := command.CreateEvent(in)
   if err != nil {
      return nil, err
   }
   // Publish event on NATS Streaming Server
   go publishEvent(s.StreamingComponent, in, )
   return &pb.Response{IsSuccess: true}, nil
}

// publishEvent publishes an event via NATS Streaming server
func publishEvent(component *natsutil.StreamingComponent, event
*pb.Event) {
   sc := component.NATS()
   channel := event.Channel
   eventMsg := []byte(event.EventData)
   // Publish message on subject (channel)
   sc.Publish(channel, eventMsg)
   log.Println("Published message on channel: " + channel)
}
```

Whenever a new Event is persisted into Event Store as an immutable log via its gRPC API, it publishes an event via NATS Streaming server to let other Microservices know that a new event is published so all subscriber Microservices can be reactive to those events. In this example, the event is published into the messaging system (NATS Streaming) from Event Store API itself. In real-world scenarios, it might be from individual Microservices and sometimes it might be from a saga coordinator (Distributed Saga) that coordinates a single business transaction, which spans into multiple Microservices.

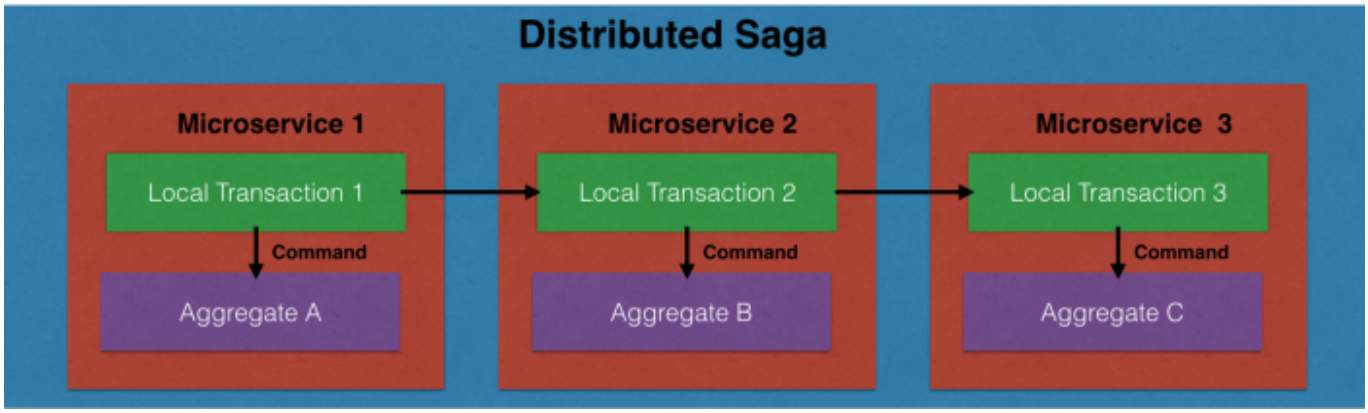

Figure 3. Distributed saga coordinates transactions in a business transaction

Subscribe events for building reactive Microservices

When you persist an event into Event Store, a new event is published over NATS Streaming, and if your Microservices are interested to events to be reactive on those events in order to doing its own actions, these Microservices can subscribe those events. Here we subscribe events by using NATS Streaming. In this example, when an event "order-created" is created the *PaymentService* subscribes the event, and execute another event called "order-payment-debited".

Listing 3. A NATS Streaming subscriber client that's reactive to an event "order-created

```
const (
   clusterID = "test-cluster"
   clientID  = "payment-service"
   subscribeChannel   = "order-created"
   durableID = "payment-service-durable"

   event     = "order-payment-debited"
   aggregate = "order"

   grpcUri   = "localhost:50051"
)

func main() {
   // Register new NATS component within the system.
   comp := natsutil.NewStreamingComponent(clientID)

   // Connect to NATS Streaming server
   err := comp.ConnectToNATSStreaming(
      clusterID,
      stan.NatsURL(stan.DefaultNatsURL),
   )
   if err != nil {
```

```
            log.Fatal(err)
        }
        // Get the NATS Streaming Connection
        sc := comp.NATS()
        // Subscribe with manual ack mode, and set AckWait to 60 seconds
        aw, _ := time.ParseDuration("60s")
        // Subscribe the channel
        sc.Subscribe(subscribeChannel, func(msg *stan.Msg) {
            msg.Ack() // Manual ACK
            order := pb.OrderCreateCommand{}
            // Unmarshal JSON that represents the Order data
            err := json.Unmarshal(msg.Data, &order)
            if err != nil {
                log.Print(err)
                return
            }
            // Create OrderPaymentDebitedCommand from Order
            command := pb.OrderPaymentDebitedCommand {
                OrderId: order.OrderId,
                CustomerId: order.CustomerId,
                Amount: order.Amount,
            }
            log.Println("Payment has been debited from customer account for
    Order:", order.OrderId)
            if err:= createPaymentDebitedCommand(command); err!=nil {
                log.Println("error occured while executing the
    PaymentDebited command")
            }
        }, stan.DurableName(durableID),
            stan.MaxInflight(25),
            stan.SetManualAckMode(),
            stan.AckWait(aw),
        )
        runtime.Goexit()
    }
```

Subscribe events for building query model for the views of aggregates

In CQRS, which segregates an application into two parts: Commands and Queries. Here commands are implemented by using Event Sourcing, which uses an Event Store of immutable log of events to construct the application state. In order to create query models, we can also subscribe events when command operations are executed. Here, in this example, services *orderquery-store*1 and *orderquery-store2* subscribe the event "order-created" via NATS Streaming by forming a *QueueGroup* Subscriber. A *QueueGroup* Subscriber in NATS lets you implement load balanced workers without having any configuration.

Listing 4. A NATS Streaming QueueGroup subscriber client for syncing query model

```
const (
   clusterID  = "test-cluster"
   clientID   = "order-query-store1"
   channel    = "order-created"
   durableID  = "store-durable"
   queueGroup = "order-query-store-group"
)

func main() {
   // Register new component within the system.
   comp := natsutil.NewStreamingComponent(clientID)

   // Connect to NATS Streaming server
   err := comp.ConnectToNATSStreaming(
      clusterID,
      stan.NatsURL(stan.DefaultNatsURL),
   )
   if err != nil {
      log.Fatal(err)
   }
   // Get the NATS Streaming Connection
   sc := comp.NATS()
   sc.QueueSubscribe(channel, queueGroup, func(msg *stan.Msg) {
      order := pb.OrderCreateCommand{}
      err := json.Unmarshal(msg.Data, &order)
      if err == nil {
         // Handle the message
         log.Printf("Subscribed message from clientID - %s: %+v\n",
clientID, order)
         queryStore := store.QueryStore{}
         // Perform data replication for query model into CockroachDB
         err := queryStore.SyncOrderQueryModel(order)
         if err != nil {
            log.Printf("Error while replicating the query model %+v",
err)
         }
      }
   }, stan.DurableName(durableID),
   )
   runtime.Goexit()
}
```

Here whenever an event “order-created” is published, our workers are subscribing the event, then executes some logic to sync the data model with the events stored in Event Store and creates a denormalised views of aggregates by persisting data into the data store to be used for *Query* model in CQRS architecture.

Persistent store with CockroachDB

One major benefit of using CQRS is that you can have different data models for both write operations and query operations, thus you can also use different database technologies. In this example demo, we use CockroachDB for executing both commands and query models. CockroachDB is a distributed SQL database for building global, scalable cloud services that survive disasters. CockroachDB. In your Go applications, you can work with package *database/sql* using a PostgreSQL-compatible driver like *github.com/lib/pq*. If you're making transactions on package *database/sql* use it with CockroachDB's Go package *github.com/cockroachdb/cockroach-go/crdb*. In the example demo, persistence logic with CockroachDB is implemented in the package ***store***. The method *ExecuteTx* of package *crdb* helps you execute transactions into CockroachDB.

Listing 5. Transaction implementation in CockroachDB using package *crdb*

```
func (store QueryStore) SyncOrderQueryModel(order
pb.OrderCreateCommand) error {

   // Run a transaction to sync the query model.
   err := crdb.ExecuteTx(context.Background(), db, nil, func(tx
*sql.Tx) error {
      return createOrderQueryModel(tx, order)
   })
   if err != nil {
      return errors.Wrap(err, "Error on syncing query store")
   }
   return nil
}

func createOrderQueryModel(tx *sql.Tx, order pb.OrderCreateCommand)
error {
   // Execute transactions here
return nil

}
```

Here's immutable logs of Event Store for creating an order that was finally approved from a service *restaurantservice*.

id	eventtype	aggregateid	aggregatetype
087a83cc-171b-49cc-bf22-96fd2c930476	order-created	010c16c3-cdfa-4d4f-a73e-f15f0d341a62	order
18f3cdfc-6f74-425a-9a7a-3f51f108b6aa	order-payment-debited	010c16c3-cdfa-4d4f-a73e-f15f0d341a62	order
562778b9-2c95-4c8d-9782-563de6467fb9	order-approved	010c16c3-cdfa-4d4f-a73e-f15f0d341a62	order

Figure 4. Immutable logs of events in Event Store

The events table of Event Store has a field named *eventdata* to which we persist the entire event data as a JSON document so that it will be useful for building application state as well as building query views.

Source Code

The source code of the example demo is available here:
https://github.com/shijuvar/go-distributed-sys

Summary

The primary objective of the post is to give some insight on building event-driven Microservices by using Event Sourcing and CQRS. It's very complex to build real-world Microservices where the hardest part is deal with data that is scattered amongst several databases owned by individual Microservices. This makes complexities to build business transactions that spans into multiple microservices, and querying data where making a join query is not possible with multiple databases. Thus, you must use some architectural approaches to build Microservices from a real-world perspective where you can consider to use an event-driven architecture like Event Sourcing. The Event Sourcing architecture is better to be paired with CQRS. In this simple example demo, I've used three of my favourite technologies: gRPC, NATS and CockroachDB.

You can follow me on twitter at @shijucv. I do provide *training and consulting on Go programming language (Golang)* and distributed systems architectures, in India.

Microservices Event Sourcing Golang Grpc Nats Streaming