Event-Driven Microservices with NATS Streaming
Shiju Varghese
Consulting Solutions Architect

https://medium.com/@shiuvar
About Me

• Consulting Solutions Architect
• Consulting and Training on Golang and Cloud-Native Distributed Systems
• Early adopter of Go programming language
• Published Author
• Honoured with Microsoft MVP award seven times
• Blog: medium.com/@shijuvar | Twitter: @shijucv
Author
Agenda

• Overview about the practical Challenges of Microservices Architecture
• Introduction to Event-Driven Architectures
• Introduction to NATS Streaming Server
• Using NATS Streaming Server to build Event-Driven Microservices
MICROSERVICES ARCHITECTURE AND ITS PRACTICAL CHALLENGES
Monolithic Architecture

- eCom Store
- Catalog
- Customer
- Orders
- Payment

Browser Client

Mobile Client

Monolithic Database
Microservices Architecture

Browser Client -> REST -> API Gateway -> Product Catalog Service -> Product Catalog Database

Mobile Client -> REST -> API Gateway -> Customer Service -> Customer Database

API Gateway -> Order Service -> Order Database

API Gateway -> Payment Service -> Payment Database
Autonomous Services Around Bounded Context

- Software broken up into functional components.
- Componentization via Services in which each service is packaged as one unit of execution.
- Independent, autonomous process with no dependency on other Microservices.
- Services are organized around business capability.
- Decentralization of Data Management.
- Independently replaceable and upgradeable.
Practical Challenges

- Database per service, Isolated persistence
- A transaction may span into multiple microservices
- Distributed transactions with Two-Phase Commit (2PC) is not viable option
- How to manage failures while performing distributed transactions
- How to manage data consistency
- How to querying data from multiple data store
The greatest challenge of any Microservices based implementation is how to manage its data and transactions.
EVENT-DRIVEN ARCHITECTURES
Event-Driven Architecture

- **Order Service**
  - Publish Events: Event: OrderCreated
  - Subscribe Events: Event: OrderApproved

- **Message System**
  - Publish Events: Event: PaymentDebited

- **Order DB**
  - Event: OrderCreated

- **Restaurant Service**
  - Publish Events: Event: OrderApproved

- **Payment Service**
  - Subscribe Events: Event: PaymentDebited
Events

• A fact represents something that has happened in the system (Eg: OrderCreated, PaymentDebited, OrderApproved)
• Events are immutable
Event-Driven Architectures

- Application Event
- Transaction Log Tailing
- Event Sourcing
Event Sourcing

• An event-centric pattern for implementing business logic and persisting aggregates
• Events represent state changes of aggregates; Publishes domain events on mutations of aggregates
• Event log for the aggregates
• Preserves the history of aggregates
### Event Store of Immutable Log of Events

<table>
<thead>
<tr>
<th>Aggregate ID</th>
<th>Aggregate Type</th>
<th>Event ID</th>
<th>Event Type</th>
<th>Event Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>Order</td>
<td>1001</td>
<td>OrderCreated</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>Order</td>
<td>1002</td>
<td>OrderApproved</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>Order</td>
<td>1003</td>
<td>OrderShipped</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>Order</td>
<td>1004</td>
<td>OrderDelivered</td>
<td></td>
</tr>
</tbody>
</table>
Command Query Responsibility Segregation (CQRS)

Image Courtesy: http://martinfowler.com

NATS is now a hosted CNCF Project
NATS

- A Cloud Native Computing Foundation (CNCF) hosted incubation project
- NATS is an open-source, high performant cloud-native messaging system
- NATS Streaming is an extremely performant, lightweight reliable streaming platform built on the top of core NATS platform that provides persistent logs.
- Both NATS and NSTS Streaming Server are written in Go
- NATS was created by Derek Collison, and supported by his team at Synadia
- NATS is deployed in some of the largest cloud platforms, including: VMware, CloudFoundry, Baidu, Siemens, and GE
Publish - Subscribe

• Basic messaging pattern
• A producer publishes a message with a specific subject/channel
• All active consumers with subscriptions matching the subject/channel receive it
NATS

NATS Server

Synadia Supported Clients
C / C# / Elixir / Go / Java / NGINX / Node.js / Pure Ruby / Python Asyncio / Python Tornado / Ruby

Community Clients
.NET / Arduino / Clojure / Elixir / Elm / Erlang / Haskell / Lua / MicroPython / PHP / Perl / Python / Python Twisted / Qt5 C++ / Rust / Scala / Spring API / Swift
NATS STREAMING SERVER

A Data Streaming Platform Powered by NATS for Microservices and Distributed Systems
NATS Streaming Server

Application Code

NATS Streaming Client API

NATS Client

NATS Streaming Server

NATS Server

NATS Streaming Server Module

Storage
- Message/event persistence
- At-least-once-delivery
- Publisher rate limiting
- Rate matching/limiting per subscriber
- Historical message replay by subject
- Durable subscriptions
// publishEvent publishes an event via NATS Streaming server
func publishEvent(event *pb.Event) {
    // Connect to NATS Streaming server
    sc, err := stan.Connect(
        clusterID,
        clientID,
        stan.NatsURL(stan.DefaultNatsURL),
    )
    if err != nil {
        log.Println(err)
        return
    }
    defer sc.Close()
    channel := event.Channel
    eventMsg := byte(event.EventData)
    // Publish message on subject (channel)
    sc.Publish(channel, eventMsg)
    log.Println("Published message on channel: " + channel)
Subscribe Events

```go
// Subscribe with manual ack mode, and set AckWait to 60 seconds
aw, _ := time.ParseDuration(s: "60s")
sc.Subscribe(channel, func(msg *stan.Msg) {
    msg.Ack() // Manual ACK
    order := pb.Order{}
    // Unmarshal JSON that represents the Order data
    err := json.Unmarshal(msg.Data, &order)
    if err != nil {
        log.Printf(err)
        return
    }
    // Handle the message
    log.Printf("Subscribed message from clientID - %s for Order: %+v\n", clientID, order)
}, stan.DurableName(durableID),
    stan.MaxInflight(m: 25),
    stan.SetManualAckMode(),
    stan.AckWait(aw),
)```

Subscribe with QueueGroup

```go
const {
    clusterID = "test-cluster"
    clientID = "order-query-store1"
    channel = "order-notification"
    durableID = "store-durable"
    queueGroup = "order-query-store-group"
}

sc.QueueSubscribe(channel, queueGroup, func(msg *stan.Msg) {
    order := pb.Order{}
    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),
)}
```
1: Request for place an Order from client app to OrderService
2: OrderService calls to a gRPC API provided by EventStore to create events
3: Event Store persist event data into Event Store and publishes events via NATS streaming server
4: Microservices subscribe events published by Event Store
5: By subscribing events, application state is constructed by composing event data, query model creates read model for views.
6: Microservices reactive to events by subscribing events, and do its own actions and publish other set of events

https://github.com/shijuvar/gokit/tree/master/examples/nats-streaming
Thank you
GREAT INDIAN DEVELOPER SUMMIT 2019
Conference: April 23-26, Bangalore

Register early and get the best discounts!

www.developersummit.com  @greatindiandev  bit.ly/gidslinkedin