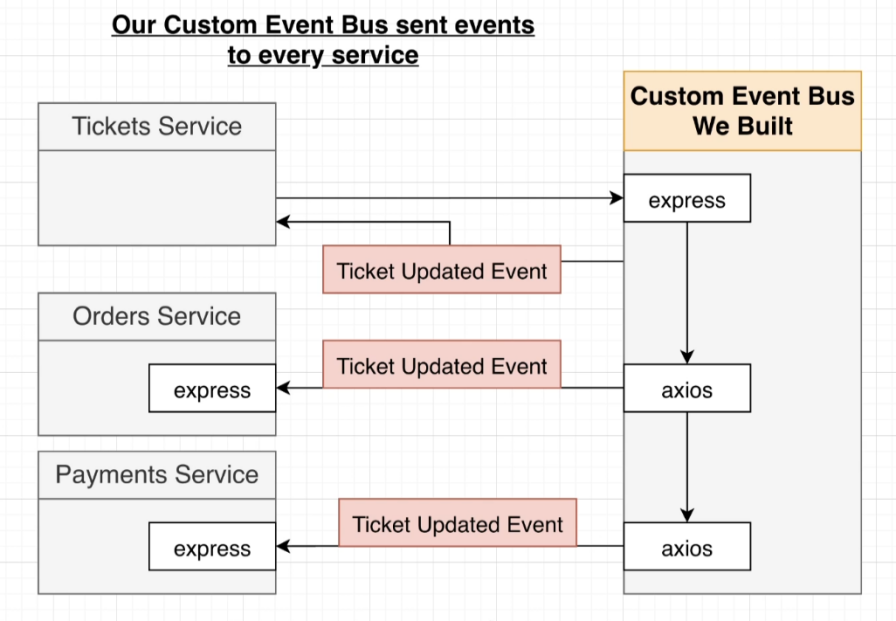
**NATS Streaming Server** (docs.nats.io)

It is basically an event bus, which will help to transmit data among listeners and publishers.

docker image -> nats-streaming

Without NATS Streaming Server we need to declare, define, publish, listen manually with API CALLS.

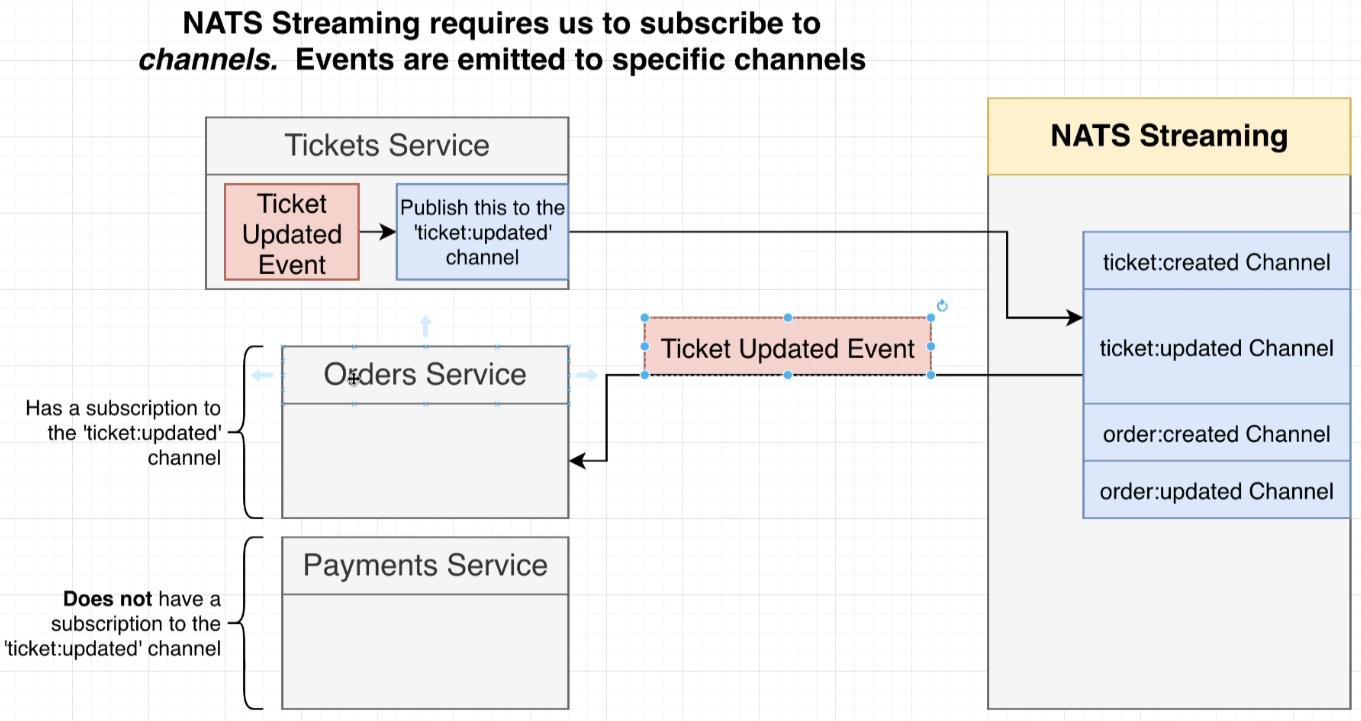
Eg



With NATS Streaming Server we can Create Events, Publish Events, Listen Events without any Manual API CALLS, it will take care of it automatically.

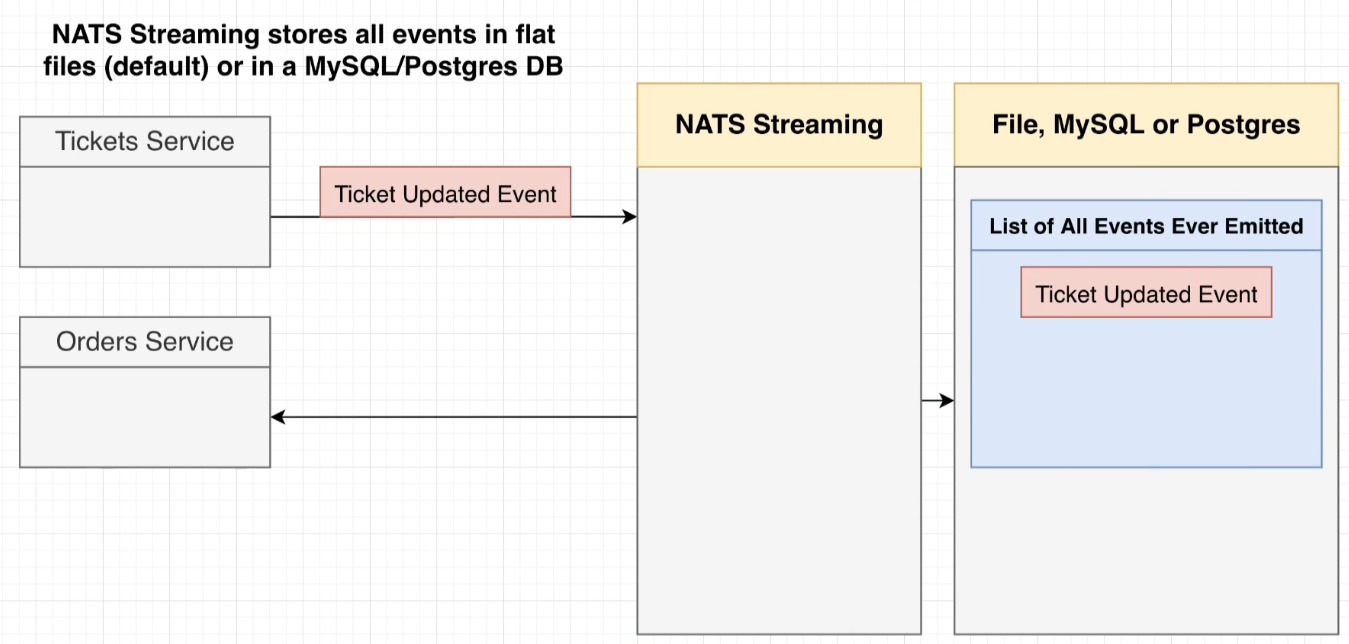
Example 1

(Ticket Update Event Published from ticket Service and received by Order Service with active listner of ticket:updated)



Example 2

Storing Event in Custom Events Store So that no Events Integrity get Exploit or can say not event get miss even if any service went down for a while



Steps to Create and NATS Streaming Server or (STAN)

1. Create an deployment for nats streaming server in your kubernetes cluster.

use official docker image = nats-streaming

(Command Explanation => https://docs.nats.io/legacy/stan/changes/configuring/cmdline )

**apiVersion:** apps/v1

**kind:** Deployment

**metadata**:

**name:** nats-depl

**spec:**

**replicas:** 1

**selector:**

**matchLabels:**

**app:** nats

**template:**

**metadata:**

**labels:**

**app:** nats

**spec:**

**containers:**

**- name:** nats

**image:** nats-streaming:0.17.0

**args:**

[

"-p",

"4222",

"-m",

"8222",

"-hbi",

"5s",

"-hbt",

"5s",

"-hbf",

"2",

"-SD",

"-cid",

"ticketing",

]

---

**apiVersion:** v1

**kind:** Service

**metadata:**

**name:** nats-srv

**spec:**

**selector:**

**app:** nats

**ports:**

- **name:** client

**port:** 4222

**targetPort:** 4222

**protocol:** TCP

- **name:** monitoring

**port:** 8222

**targetPort:** 8222

**protocol:** TCP

-p : Listening Port **,** -cid : Cluster ID (default = test-cluster)

-m: Monitoring Port **,** -SD : Stan Debug => Enable STAN debugging output

-hbi: ( Heart Beat Interval) => Interval at which server sends heartbeat to a client

-hbt: (Heart Beat Timout) => How long server waits for a heartbeat response

-hbf: (Hear Beat FailCount) => Number of failed heartbeats before server closes the client connection.

1. Create a folder named nats-test

A STAN (client) is needed for creating connection, for listening events, for publishing events, to the NATS Streaming Server.

ClusterID -> it the string value given to client while connecting to cluster of ClusterID

ClientID -> it is the random unique id given to the Client for data seperation. eg Give Pod name generated by kubernetes.

URL -> it is the main point where NATS Streaming server is going to connect.

example 1 (IF you are trying to access localhost)

URL= <http://localhost:4222> (Remember to portForward 4222 of localhost to 4222 of NATS Streaming Server pod running inside the Cluster)

example 2 (IF you try to access inside Kubernetes Cluster )

URL = <http://nats-srv:4222> (Remember to enable ClusterIP service of NATS Streaming Server POD which will use port 4222)

1. in src/ create listener and publisher files for testing

**listner.ts file** will be responsible for listening the event passed over nats streaming server

**import** { randomBytes } from "crypto";

**import** nats from "node-nats-streaming";

**import** { CreateTicketEventListener } from "./events/ListenCreateTicket";

**import** { UpdateTicketEventListener } from "./events/ListenUpdateTicket";

**console.**clear();

**const** stan = nats.connect("ticketing", randomBytes(4).toString("hex"), {

url: "http://localhost:4222",

});

**stan.**on("connect", () => {

**console.**log("Listener Connected to NATS");

**stan.**on("close", () => {

**console.**log("Nats Connection Closed!");

**process.**exit();

});

**const** li = new CreateTicketEventListener(stan);

**li.**listen();

**const** l2 = new UpdateTicketEventListener(stan);

**l2.**listen();

});

**process.**on("SIGINT", () => {

**stan.**close();

});

**process.**on("SIGTERM", () => {

**stan.**close();

});

**Explanation:**

1. **nats.connect(ClusterID,ClientID,{url})**
2. **stan** will hold the **client** of Connection with **ClusterID:**ticketing**, ClientID:**randomBytes(4)**, url:**http://localhost:4222
3. localhost at 4222 is not listening to NATS Straming Server POD right now
4. so make it listen ->kubectl port-forward **podName** 4222:4222
5. Remember fetch out PodName of NATS Streaming Server via -> kubectl get **pods**
6. CreateTicketEventListener,UpdateTicketEventListener are the custom class located in nats-test/src/events
7. There classes require stan(client) as an contructor and will listen on calling to object with listen();

**Example -> CreateTicketListener.ts**

**import** { Message } from "node-nats-streaming";

**import** { Listener, ListenerData } from "./base-listener";

**import** { Subjects } from "./subjects";

**import** { TicketCreatedEvent } from "./ticket-created-event";

**export class** CreateTicketEventListener **extends** Listener**<TicketCreatedEvent>** {

**subject:** Subjects.TicketCreate = Subjects.TicketCreate;

**queueGroupName:** string = "payment-service";

**messageToListener**(

**parsedMessage:** TicketCreatedEvent["data"],

**msg:** Message

): void {

**console.**log(

`${**this.**subject} \n Sequence No ${**msg.**getSequence()} data = ${

**parsedMessage.**title

}`

);

**msg.**ack();

}

}

**Explanation:**

Here we are creating **CreateTicketEventListener** which will **extends** an **Abstact Generic Class** Listener

**Generic Class** -> It is an type of class which reqire an interface so that it can detect and correct the types of variables or methods going to declare in the new Class we creating, which is **CreateTicketEventListner.**

**TicketCreatedEvent ->** It is an **Interface** used by Generic Abstract Class named **Listener**

**import** { Subjects } **from** "./subjects";

**export interface** TicketCreatedEvent {

**subject:** Subjects.TicketCreate;

**data:** {

**id:** string;

**title:** string;

**price:** number;

**userId:** string;

};

}

1. Now **Stan.on** will listen for ‘connect’ , ‘error’, ‘close’ events provided to client while connection with NATS Streaming Server.
2. Remember implement **Gracefull Shutdown**

Our CMD or Terminal are designed to listen for Signals, with respect to environment(NODE) like, Interupt Signal, Terminate Signal ,etc

Interupt Signal : **SIGINT**

terminate Signal: **SIGTERM**

So add some line to **process.on** to catch the signal of SIGINT,SIGNTERM . At these types of signal will create an immediate shutdown to NATS Streaming Server without Closing it and will result in memory leak problems.

**SOLUTION ->** call **stan.close(),** Wheneven an interupt or terminate signal given to Terminal

1. Similary Implement Publisher create-ticket-publisher Class extends Abstract Class base-publisher as it is **generic** , will need interface TicketCreatedEvent.
2. Now implement script “listen”:”ts-node-dev --rs --notify false src/listener”.
3. Now implement script “publish”:”ts-node-dev --rs --notify false src/publisher”.
4. Run publisher with => **npm run** publish
5. Run listener with => **npm run** listen.

Lets Understanding **Listener** Class (base-listener.ts)

**import** { Stan, Message } from "node-nats-streaming";

**import** { Subjects } from "./subjects";

**export** **interface** ListenerData {

**subject:** Subjects;

**data:** any;

}

**export abstract class** Listener<L extends ListenerData> {

**abstract** subject: L["subject"];

**abstract** messageToListener(parsedMessage: L["data"], msg: Message): void;

**abstract** queueGroupName: string;

**protected** ackWait = 5 \* 1000;

**private** client: Stan;

constructor(client: Stan) {

**this.**client = client;

}

subscriptionOption() {

return **this.client**

.subscriptionOptions()

.setManualAckMode(true)

.setDeliverAllAvailable()

.setAckWait(this.ackWait)

.setDurableName(this.queueGroupName);

}

listen() {

const subscribe = this.client.subscribe(

this.subject,

this.queueGroupName,

this.subscriptionOption()

);

subscribe.on("message", (msg: Message) => {

**const** parsedMessage = **this.**messageDataConvertor(msg);

**this.messageToListener**(parsedMessage, msg);

});

}

messageDataConvertor = (**msg:** Message) => {

const data = **msg.**getData();

**return** typeof data === "**string**"

? JSON.parse(data)

: JSON.parse(data.toString("**utf8**"));

};

}

Lets understand how to start listening to an subject via subscribe

Here client.subscribe(subject, queueGroupName, subscriptionOption)

1. subject => it is the channel to whom we want to subscribe and want to listen when even that channel publish any Event.
2. queueGroupName => it is like an identifier to many same type subject listener and act as loadBalancer. It is responsible to listen equaly from all listeners with same subject. It will be help full for setDurableName, setAllDeliverAvailable because when ever any listener gets down and publisher are continuously publishing event then natsStreamingServer will preserve every Events with the name of queueGroupName and

setAllDeliverAvailable will help to send all Events to listener which we preserved in NatsStreamingServer but

setDurableName will help to identify(filter) that how many events out of all events which are not listened by any listeners and will only send those events to listeners when it get starts.

1. subscriptionOption =>

subscriptionOption() will help to create all subsription and put it in object format.

setManualAckMode() help to send message recived acknowledgment manual. Remember to use message.ack() or msg.ack()

setAckWait() it receives the time in seconds (sec) , it hekps to define time for auto send message acknowledment,in case listener is not responding or user forget to handle the case of acknowledgement.

setAllDeliverAvailable() will help to send all Events to listener which we preserved in NatsStreamingServer but

setDurableName() will help to identify(filter) that how many events out of all events which are not listened by any listeners and will only send those events to listeners when it get starts.

**HOW TO START MULTIPLE LISTENERS WITH SAME SUBJECT ?**

Just start listener code in different terminal simultanously to recive events of same subject and sameQueueGroup for better preserving of events in NATS Streaming Server memory.

**How to Create Publisher ?**

Code of base-publisher (Abstract Class Publisher)

**import** { Stan } from "node-nats-streaming";

**import** { Subjects } from "./subjects";

**interface** PublisherData {

subject: Subjects;

data: any;

}

export **abstract class** Publisher<**P** **extends** PublisherData> {

**private** **client**: Stan;

**abstract** **subject**: **P["subject"]**;

constructor(client: Stan) {

this.client = client;

}

**publish** = (data: **P["data"]**): Promise<void> => {

return new Promise((resolve, reject) => {

**this.client.publish**(this.subject, JSON.stringify(data), (err) => {

if (err) {

reject(err);

}

**console.log**(`${this.subject} Event Published`);

resolve();

});

});

};

}

Here PublisherData is an interface which will help to declare the **Publisher Class** it type of **Generic** (Data Type inside class).

client.publish(subject,data,cbFunctionWithError).

subject => It is the type of Event we want to publish to NATS StreamingServer.

data => Alaways in string (convert **json or object** to string ) it is the data send to NATS Streaming Server for Listner to receive it.

cbFunction => it is an function who run after publishing Event and will recive an error Arg in case of Error while publishing.

Code to Launch Publish Event.

**import** nats from "node-nats-streaming";

**import {** PublishCreateTicket } from "./events/PublishCreateTicket";

**console.clear()**;

**const** **stan** = nats.connect("ticketing", "abc", {

url: "**http://localhost:4222**",

});

**stan.on**("connect", async () => {

**console.log**("Publisher connected to NATS");

**try** {

await **new** PublishCreateTicket(**stan**).publish({

id: "123",

title: "concert",

price: 20,

userId: "sadfsfssd",

});

} catch (err) {

**console.error**(err);

}

// **const** data = JSON.stringify({

// id: "123",

// title: "concert",

// price: 20,

// });

// **stan.publish**("ticket:created", data, () => {

// console.log("Event published");

// });

});