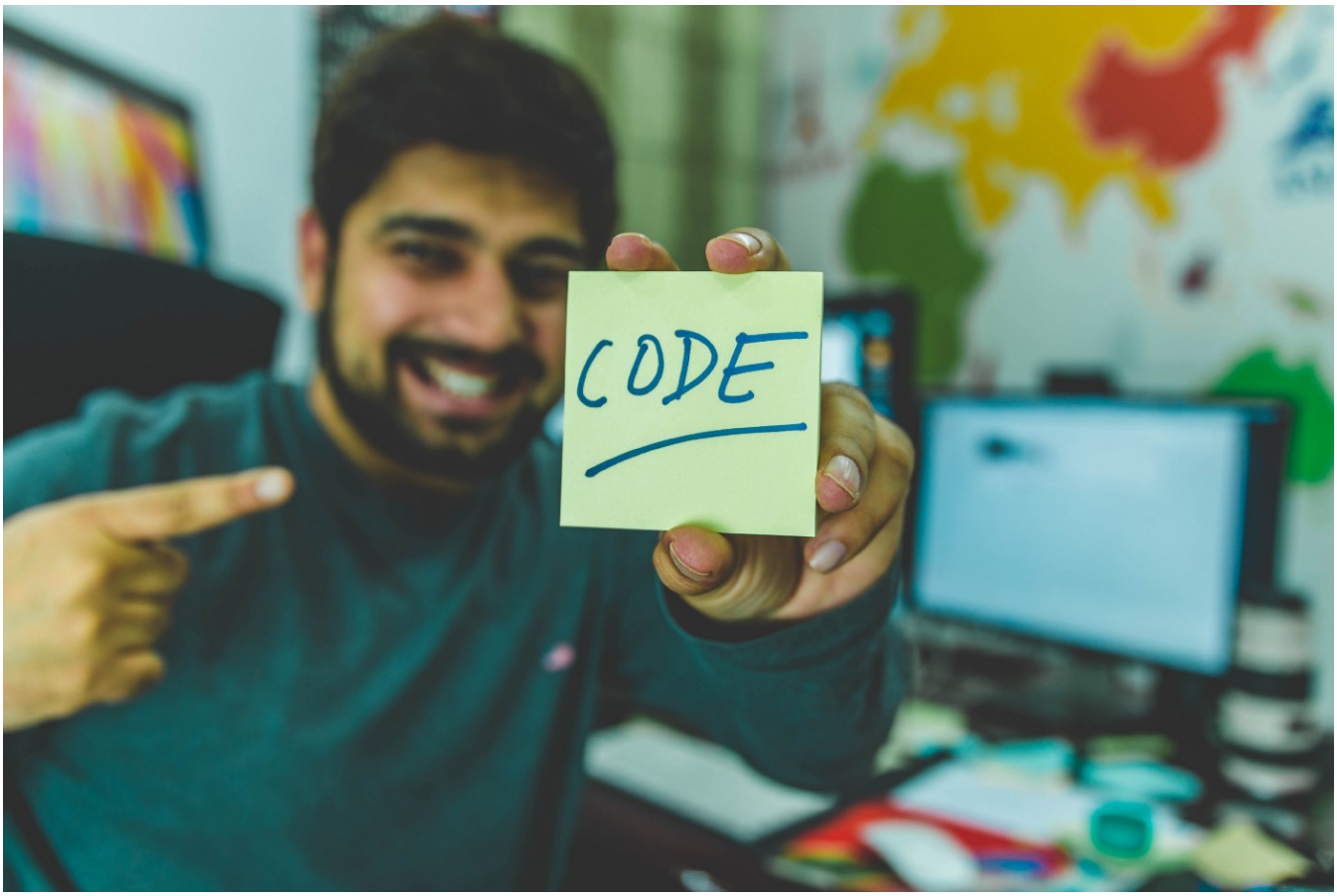
ganesh mani

55 Followers About Follow

Implementing Saga Pattern in Nodejs Microservices — Cloudnweb

ganesh mani Jan 18, 2020 · 8 min read



In this article, we will how to implement saga pattern in nodejs microservices. Implementing Saga Pattern in Nodejs Microservices.

Firstly, what is a saga pattern in microservices and why kind of problem that it solves.

Saga Pattern

Let's try to understand the saga pattern with an example. consider an application like Asana where a project contains a task and task contains a subtask.

project details will be in project service and task and subtask will be in subtask service with each service has their own database.

What happens to Task and Subtask service when user deletes the Project. How can you maintain the data consistency across all services.

That is to say, Saga pattern solves the problem of data consistency across different services.

Saga Pattern types

There are two kind of methods that are used in saga pattern. they are,

Orchestration-based Saga

Orchestration based saga is a pattern method where a orchestrator service maintains the communication(command/reply) between services.

So, it helps to mains the data consistency across services.

Choreography-Based Saga

In this method, there is no central orchestrator. each services will have a command/reply events. so, for every reply, it will update the database consistently.

Implementation in Nodejs Microservices

Mainly, we will see an example of nodejs microservices where data consistency is a crucial part.

Complete Source code can be found here

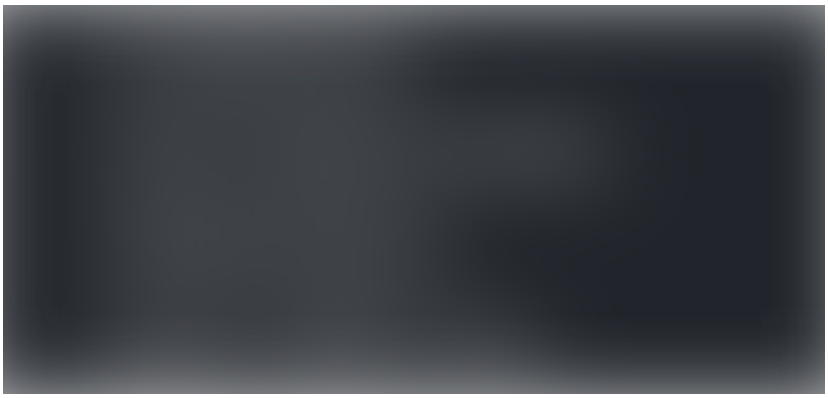
Note : This Code is just to demonstrate how saga pattern will work in nodejs microservices. it doesn't have all the business logics on services. Feel free to complete the service if you are interested. (PR's are always welcome)

In this example, we have an e-commerce application. it contains order service, payment service and stock service.

whenever user places an order, we need to implement the complete flow of order, payment and delivery of items which involves order service, payment service and stock service.

Here, data consistency place a crucial. Let's see how to implement it Orchestator based saga pattern.

Structure



- **KafkaBroker** — it contains all the kafka producer, consumer and routes logic. All the services will be using this to publish and receive events from kafka.
- **orchestatorService** — it contains all the logics to implement the orchestration of saga pattern.

- **orderService** — this service will handle all the order business logics.
- **paymentService** — it will handles all the payment business logics.

we will be using kafka-node for kafka communication in nodejs. if you prefer kafkajs, feel free to do that.

KafkaBroker

If you are new to kafka, read this article to get a good grasp of it.

Create a directory **kafkaHandler** inside kafkaBootstrap. Here, we are going to create **producer** and **consumer** logics for kafka.

After that, create a file called **producer.js** and add the following code

```
const Kafka = require('kafka-node');

const Producer = Kafka.Producer;
const client = new Kafka.KafkaClient();

// For creating Topics.
// Only admins were able to create topics
const admin = new Kafka.Admin(client);

let producer;

let producerReady;

const bindListeners = function bindListeners() {

  producerReady = new Promise((resolve, reject) => {
    producer.on('ready', () => {
      console.log('producer ready');
      resolve(producer);
    });

    producer.on('error', err => {
      console.log('producer err', err);
      reject(err);
    })
  })
};

const initializeProducer = () => {
  producer = new Producer(client);

  bindListeners();
}

/*
 * A Higher level producer which sends a message to a particular
```

```
topic
 */
const ProducerService = function ProducerService() {
  initializeProducer();
}

/*
 * Sends a message from the kafka instance
 **/
ProducerService.prototype.produce = function produce(
  topic,
  messages,
  partition = 0,
) {
  // Returns data if producer success
  return producerReady
    .then(producer => {

      const payload = [{ topic, messages, partition }];
      return new Promise((resolve, reject) => {
        producer.send(payload, function (err, data) {
          if (err) {
            console.log('Error while producing data in this
service')
            reject(err);
          }
          resolve(data);
        })
      });
    });
}

ProducerService.prototype.createTopic = function createTopic(
    topics
) {
    return producerReady.then(producer => {
        return new Promise((resolve,reject) => {
            producer.createTopics(topics,(err,res) => {
                if(err){
                    console.log('Error while creating a topic');
                    reject(err);
                }

                console.log('Topics created successfully');
                resolve(res);

            })
        })
    })
}

module.exports = ProducerService;
```

Here, we have few methods on the producer. they are,

- Initializing the producer by binding **on ready** and **on error** call back functions.
- **produce** method that takes the topic and message and send the message to the specified topic.
- **createTopic** method that creates a topic if not exists.

create a file called **Consumer.js** and add the following code

```
const kafkaNode = require('kafka-node');

const client = new kafkaNode.KafkaClient();
const offset = new kafkaNode.Offset(client);

const Consumer = kafkaNode.Consumer;

let consumer;

let consumerReady;

var defaultOptions = {
  encoding: 'utf8', // default is utf8, use 'buffer' for binary data
  fromOffset: -1, // default,
  autoCommit: true,
};

const bindEventListeners = function bindEventListeners(options,
topic) {
  consumerReady = new Promise((resolve, reject) => {
    try {
      consumer = new Consumer(
          client,
          [],
          options
      );
      consumer.on('error', (err) => {
        console.log(`Error occured on consumer group ${topic}`);
      })
      resolve(consumer);
    } catch (e) {
      reject(e);
    }
  });
};

const initializeConsumer = function initializeConsumer(defaultTopic)
{
  const options = defaultOptions;

  bindEventListeners(options, defaultTopic);
};

const ConsumerService = function ConsumerService(defaultTopic) {
```

```
  console.log('initializing consumer ')
  initializeConsumer(defaultTopic);
}

ConsumerService.prototype.addTopics = function addTopics(topicArray)
{
  return new Promise((resolve, reject) => {
    consumerReady
      .then((consumer) => {
        console.log('adding topics ', topicArray);
        consumer.addTopics(topicArray, function (err, added) {
          console.log('topics added ', err, added);
          resolve(added);
        });
      })
      .catch((e) =>{
        console.log('errror while creating topic ', e);
      });
  });
};

ConsumerService.prototype.consume = function consume(cb) {
  consumerReady
    .then((consumer) => {
      console.log('consumer ready');
      consumer.on('message', (message) => {
        // console.log('recieved message ', message);
        cb(message);
      })
    })
    .catch((e) =>{
      console.log('errror while consuming', e);
    })
}

module.exports = ConsumerService;
```

Here, we have few methods for consumer. they are,

- Initializing the consumer by binding **on ready** and **on error** call back functions.
- **addTopic** method will add the topic for the consumer to consume.
- **consume** method will receive the message from producer and sends it to callback

After that, create a file called kafkaBootstrap.js and add the following code

```
const kafka = require('kafka-node');

const Producer = require('../kafkaBroker/kafkaHandler/Producer');
```

```
const producer = new Producer();

const topics = [
    { topic : 'ORDER_SERVICE',partitions : 1,replicationFactor : 1
},
    { topic : 'PAYMENT_SERVICE',partitions : 1,replicationFactor : 1
},
    { topic : 'STOCK_SERVICE',partitions : 1,replicationFactor : 1
},
    { topic : 'ORCHESTATOR_SERVICE',partitions : 1,replicationFactor
: 1 }
]

producer.createTopic(topics).then(res => {

})
.catch(err => {
    console.log(`Error ${err}`)
})
```

Here, we create a topic if not exists, run these code for the first time to create topics.

Order Service

- **Controller** — it handles the request and business logics.
- **eventHandler** — it helps to handle all the kafka messages and maps it with business logics.

- **Model** — it contains all the database models.

After that, create a file **app.js** and add the following code

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');

const Consumer = require('../../kafkaBroker/kafkaHandler/Consumer');
const eventHandler = require('./eventHandler');
const CreateOrder = require('./Controller/createOrder');
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended : false}));

mongoose.connect("mongodb://localhost:27017/orderdb",{
useNewUrlParser : true,useUnifiedTopology : true }).then(data => {

    app.post('/createorder',CreateOrder);

    const PORT = 3000;

    app.listen(PORT,() => {
        console.log('server is running on port 3000');
    })

    const consumer = new Consumer();

    consumer.addTopics(["ORDER_SERVICE","SERVICE_REPLY"]).then(() =>
{
        consumer.consume(message => {
            console.log("consumed message",message);
            eventHandler(JSON.parse(message));
        })
    })

})
.catch(err => {
    console.log(`Error in Mongo Connection ${err}`)
})
```

Here, we setup a mongodb connection and add topics to kafka consumer of order service.

Once it consumes the message, eventHandlers takes those message and performs some business logics.

Further, create a file **createOrder.js** in Controller and add the following code,

```
const uuidv1 = require('uuid/v1');

const OrderModel = require('../Model/orderModel');
const Producer =
require('../../../kafkaBroker/kafkaHandler/routes');
const CreateOrder = async (req,res) => {

    try {
        const name = req.body.name;
        const itemCount = req.body.itemCount;
        const amount = req.body.amount;

        const order = await new OrderModel({ name : name,itemCount :
itemCount,transactionId : uuidv1(),status : 'PENDING' });

        await order.save();

        res.send(order);

        Producer({
            topic : 'ORDER_CREATION_TRANSACTIONS',
            type : 'ORDER_CREATED',
            payload : {
                data : {
                    id : order._id,
                    transactionId : order.transactionId,
                    amount : amount
                }
            }
        })
    }
    catch(e){
        console.log(e);
    }
}
module.exports = CreateOrder
```

Mainly, controller takes the request and insert the data into database. once it does that, it will sends that data to kafka producer by initiating the create order transaction.

Orchestator service

Like said, Main purpose of having orchestator service to orchestrate the command and reply.

Mainly, Everything is a Transaction here. For every transactions, it will orchestrate the status across different services.

Firstly, create a file **bootstrap.js** and add the following code,

```
const Consumer = require('../../kafkaBroker/kafkaHandler/Consumer');
const Transactions = require('./Transactions');
try {

const consumer = new Consumer();

consumer.addTopics(["ORCHESTATOR_SERVICE"]).then(() => {
    consumer.consume(message => {
        console.log("consumed message",message);
        Transactions(JSON.parse(message.value));
    })
})

console.log("Orchestator Started successfully");

}
catch(e){
    console.log(`Orchestrator Error ${e}`);
}
```

create a file **orderCreationTransactions.js** and add the following code,

```
const Producer =
require('../../../kafkaBroker/kafkaHandler/routes');

module.exports = (message) => {

    switch(message.type) {
```

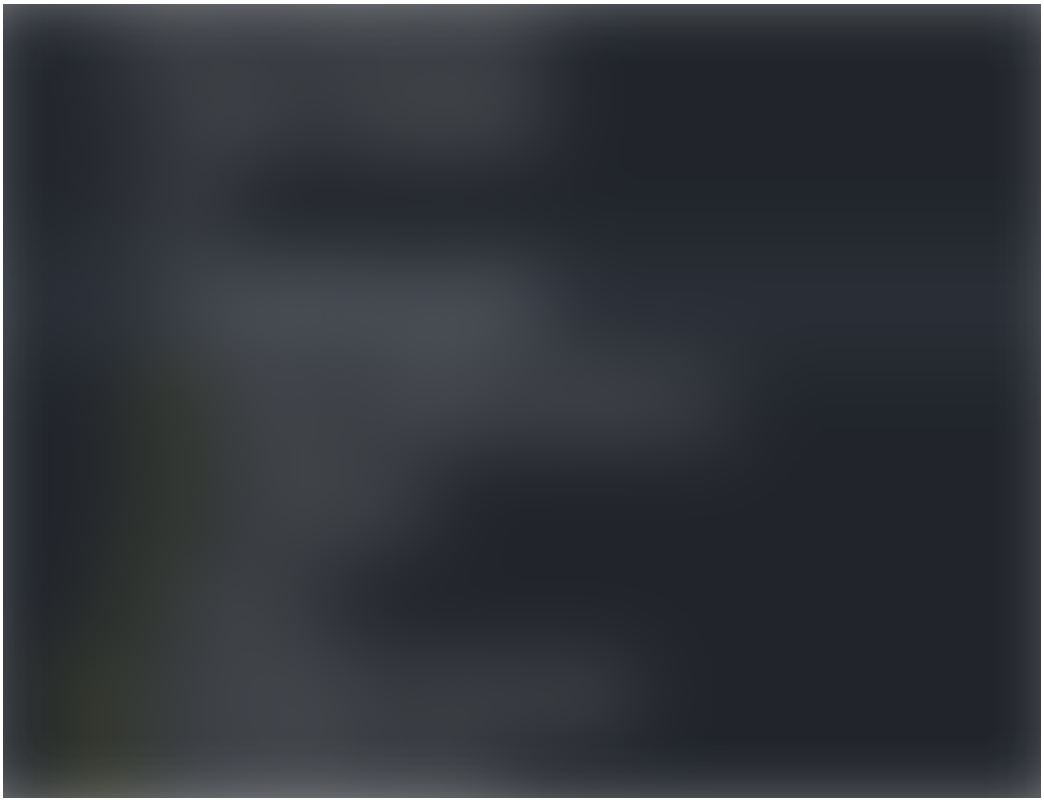
```
            case 'ORDER_CREATED':
                 Producer({
                    topic : 'EXECUTE_PAYMENT',
                    payload : {
                        data : message.payload.data
                    }
                })
                break;
            case 'PAYMENT_COMPLETED_STATE' :
                Producer({
                    topic : '',
                    payload : {
                        data : message.payload.data
                    }
                })
            default:
                break;

        }
    }
```

Once, it receives the state. orchestator will directs the transaction state to appropriate services.

Payment Service

Once payment service receives a command from orchestator service. it will do the business logic and updates the status to orchestaor servcice.

based on the status, it will perform the further actions to respective services.



add the following code in **app.js**

```
const Consumer = require('../../kafkaBroker/kafkaHandler/Consumer');
const eventHandler = require('./eventHandler');
try {

    const consumer = new Consumer();

    consumer.addTopics(["PAYMENT_SERVICE"]).then(() => {
        consumer.consume(message => {
            console.log("consumed message",message);
            eventHandler(JSON.parse(message.value));
        })
    })

    console.log("Payment service Started Successfully");

    }
    catch(e){
        console.log(`Orchestrator Error ${e}`);
    }
```

It add the topics and when a message is received. it will send it to eventhandler.

executePayment.js

```
const Producer  =
require('../../../kafkaBroker/kafkaHandler/routes');
module.exports = (data) => {
    /** Database Layer Logic Comes Here  */
    try {
        console.log("data",data);
        Producer({
            topic : 'ORDER_CREATION_TRANSACTIONS',
            type : 'PAYMENT_COMPLETED_STATE',
            payload : {
                transactionId : data.transactionId
            }
        })

    }
    catch(e){
        console.log(e);
    }
}
```

Summary

In Conclusion, maintaining microservice that implement saga pattern will be a bit complex. but, it is worth to solve the problem using saga pattern.

we will see how to implement Choreography-Based Saga in upcoming article.

Originally published at https://cloudnweb.dev on January 18, 2020.

Microservices Nodejs