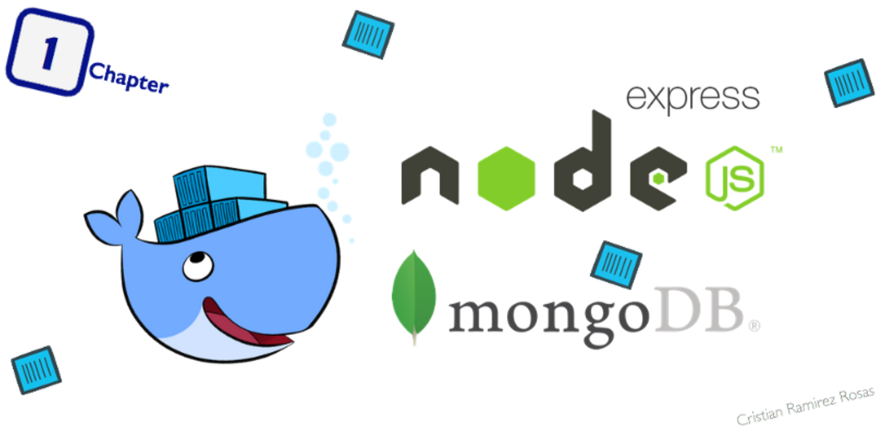
Cristian Ramirez Follow
FullStack MEAN Engineer with some Docker knowledge Linkedin: https://www.linkedin.com/in/cristian-r...
Jan 23 · 11 min read

Build a NodeJS cinema microservice and deploying it with docker—part 1



Images from google—cover made by me

This a the first chapter of the series "Build a NodeJS cinema microservice", this series is about, building **NodeJS microservices** and deploying them into a **Docker Swarm Cluster.**

In this article i'm going to show you, what all the fuss is about in, building a **microservice**, and then deploy it to a **Docker** container. We'll use a simple NodeJS service with a MongoDB for our backend.

What we are going to use for this article is:

- NodeJS version 7.2.0
- MongoDB 3.4.1
- Docker for Mac 1.12.6

Prerequisites to following up the article:

- Basic knowledge in NodeJS
- Basic knowledge in Docker (docker installed)

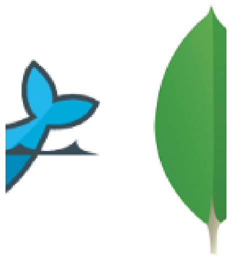
- Basic knowledge in MongoDB (a database service running)

I highly suggest to follow up my previous article “*How deploy a mongoDB replica set with Docker*”, to have a **database service up and running**.

How to deploy a MongoDB Replica Set using Docker

This article is going to be a walk-through in how to set up a MongoDB replica set with authentication using docker.

medium.com



But first what is a microservice ?

> *A microservice is a single self-contained unit which, together with many others, makes up a large application. By splitting your app into small units every part of it is independently deployable and scalable, can be written by different teams and in different programming languages and can be tested individually.—Max Stoiber*

> *A microservice architecture means that your app is made up of lots of smaller, independent applications capable of running in their own memory space and scaling independently from each other across potentially many separate machines.—Eric Elliot*

BENEFITS OF MICROSERVICES

- The application starts faster, which makes developers more productive, and speeds up deployments.
- Each service can be deployed independently of other services— easier to deploy new versions of services frequently
- Easier to scale development and can also have performance advantages.
- Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack.

- Microservices are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components.
- Decoupled services are also easier to recompose and reconfigure to serve the purposes of different apps (for example, serving both the web clients and public API).

DRAWBACKS OF MICROSERVICES

- Developers must deal with the additional complexity of creating a distributed system.
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
- As you're building a new microservice architecture, you're likely to discover lots of cross-cutting concerns that you did not anticipate at design time.

# How to build a microservice with NodeJS

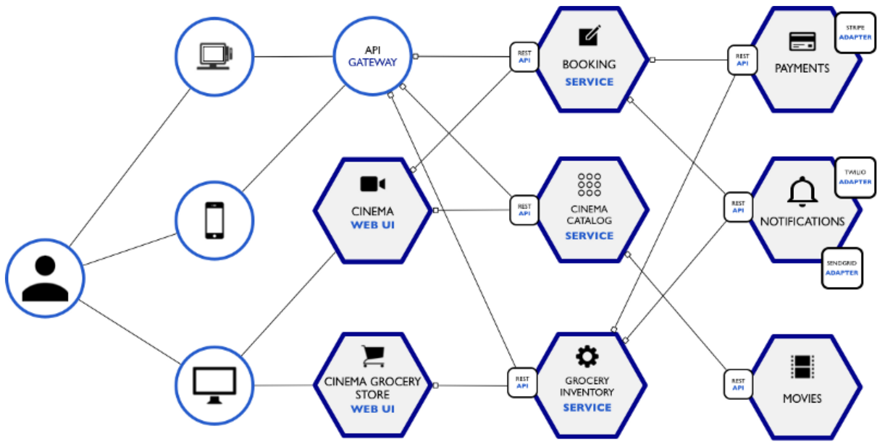
Microservices can build applications from many **simple**, **single-purpose**, **easy to use** components that enables delivering better software faster. Even existing monolith architectures could be transformed using the microservices pattern, but the question is how we can make one with **NodeJS ?**

Well JavaScript is currently the most popular programming language with the richest OSS module ecosystem. So, for building a microservice, we need to build an **API (**Application Programming Interface**)**, but what module, library, framework should i use ?, i just found a smiliar question in quora like this: **Which Node.js framework is better for building a RESTful api ?**

And in inside the question one user gives a useful answer by providing a very good **website** which literately shows every framework and library, that we can use for building an API, so is up to you which will you use. In this article we will use **ExpressJS** to build our API's and our microservices.

So let's leave behind all the theory and get our hands 🖖 dirty with coding an to learn all about this fuss 👨💻👨🎨🖥.

# Architecture for our microservice



Cinema Microservice Example

Ok, let's imagine that we are woking in the IT department of **Cinépolis** (A Mexican cinema), and they give us the task to restructure their tickets and grocery store monolithic system to a microservice.

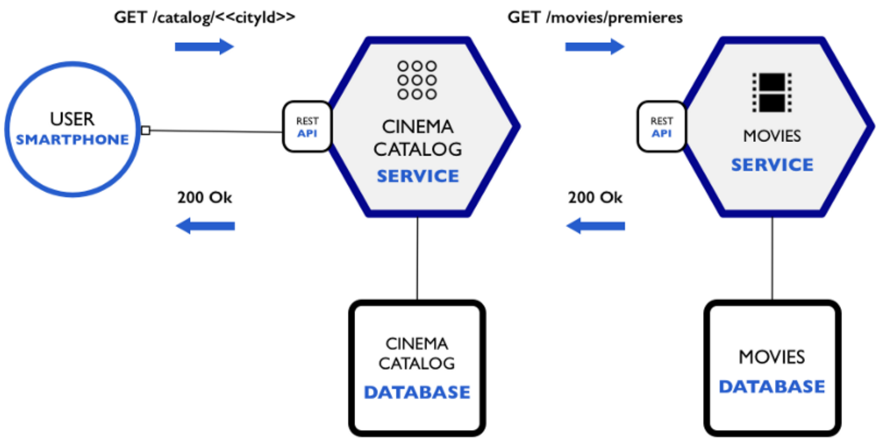
So for the first part of the "Build a NodeJS cinema microservice"— series, we are going to focus only in the **movies catalog service.**

In this architecture we saw that we have 3 different devices who uses the microservice, the POS(point of sale), a mobile/tablet, and a computer, where the POS and the mobile/tablet has its own application developed (in electron) and consumes directly the microservice, and the computer accesses the microservice through web apps (web apps are considered by the gurus also like microservices 🤓).

# Building the microservice

Ok so, let's simulate that we are going to request a booking in our favorite cinema for a movie premiere.

First we want to see which movies are currently available in the cinema. The following diagram shows us how is going to be the inner communication with microservices through REST.



Our API for the **movies service** will have this raml specifications:

```
#%RAML 1.0
title: cinema
version: v1
baseUri: /

types:
  Movie:
    properties:
      id: string
      title: string
      runtime: number
      format: string
      plot: string
      releaseYear: number
      releaseMonth: number
      releaseDay: number
    example:
      id: "123"
      title: "Assasins Creed"
      runtime: 115
      format: "IMAX"
      plot: "Lorem ipsum dolor sit amet"
      releaseYear : 2017
      releaseMonth: 1
      releaseDay: 6

  MoviePremieres:
    type: Movie []

```

> *If you don't know what **RAML** is you can check this great tutorial*

The structure for the API projects will look like this:

```
- api/                    # our apis
- config/                 # config for the app
- mock/                   # not necessary just for data
examples
- repository/             # abstraction over our db
- server/                 # server setup code
```

```
  - package.json          # dependencies
  - index.js              # main entrypoint of the app
```

Ok so let’s start. The first section to look at is the `repository.` Here is where we do our query’s to the database.

```
// repository.js
'use strict'
// factory function, that holds an open connection to the d
// and exposes some functions for accessing the data.
const repository = (db) => {

  // since this is the movies-service, we already know
  // that we are going to query the `movies` collection
  // in all of our functions.
  const collection = db.collection('movies')

  const getMoviePremiers = () => {
    return new Promise((resolve, reject) => {
      const movies = []
      const currentDay = new Date()
      const query = {
        releaseYear: {
          $gt: currentDay.getFullYear() - 1,
          $lte: currentDay.getFullYear()
        },
        releaseMonth: {
          $gte: currentDay.getMonth() + 1,
          $lte: currentDay.getMonth() + 2
        },
        releaseDay: {
          $lte: currentDay.getDate()
        }
      }
      const cursor = collection.find(query)
      const addMovie = (movie) => {
        movies.push(movie)
      }
      const sendMovies = (err) => {
        if (err) {
          reject(new Error('An error occured fetching all m
        }
        resolve(movies)
      }
      cursor.forEach(addMovie, sendMovies)
    })
  }

  const getMovieById = (id) => {
    return new Promise((resolve, reject) => {
      const projection = { _id: 0, id: 1, title: 1, format:
      const sendMovie = (err, movie) => {
        if (err) {
          reject(new Error(`An error occured fetching a mov
        }
        resolve(movie)
```

As you may noticed, we provide a `connection` object to the only exposed method of the repository **connect(**connection**)**, you can see here one of the biggest powers that javascript has "**closures**" the repository object is returning a closure where every function has access to the `db` object and to the `collection` object, the `db` object is holding the database connection. Here we are abstracting the type of database we are connecting to, the repository object doesn't know what kind of database is, in our case is a **MongoDB** connection, even though it doesn't have to know if it's a single database or a replica set connection, although that we are using mongodb syntax, we can abstract the repository functions even more by applying the *Dependency Inversion principle* from *solid principles*, from taking mongo syntax to another file and just call the interface of database actions (e.g. using mongoose models).

There's a `repository/repository.spec.js` file for testing this module, i am going to talk about test later on the article, but if you want to check it, you can find it here at the github repo branch step-1.

The next file we are going to look at is the `server.js.`

```
'use strict'
const express = require('express')
const morgan = require('morgan')
const helmet = require('helmet')
const movieAPI = require('../api/movies')

const start = (options) => {
  return new Promise((resolve, reject) => {
    // we need to verify if we have a repository added and
    if (!options.repo) {
      reject(new Error('The server must be started with a c
    }
    if (!options.port) {
      reject(new Error('The server must be started with an
    }
    // let's init a express app, and add some middlewares
    const app = express()
    app.use(morgan('dev'))
    app.use(helmet())
    app.use((err, req, res, next) => {
      reject(new Error('Something went wrong!, err:' + err)
      res.status(500).send('Something went wrong!')
```

Here what we are doing is, instantiating a new express app, verifying if we provide a repository and server port objects, then we apply some middleware to our express app, like `morgan` for logging, `helmet` for security, and a `error handling` function, and at the end we are exporting a start function to be able to start the server 😎.

> *Helmet includes a whopping 11 packages that all work to block malicious parties from breaking or using an application to hurt its users.*

If you want to hardener the microservice you can check this great article.

Ok now since our server is using our movieAPI, let's continue checking the file `movies.js`

```
'use strict'
const status = require('http-status')

module.exports = (app, options) => {
  const {repo} = options

  // here we get all the movies
  app.get('/movies', (req, res, next) => {
    repo.getAllMovies().then(movies => {
      res.status(status.OK).json(movies)
    }).catch(next)
  })

  // here we retrieve only the premieres
  app.get('/movies/premieres', (req, res, next) => {
    repo.getMoviePremiers().then(movies => {
      res.status(status.OK).json(movies)
    }).catch(next)
```

What we are doing here is creating the routes for our API, and calling our repo functions depending on the route listened, if you can see, our repo here is using an interface technique approach, here we are using the famously "coding for an interface not to an implementation", since the express routes dosen't know about if there's a database object, database queries logic, etc, it only calls the repo functions that handles all of the database concerns.

Ok all of our files has unit tests adjacent to the source, let's see how is our test for the `movies.js`

> *You can think of tests as safeguards for the applications you are building. They will run not just on your local machine, but also on the CI services so that failing builds won't get pushed to production systems.—Trace by RisingStack*

To write unit test, all the dependencies must be stubbed, meaning we are providing fake dependencies for a module. Let's see how is our `spec files.`

```
/* eslint-env mocha */
const request = require('supertest')
const server = require('../server/server')

describe('Movies API', () => {
  let app = null
  let testMovies = [{
    'id': '3',
    'title': 'xXx: Reactivado',
    'format': 'IMAX',
    'releaseYear': 2017,
    'releaseMonth': 1,
    'releaseDay': 20
  }, {
    'id': '4',
    'title': 'Resident Evil: Capitulo Final',
    'format': 'IMAX',
    'releaseYear': 2017,
    'releaseMonth': 1,
    'releaseDay': 27
  }, {
    'id': '1',
    'title': 'Assasins Creed',
    'format': 'IMAX',
    'releaseYear': 2017,
    'releaseMonth': 1,
    'releaseDay': 6
  }]

  let testRepo = {
    getAllMovies () {
      return Promise.resolve(testMovies)
    },
    getMoviePremiers () {
      return Promise.resolve(testMovies.filter(movie => mo
    },
    getMovieById (id) {
      return Promise.resolve(testMovies.find(movie => movi
    }
  }

  })

  afterEac
    app.clo
    app = nul
  })

  it('can return al
    request(app)
      .get('/movies')
      .expect((res) => {
        res.body.should.co
          'id': '1',
          'title': 'Assasins C
          'format': 'IMAX',
          'releaseYear': 2017,
          'releaseMonth': 1,
          'releaseDay': 6
        })
      })
      .expect(200, done)
  })

  it('can get movie premiers', (done) => {
    request(app)
    .get('/movies/premiers')
    .expect((res) => {
      res.body.should.containEql({
        'id': '1',
```

As you can see we are stubbing the dependencies for the `movies API` , and for the server we are verifying that we need to provided a server port and a repository object.

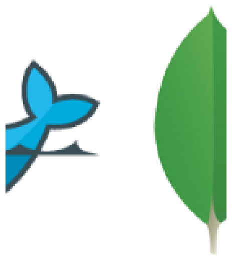
You can check all test files in the github repo of the article.

Let’s continue with how to create the `db connection object` we passed to the **repository module**, now definition says that every microservice has have it’s own database, but for our example we are going to use a **mongoDB replica set server**, but each microservice will have it own database, if you don't now how to configure a mongoDB replset server, you can check this article for a deeper explanation.

How to deploy a MongoDB Replica Set using Docker

This article is going to be a walk-through in how to set up a MongoDB replica set with authentication using docker.

medium.com



Here is the configuration we need to connect to a **MongoDB** database from **NodeJS.**

```
const MongoClient = require('mongodb')

// here we create the url connection string that the driver
const getMongoURL = (options) => {
  const url = options.servers
    .reduce((prev, cur) => prev + `${cur.ip}:${cur.port},`,

  return `${url.substr(0, url.length - 1)}/${options.db}`
}

// mongoDB function to connect, open and authenticate
const connect = (options, mediator) => {
  mediator.once('boot.ready', () => {
    MongoClient.connect( getMongoURL(options), {
        db: options.dbParameters(),
        server: options.serverParameters(),
        replset: options.replsetParameters(options.repl)
      }, (err, db) => {
        if (err) {
          mediator.emit('db.error', err)
        }
```

There's probably a lot of better ways to do this, but basically we can create a connection to a mongoDB with replica set like this.

As you can see, we are passing a `options` object, that has all the parameters that the mongo connection needs, and also we are passing an `event—mediator` object that will emit the the `db` object when we pass the authentication process.

> ***Note*** here i am using an event-emitter object because, with a promise approach for some reason it didn't return the db object once it pass the authentication, the sequence gets idle.—so this could be a good challenge to see what's happening and try to use a promise approach.*

Now since we are passing a `options` object for the parameters, let's see from where this is coming from, so the next file to look at is the `config.js`

```
// simple configuration file

// database parameters
const dbSettings = {
  db: process.env.DB || 'movies',
  user: process.env.DB_USER || 'cristian',
  pass: process.env.DB_PASS || 'cristianPassword2017',
  repl: process.env.DB_REPLS || 'rs1',
  servers: (process.env.DB_SERVERS) ? process.env.DB_SERVER
    '192.168.99.100:27017',
    '192.168.99.101:27017',
    '192.168.99.102:27017'
  ],
  dbParameters: () => ({
    w: 'majority',
    wtimeout: 10000,
    j: true,
    readPreference: 'ReadPreference.SECONDARY_PREFERRED',
    native_parser: false
  }),
  serverParameters: () => ({
    autoReconnect: true,
    poolSize: 10,
    socketoptions: {
      keepAlive: 300,
      connectTimeoutMS: 30000,
      socketTimeoutMS: 30000
    }
  }),
  replsetParameters: (replset = 'rs1') => ({
    replicaSet: replset,
```

Here is our config file, mostly all config codes are hard coded, but as you can see some attributes uses environment variables as an option. Environment variables are considered best practices, because this can hide database credentials, server parameters, etc.

And finally the last step for coding our API for the `movies-service` is putting together everything with an `index.js.`

```
'use strict'
// we load all the depencies we need
const {EventEmitter} = require('events')
const server = require('./server/server')
const repository = require('./repository/repository')
const config = require('./config/')
const mediator = new EventEmitter()

// verbose logging when we are starting the server
console.log('--- Movies Service ---')
console.log('Connecting to movies repository...')

// log unhandled execpetions
process.on('uncaughtException', (err) => {
  console.error('Unhandled Exception', err)
})
process.on('uncaughtRejection', (err, promise) => {
  console.error('Unhandled Rejection', err)
})

// event listener when the repository has been connected
mediator.on('db.ready', (db) => {
  let rep
  repository.connect(db)
    .then(repo => {
      console.log('Repository Connected. Starting Server')
      rep = repo
      return server.start({
        port: config.serverSettings.port,
        repo
```

Here we are composing all the movies API service, we have a little error handling, then we are loading the configurations, starting the repository and finally starting the server.

So until now we have finished everything that concerns with a API development, you can check the repo at the step-1 branch.

If you go to the github repo of the article, you will see that there’s some commands for:

```
npm install          # setup node dependencies
npm test             # unit test with mocha
npm start            # starts the service
npm run node-debug   # run the server in debug mode
npm run chrome-debug # debug the node with chrome
npm run lint         # lint the code with standard
```

And finally we got our first microservice up and running locally executing the `npm start` command, but that is not what the title of the article said 🤔.

Now is time to put it in a **Docker** container as we mention it in the title of the article😁.

But first what we need is, have the **Docker** environment from the article for “creating a mongoDB replica set with docker”, and if you don’t have it you will have to do some additional modification steps to setup a database to our microservice, here some commands to be up to date just for testing purposes our movies-service.

So first let’s create our **Dockerfile** to dockerize our **NodeJS microservice**.

```
# Node v7 as the base image to support ES6
FROM node:7.2.0

# Create a new user to our new container and avoid the root
user
RUN useradd --user-group --create-home --shell /bin/false
nupp && \
    apt-get clean

ENV HOME=/home/nupp

COPY package.json npm-shrinkwrap.json $HOME/app/

COPY src/ $HOME/app/src

RUN chown -R nupp:nupp $HOME/* /usr/local/

WORKDIR $HOME/app
RUN npm cache clean && \
    npm install --silent --progress=false --production
```

```
RUN chown -R nupp:nupp $HOME/*
USER nupp

EXPOSE 3000

CMD ["npm", "start"]
```

We are taking the NodeJS image as the base for our docker image, then we create a user for avoiding non-root user, then we copy the src to our image then we install the dependencies, we expose a number port and finally we instantiate our movies-service.

Next we have to build our docker image, with the following command:

```
$ docker build -t movies-service .
```

Let's look at the build command first.

1. `docker build` tell the engine we want to create a new image.
2. `-t movies-service` tag this image with the tag `movies-service`. We can refer to this image by tag from now on.
3. `.` use the current directory to find the `Dockerfile`.

After some console output we have our new image with our NodeJS app, so now what we need to do is to run our image with the following command:

```
$ docker run --name movie-service -p 3000:3000 -e
DB_SERVERS="192.168.99.100:27017 192.168.99.101:27017
192.168.99.100:27017" -d movies-service
```

In the command above we are passing an env variable which is an array of servers that needs to connect to the mongoDB replset, this is just for ilustration, there are better ways to do this, like reading an env file for example.

Now that we have our container up and running let's retrieve our `docker-machine ip machine-name` to have the ip of our microservice, and we are ready to make an integration test to our microservice, another option for testing could be JMeter, is a good tool for simulating http requests, and here is a great tutorial of JMeter.

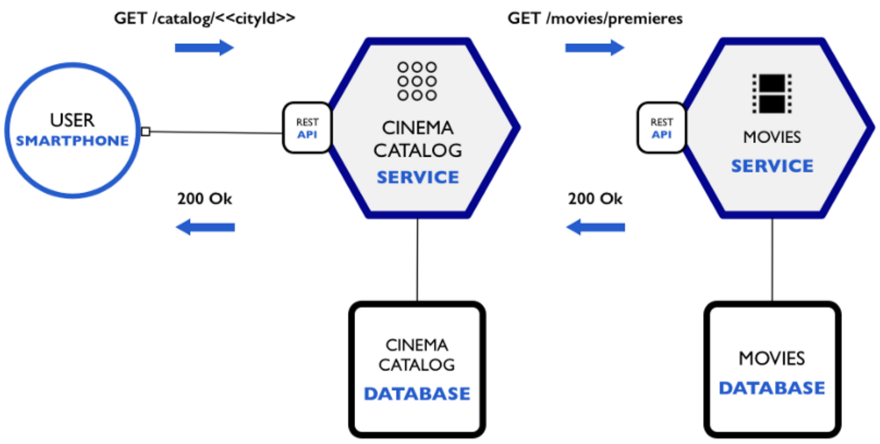
This is our `integration-test` that will check an API call :D.

```
/* eslint-env mocha */
const supertest = require('supertest')

describe('movies-service', () => {

  const api = supertest('http://192.168.99.100:3000')

  it('returns a 200 for a collection of movies', (done) =>

```

. . .

# Time for a recap

What we have done…



RESTful communication

We've made only the first part of this communication flow, we made the **movies service** for consulting the movie premiers available in the cinema, we built the **movies services** API in NodeJS, first we **design the api** with a **RAML** specification, then we start building our **API**, and made the corresponding **unit test**, finally we compose everything to have our **microservice complete,** and be able to start our movies service **server.**

Then we put our **microservice** into a **Docker** container, to be able to make some **integration test.**

We've seen a lot of development in **NodeJS**, but there's a lot more that we can do and learn, this is just a sneak peak. I hope this has shown some of the interesting and useful things that you can use for **Docker and NodeJS** in your workflow.

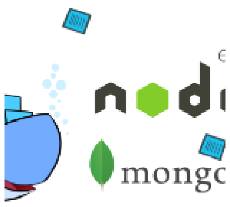
Let me remember you, this article is part of "***Build a NodeJS cinema microservice and deploying it with docker***" series.

I will put below the link of the continuation, the part 2 of this series.

Build a NodeJS cinema microservice and deploying it with docker (part 2)

This is the ✌ second article from the series "Build a NodeJS cinema microservice".

medium.com



. . .

# Complete code at Github

You can check the complete code of the article at the following link.

Crizstian/cinema-microservice

cinema-microservice - Example of a cinema microservice

github.com

. . .

# Further reading

To get better with NodeJS you can check this sites

- 10 Tips to Become a Better Node Developer in 2017
- NodeJS tutorial series—**Node Hero** (covers almost all node topics)
- NodeJS Security Checklist
- NodeJS at Scale

# References

- MongoDB NodeJS Driver
- MongoDB NodeJS Driver legacy connection
- API Workbench—Create an API Definition with RAML
- Pattern: Microservice Architecture

. . .

I hope you enjoyed this article, i'm currently still exploring the NodeJS and Microservices world, so i am open to accept feedback or contributions, and if you liked it, recommend it to a friend, share it or read it again.

You can follow me at twitter @cramirez_92
https://twitter.com/cramirez_92

Until next time 😁👦🎨👦💻