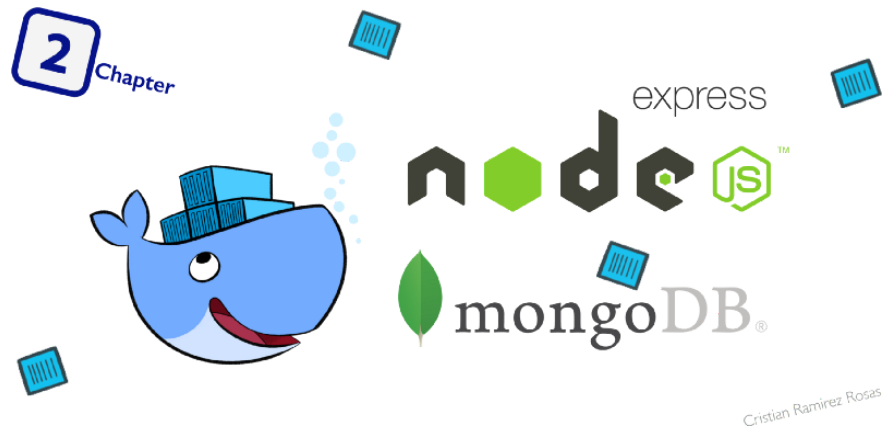




Cristian Ramirez [Follow](#)

FullStack MEAN Engineer with some Docker knowledge LinkedIn: <https://www.linkedin.com/in/cristian-r...>  
Jan 30 · 14 min read

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




images taken from the web—cover made by me

This is the *second* article from the series “Build a NodeJS cinema microservice”.

### ## A quick recap from our first chapter

- we talk about **what is a microservice**.
- we saw what are the **benefits** and **drawback** of **microservices**.
- we define a **cinema microservice architecture**.
- we design our **movies service API** specification with **RAML**.
- we develop our **movies service API** with **NodeJS** and **ExpressJS**.
- we made **unit testing** to our API.
- we compose our API to make it a service and run our **movies service** into a **Docker** container.
- we made an **integration test** to our **movies service** running on **Docker**.

if you haven't read the first chapter i will put the link below, so you can give it a look .

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

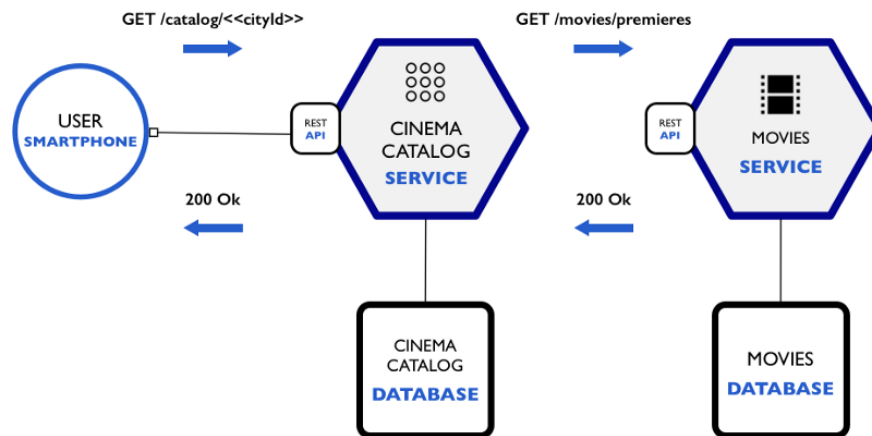
This is the first chapter of the series "Build a NodeJS cinema microservice", this series is about, building NodeJS...

medium.com



. . .

In this article we will be continuing building our **cinema microservice** and this time we are going to develop the **cinema catalog service**, to complete the following diagram.



What we are going to use for this article is:

- NodeJS version 7.2.0
- MongoDB 3.4.1
- Docker for Mac 1.13

Prerequisites to following up the article:

- Have completed the examples from the last chapter.

If you haven't, I have uploaded the repo at the following github repository, so you can be up to date, [repo link](#) at branch **step-1**.

## # Microservices security and ¿ HTTP/2 ?

In the first chapter we made a simple microservice that was implementing the HTTP/1.1 protocol. **HTTP/2 is the first major upgrade** to HTTP protocol in 15 years, is highly optimized and the performance is better. **HTTP/2 is the new web standard** which started as Google's SPDY protocol. It's already used by many popular websites and supported by most major browsers.

**HTTP/2 only has a few rules that must be met in order to implement it.**

- It only works with the HTTPS protocol (we'll need a valid SSL certificate).
- It's backward compatible. If a browser or a device where your application is running doesn't support **HTTP/2** it will fall back to HTTP1.1.
- It comes with great performance improvements out-of-the-box.
- It doesn't require you to do anything on the client side, only the server side for a basic implementation.
- A few new interesting features will speed up the load time of your web project in a way that is not even imaginable with HTTP1.1 implementation.

### An HTTP/2 Enabling Network Architecture for Microservices

*That means we need to enable a single connection between the client and the server and then utilize capabilities like Y-axis sharding (talking more about the scale cube on the series) in "the network" to maintain the performance benefits of HTTP/2 to the client while enabling all the operational and development benefits of a microservices architecture.*

So why are we implementing the newly fresh **HTTP/2 protocol**, well it is because as good developers we must secure our application, infrastructure, communications, the most we can, to prevent malicious

attacks, and also is because as good developers we follow the best practices that we consider benefit for us, like this one.

Some of the security best practices for microservices are like the following:

*Security will clearly play an important part in the decision to adopt and deploy microservices applications for production use. According to the research note, 2016 Software Defined Infrastructure Outlook by 451 Research, nearly 45% of enterprises have either already implemented or plan to roll out container-based applications over the next 12 months. As DevOps practices gain a foothold in enterprises and container applications become more prevalent, security administrators need to arm themselves with the know-how to secure applications. —@Ranga Rajagopalan*

- **Discover and monitor inter-service communications**
- **Segment and isolate applications and services**
- **Encrypt data in transit and at rest**

What we are going to do is to encrypt our microservices communication to meet compliance requirements and to improve security especially when the traffic crosses public networks, and that's one of the reasons of why we are going to implement **HTTP/2**, for better performance and security improvement.

## # Implementing HTTP/2 to the microservice

Firste let's update our **movies service** from the previous chapter and implement the **HTTP/2** protocol, but after we are going to create a **ssl** folder inside the **config** folder.

```
movies-service/config:/ $ mkdir ssl
```

```
movies-service/config:/ $ cd ssl
```

Now once inside the **ssl** folder, let's create a self-signed SSL certificate, to start implementing the **HTTP/2** protocol in our services.

```
# Let's generate the server pass key

ssl/: $ openssl genrsa -des3 -passout pass:x -out
server.pass.key 2048

# now generate the server key from the pass key

ssl/: $ openssl rsa -passin pass:x -in server.pass.key -out
server.key

# we remove the pass key

ssl/: $ rm server.pass.key

# now let's create the .csr file

ssl/: $ openssl req -new -key server.key -out server.csr
...

Country Name (2 letter code) [AU]:MX
State or Province Name (full name) [Some-State]:Michoacan
...
A challenge password []:
...

# now let's create the .crt file

ssl/: $ openssl x509 -req -sha256 -days 365 -in server.csr -
signkey server.key -out server.crt
```

Next we need to install **SPDY** with the following command:

```
cinema-catalog-service/: $ npm i -S spdy --silent
```

First let's create an `index.js` file in the `ssl/` folder, with the following code, here is where we load the key and cert files, this is probably one of the few cases when we can use `fs.readFileSync()` :

```
const fs = require('fs')

module.exports = {
  key: fs.readFileSync(`${__dirname}/server.key`),
  cert: fs.readFileSync(`${__dirname}/server.crt`)
}
```

Then we need to modify a couple of files, let's modify first the

`config.js` :

```
const dbSettings = { ... }

// The first modification is adding the ssl certificates to
// the
// serverSettings

const serverSettings = {
  port: process.env.PORT || 3000,
  ssl: require('./ssl')
}

module.exports = Object.assign({}, { dbSettings,
  serverSettings })
```

Next let's modify the `server.js` file like the following:

```
...
const spdy = require('spdy')
const api = require('../api/movies')
const start = (options) => {
  ...

  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!')
  })
  api(app, options)

  // here is where we made the modifications, we create a
  // spdy
  // server, then we pass the ssl certs, and the express app
```

```
const server = spdy.createServer(options.ssl, app)
  .listen(options.port, () => resolve(server))
})
}

module.exports = Object.assign({}, {start})
```

Finally let's modify the `index.js` main file:

```
'use strict'
const {EventEmitter} = require('events')
const server = require('./server/server')
const repository = require('./repository/repository')
const config = require('./config/')
const mediator = new EventEmitter()

...

mediator.on('db.ready', (db) => {
  let rep
  repository.connect(db)
    .then(repo => {
      console.log('Connected. Starting Server')
      rep = repo
      return server.start({
        port: config.serverSettings.port,
        // here we pass the ssl options to the server.js
file
        ssl: config.serverSettings.ssl,
        repo
      })
    })
    .then(app => { ... })
})

...
```

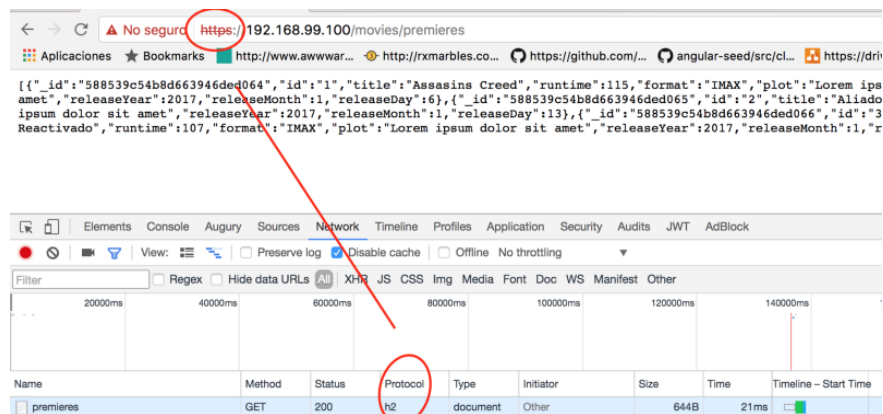
And now we need to rebuild our docker image with the following command:

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

and run our docker image `movies-service` with the following parameters:

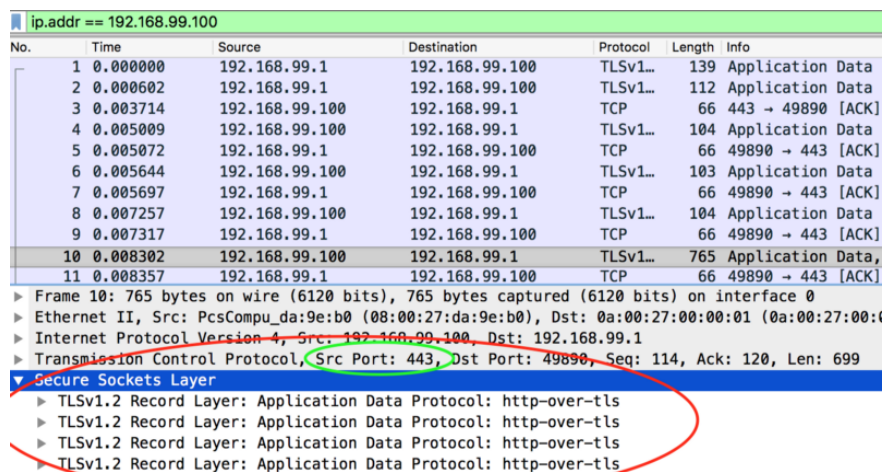
```
$ docker run --name movies-service -p 443:3000 -d movies-service
```

finally we test it with a chrome browser, and we can corroborate that our **HTTP/2** protocol is fully working.



chrome dev tools

And we can also corroborate with making some network capture with **wireshark** we can see that **ssl** is truly working.



Wireshark frame capture



## # Implementing JWT to the microservice

Another way for encrypting and securing our microservices communication is by using the `json web token` protocol, but we will see this implementation later on the series 🚂.

## # Building the microservice

Ok so now that we know how to implement the **HTTP/2** protocol, let's continue with building the **cinema catalog service**. We will use the same project structure from the **movies service**, so less talking 🗣️ and more coding 😊💻👤💻.

Before we start to design our API, this time we need to design our **Mongo Schemas** for our database, since we are going to use the following:

- Locations (countries, states and cities)
- Cinemas (cinemas, schedules, movies)

## ## MODEL DATA DESIGN

This article is really focused on creating a microservice, so i'm not going to spend ages on the “model data design” for our cinemas database, instead i will highlight the areas and takeaways.

```
# possible collections for the cinemas db.
```

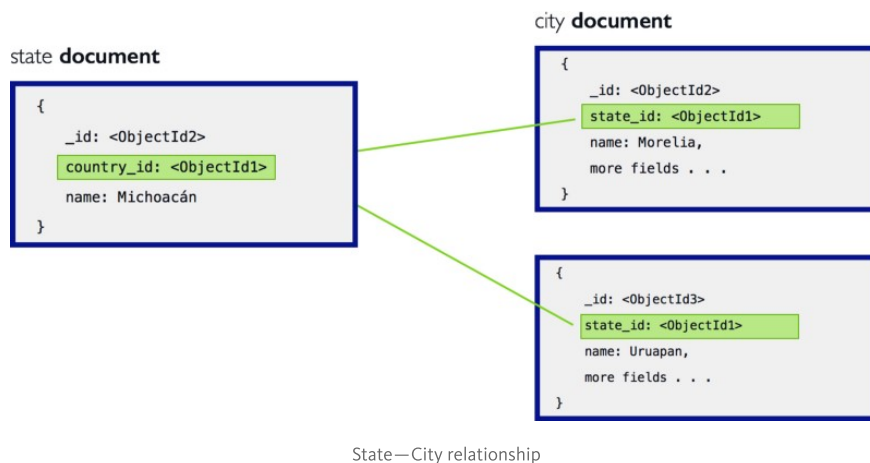
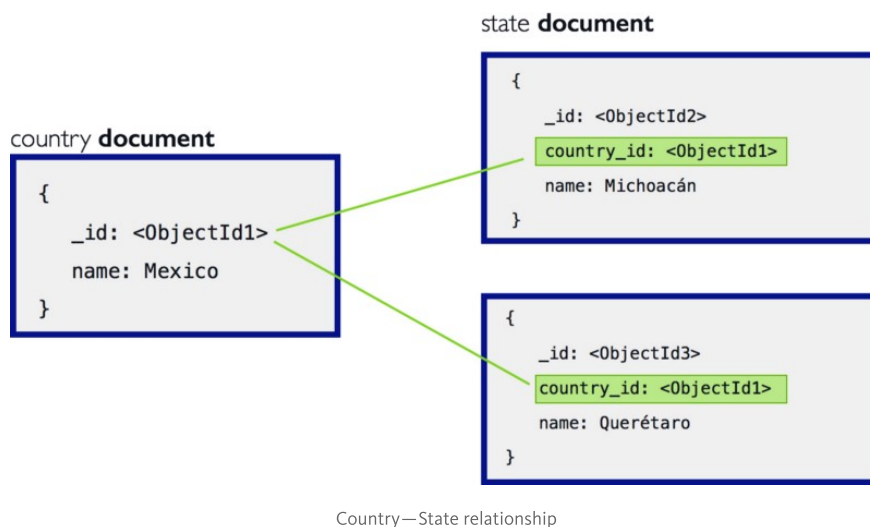
```
# For locations
```

```
- countries  
- states  
- cities
```

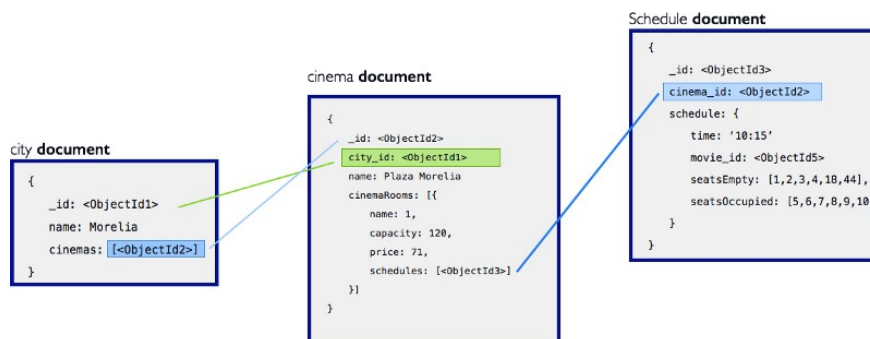
```
# For cinemas
```

```
- cinemas  
- cinemaRooms  
- schedules
```

So for our **Locations**, a country has to many states and the states has one country, so the first relationship is a **one to many**, but this also applies to, a state has many cities and a city has one state, so let's see how is our relationship example.

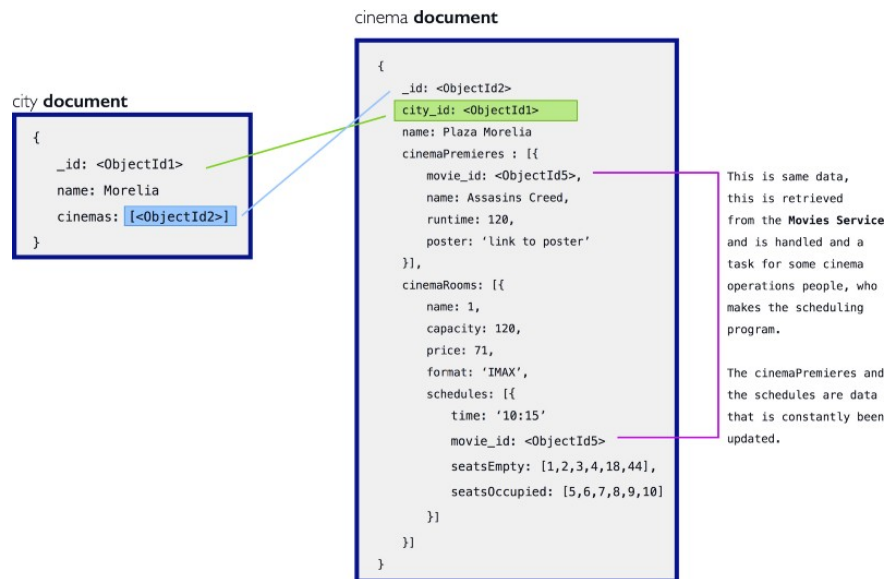


But this type of relationship is also possible for, a city has many cinemas a cinema belong to a city, another relationship we can see is that a cinema room has many schedules, a schedule belong to a cinema room, so let's see how are this relationships.



This kind of references in the diagram above may be useful, if the cinemas array or schedules array are growth limited. Let's suppose that one cinema room has maximum 5 schedules per day, so here we can embed the schedule document into the cinema document.

*Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.*  
—MongoDB Docs



So this is the final result for our database schemas design.

## # Import data to our database

I've already prepare some data example with schema design we saw above, the files are located at the github repo `cinema-catalog-service/src/mock` there 4 json files, so you can import it to the cinemas database, but first we need to know which databases server is the primary, so find out and execute the following commands:

```
# first we need to copy the files one by one or we can zip it and pass the zip file
```

```
$ docker cp countries.json mongoNodeContainer:/tmp
$ docker cp state.json mongoNodeContainer:/tmp
$ docker cp city.json mongoNodeContainer:/tmp
$ docker cp cinemas.json mongoNodeContainer:/tmp
```

Once we execute the commands above let's import it the database like the following:

```
$ docker exec mongoNode{number} bash -c 'mongoimport --db
cinemas --collection countries --file /tmp/countries.json --
jsonArray -u $MONGO_USER_ADMIN -p $MONGO_PASS_ADMIN --
authenticationDatabase "admin"'

$ docker exec mongoNode{number} bash -c 'mongoimport --db
cinemas --collection states --file /tmp/states.json --
jsonArray -u $MONGO_USER_ADMIN -p $MONGO_PASS_ADMIN --
authenticationDatabase "admin"'

$ docker exec mongoNode{number} bash -c 'mongoimport --db
cinemas --collection cities --file /tmp/cities.json --
jsonArray -u $MONGO_USER_ADMIN -p $MONGO_PASS_ADMIN --
authenticationDatabase "admin"'

$ docker exec mongoNode{number} bash -c 'mongoimport --db
cinemas --collection cinemas --file /tmp/cinemas.json --
jsonArray -u $MONGO_USER_ADMIN -p $MONGO_PASS_ADMIN --
authenticationDatabase "admin"'
```

Now we have our database schema designs ready, and our data ready to query, so we can now design our API for the **cinema catalog service**, one way of defining our routes are making some sentences, like the following:

- we need a city for displaying the available cinemas.
- we need the cinemas to display the movie premieres.
- we need the movie premieres and display the schedules.
- we need the schedules to see if there are seats available for booking.

Let's assume that other teams in the IT department of **Cinépolis** are making the other CRUD operations, and that our task is to make the "R" the read data, and let's assume that some **Cinépolis** cinema operations people have already made the scheduling for the cinemas, so our task is to retrieve those schedules.

What it concerns to the **cinema catalog service** is only about cinemas and schedules, no more, above we saw that we made **locations** collections, but that is the concern for another microservice, but we are dependent on the locations to be able to display the cinemas and the schedules.

Now that we have define our necessities we can build our RAML file, like the following:

```
1  ##RAML 1.0
2  title: Cinema Catalog Service
3  version: v1
4  baseUri: /
5  uses:
6    object: types.raml
7    stack: ../movies-service/api.raml
8
9  types:
10   Cinemas: object.Cinema []
11   Movies: stack.MoviePremieres
12   Schedules: object.Schedule []
13
14  traits:
15   FilterByLocation:
16     queryParameters:
17       city:
18         type: string
19
20  resourceTypes:
21   GET:
22     get:
23       responses:
24         200:
25           body:
26             application/json:
27               type: <<item>>
```

We have fulfill 3 of the 4 sentences we made above, the sentence number 4 is for booking in a cinema, but that my friends belongs to the

**booking service** who has that responsibility among other things, so stay tuned the “Build a NodeJs cinema microservice—series”.

Now we can continue developing our **NodeJS** API for **cinema catalog service**, the structure and configuration are almost the same as the **movies service**, so i will start showing you the `repository.js` for this API.

```
1 // more code above
2
3 const getCinemasByCity = (cityId) => {
4   return new Promise((resolve, reject) => {
5     const cinemas = []
6     const query = {city_id: cityId}
7     const projection = {_id: 1, name: 1}
8     // example of making a find query to mongoDB,
9     // passign a query and projection objects.
10    const cursor = db.collection('cinemas').find(query, p
11    const addCinema = (cinema) => {
12      cinemas.push(cinema)
13    }
14    const sendCinemas = (err) => {
15      if (err) {
16        reject(new Error('An error occured fetching cinem
17      }
18      resolve(cinemas)
19    }
20    cursor.forEach(addCinema, sendCinemas)
21  })
22 }
23
24 const getCinemaById = (cinemaId) => {
25   return new Promise((resolve, reject) => {
26     const query = {_id: new ObjectId(cinemaId)}
27     const projection = {_id: 1, name: 1, cinemaPremieres:
28     const response = (err, cinema) => {
29       if (err) {
30         reject(new Error('An error occuered retrieving a
31       }
32       resolve(cinema)
33     }
34     // example of using findOne method from mongodb,
35     // we do this because we only need one record.
36     db.collection('cinemas').findOne(query, projection, r
37   })
38 }
39
40 const getCinemaScheduleByMovie = (options) => {
41   return new Promise((resolve, reject) => {
```

```

42     const match = { $match: {
43       'city_id': options.cityId,
44       'cinemaRooms.schedules.movie_id': options.movieId
45     }
46     , { $project: {
47       'name': 1,
48     }
49   }
50 }
51

```

To see the complete `repository.js` file, you can go and check it at the “github repo branch *step-2*”.

Here we are defining 3 functions:

- **getCinemasByCity:** This function will get us all the cinemas available in the city, we pass the `city_id` to find the cinemas, the result of this function help us to call the next one.
- **getCinemaById:** This function will retrieve the name, id, and premiere movies available, by querying it by `cinema_id`, the result of this function will help us to finally get the schedules.
- **getCinemaScheduleByMovie:** This function will give us all the schedules for a movie available on all the cinemas in a city.

There could be another function or we can modify the **getCinemaById**, to display the schedules of a current cinema, this could be a good challenge for you, if you want to practice, this can't be so difficult, because i have already provide you all the need information.

The next file to check is our API file `cinemas-catalog.js`.



```
1  'use strict'
2  const status = require('http-status')
3
4  module.exports = (app, options) => {
5    const {repo} = options
6
7    app.get('/cinemas', (req, res, next) => {
8      repo.getCinemasByCity(req.query.cityId)
9        .then(cinemas => {
10          res.status(status.OK).json(cinemas)
11        })
12        .catch(next)
13    })
14
15    app.get('/cinemas/:cinemaId', (req, res, next) => {
16      repo.getCinemaById(req.params.cinemaId)
17        .then(cinema => {
18          res.status(status.OK).json(cinema)
19        })
20        .catch(next)
21    })
22  }
```

As you can see, here we implement our endpoints, as we define it in our **RAML** file, and we call the `repository.js` functions according to the route.

In our first route, we are using `req.query.cityId` to get the value and query our database to get the cinemas by city `city_id`, and in the other routes we use `req.params` to get the value of the `cinemaId` and `movieId` to be able to query the `schedules` 😊.

Finally we can see the `cinema-catalog.spec.js` file for testing:

```
1  /* eslint-env mocha */
2  const request = require('supertest')
3  const server = require('../server/server')
4  process.env.NODE = 'test'
5
6  describe('Movies API', () => {
7    let app = null
8    const testCinemasCity = [{
9      '_id': '588ac3a02d029a6d15d0b5c4',
10     'name': 'Plaza Morelia'
11   }, {
12     '_id': '588ac3a02d029a6d15d0b5c5',
13     'name': 'Las Americas'
14   }]
15
16   const testCinemaId = {
17     '_id': '588ac3a02d029a6d15d0b5c4',
18     'name': 'Plaza Morelia',
19     'cinemaPremieres': [
20       {
21         'id': '1',
22         'title': 'Assasins Creed',
23         'runtime': 115,
24         'plot': 'Lorem ipsum dolor sit amet',
25         'poster': 'link to poster...'
26       },
27       {
28         'id': '2',
29         'title': 'Aliados',
30         'runtime': 124,
31         'plot': 'Lorem ipsum dolor sit amet',
32         'poster': 'link to poster...'
33       },
34       {
35         'id': '3',
36         'title': 'xXx: Reactivado',
37         'runtime': 107,
38         'plot': 'Lorem ipsum dolor sit amet',
39         'poster': 'link to poster...'
40       }
41     ]
42   }
```

```
42
43
44
45
46
47
48
49     }
50
51     's\
52   }, {
53     'room
54     'schedu.
55   }]
56 }, {
57   '_id': 'Las Am
58   'schedules': [ {
59     'room': 2.0,
60     'schedules': [ '3.
61   }, {
62     'room': 1.0,
63     'schedules': [ '12:15',
64   }]
65   }]
66
67   let testRepo = {
68     getCinemasByCity (location) {
69       console.log(location)
70       return Promise.resolve(testCinemas\
71     },
72     getCinemaById (cinemaId) {
73       console.log(cinemaId)
74       return Promise.resolve(testCinemaId)
75     },
76     getCinemaScheduleByMovie (cinemaId, movieId) {
77       console.log(cinemaId, movieId)
78       return Promise.resolve(testSchedulesMovie)
79     }
80   }
```

Finally we can build our docker image `cinema-catalog-service` and run it inside our container, we will use the same `dockerfile` from the **movies service**, to make this process a little bit more automated let's create a bash script `start-service.sh` for this task like the following:

```
#!/usr/bin/env bash

eval `docker-machine env manager1`

docker build -t catalog-service .

docker run --name catalog-service -p 3000:3000 --env-file
env -d catalog-service
```

As we start making more services, we need to be careful with the ports available to our services, so this time, i will use the port 3000, and additionally i will use an `env file` to start using the `process.env` configurations inside our **NodeJS** service, and our env file will look something like this:

```
DB=cinemas
DB_USER=cristian
DB_PASS=cristianPassword2017
DB_REPLS=rs1
DB_SERVERS='192.168.99.100:27017 192.168.99.101:27017
192.168.99.102:27017'
PORT=3000
```

This is consider as best practice, outside in the world of devOps.

Finally we need to run our small bash script like the following:

```
# execute our script
$ bash < start-service.sh

# check running docker containes
$ docker ps
```

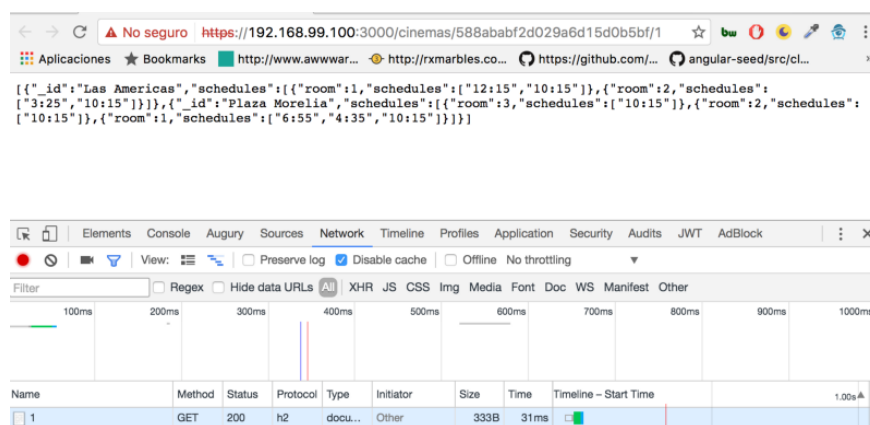
and we need to have something like this:

```
MacBook-Pro-de-Cristian:cinema-catalog-service Cramirez$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ba2e9550719	catalog-service	"npm start"	11 seconds ago	Up 10 seconds	0.0.0.0:3000->3000/tcp	catalog-service
4fda0c47569f	movies-service	"npm start"	5 days ago	Up 47 minutes	0.0.0.0:443->3000/tcp	movies-service
fd3038840378	mongo	"/entrypoint.sh --,..."	6 days ago	Up 42 hours	0.0.0.0:27017->27017/tcp	mongoNode1

docker status

and we can test our service in a chrome-browser and verify that our HTTP/2 protocol is working, and that our service is working.



Chrome browser—test

And just for fun we can make a stress test, using **JMeter**, the stress test file is also at the `integration-test/` folder at the github repo.

**Petición HTTP**

Nombre: Fetch schedules by movie

Comentarios

Basic Advanced

Servidor Web

Nombre de Servidor o IP: Puerto: Timeout (milisegundos): Conexión:

Petición HTTP

Implementación HTTP: Protocolo: Método: GET Codificación:

Ruta: /cinemas/588ababf2d029a6d15d0b5bf/\${id}

☐ Redirigir Automáticamente ☒ Seguir Redirecciones ☒ Utilizar KeepAlive ☐ Usar 'multipart/form-data' para HTTP POST

Parameters Body Data Files Upload

Enviar Parámetros Con la Petición:

Nombre:	Valor
id	1
id	2
id	3

Detail Añadir Add from Clipboard Borrar Up

Servidor Proxy

Nombre de Servidor o IP: Puerto: Nombre de Usuario:

JMeter capture

```

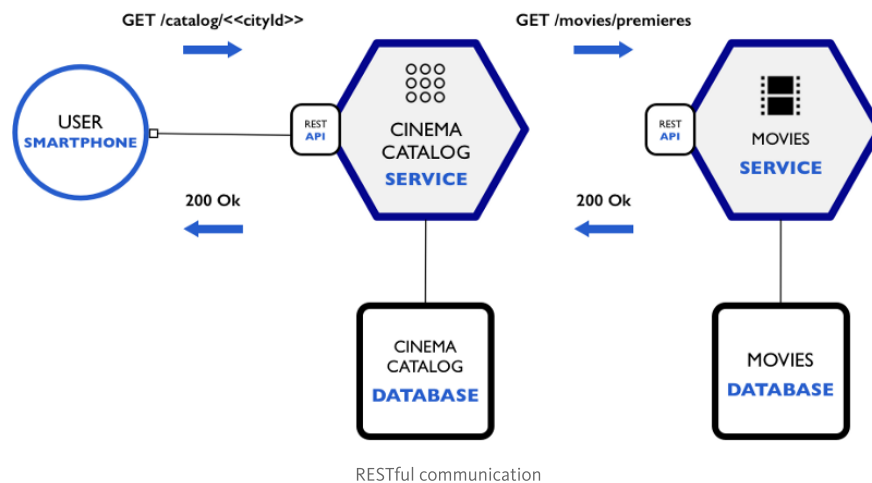
100 request(app)
101 .get('/cinemas?cityId=${location.city}')
102 .expect((res) => {
103   res.body.should.containEql(testCinemasCity[0])
104   res.body.should.containEql(testCinemasCity[1])
105 })
106 .expect(200, done)
107 })
108
109 it('can get movie premiers by cinema', (done) => {
110   request(app)
111     .get('/cinemas/588ac3a02d029a6d15d0b5c4')
112     .expect('get(key: ?)')
113     .expect(res => {
114       res.body.should.containEql(testCinemaId)
115     })
116 })

```

stress test example

## # Time for a recap

What we have done...



We've completed the microservices of this diagram, you may be saying that we aren't using the **movies service**, inside our **cinema catalog service** and yes that's correct, what we have made until know is only **GET** request from our services, and to make use of our **movies service** inside the **cinema catalog service** is by making **POST** request fulfilling the cinema premieres stack movies to be able to make the schedules, but since our task is to make the **R** from **CRUD** operations in our team so that's why we haven't seen that interaction, but later on the series we will make more **CRUD** operations between microservices be patient, stay curious :D.

So what we make in this chapter ¿ 🤖 ?, we learned about **HTTP/2** protocol and we saw how to implement it in our **microservices**. We also see how to **design a MongoDB schema**, we didn't go deeper but i highlight it to give a better picture on what is happening in the **cinema catalog service**, then we **design the API** with **RAML**, and we start building our API service, then we made the corresponding **unit test**, finally we compose everything to have our **microservice complete**.

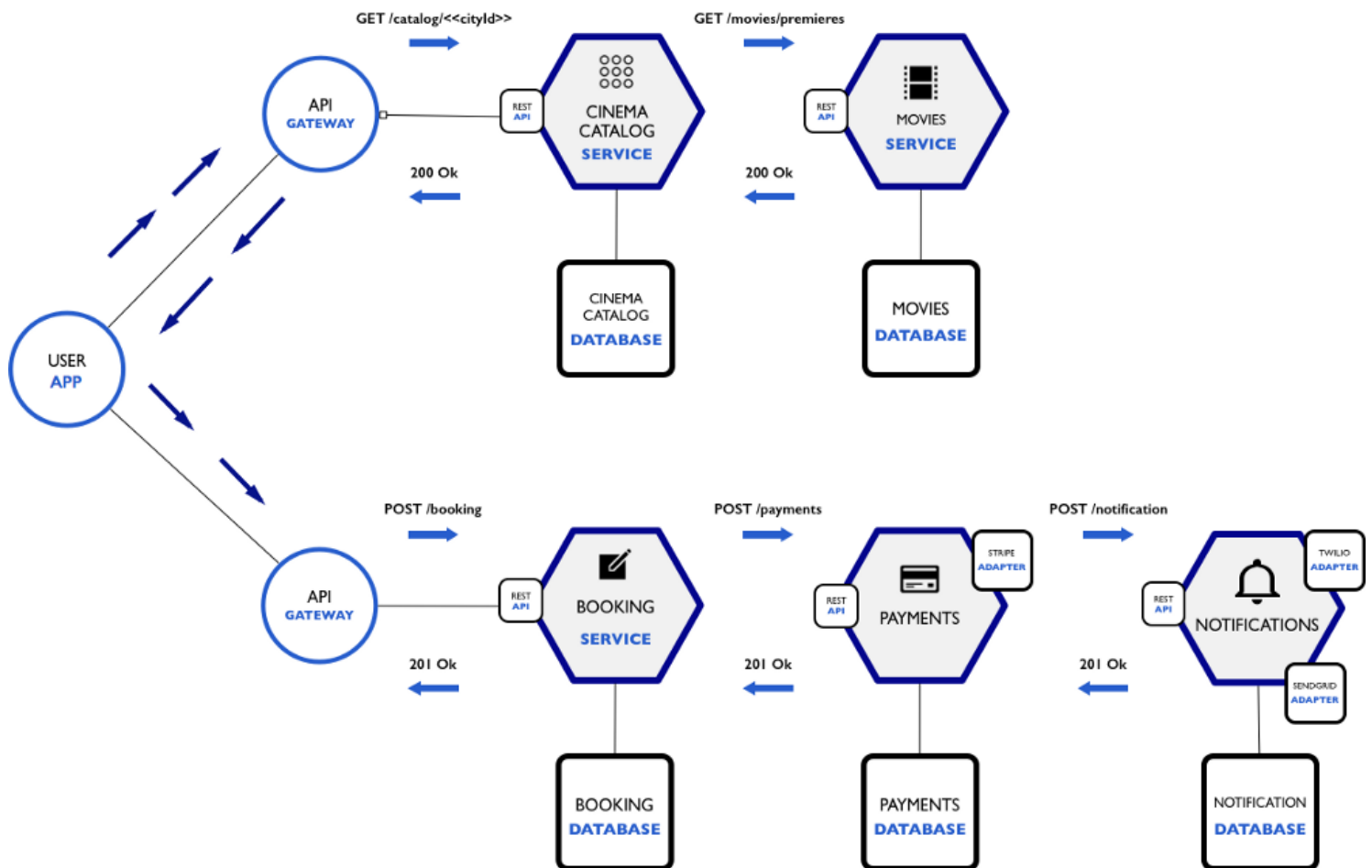
Then we use the same **dockerfile** from our previous **microservice**, and we made a script to automate this process, what we also made is that we introduce how to use and **env file** and use that **environment variables** inside the **Docker** container, with all the setup ready, i show you a brief pic of **JMeter** making a **stress test** complementing the **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.

## # Coming Next

Now that we have finished our first diagram, we are going to develop our second diagram, the **booking service diagram** and ... 🧐👤💻



Microservices Diagram

. . .

## # Complete code at Github

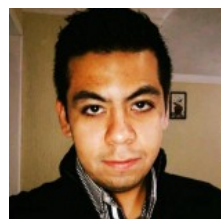
You can check the complete code of the article at the following link.



Criztian/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)

## # References

- [Easy HTTP/2 Server with Node.js and Express.js](#)
- [HTTP/2: the good, the bad and the ugly](#)

. . .

Let me remember you, this article is part of “***Build a NodeJS cinema microservice***” series so, next week i will publish another chapter of this series.



If you want to keep going on the series there is the link below of the third chapter:

Build a NodeJS cinema booking microservice and deploying it with docker (part 3)

Hello community this is the 🏠 third article from the series “Build a NodeJS cinema microservice”. This series of...

medium.com



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 , or just comment below .

Until next time     

You can follow me at twitter @cramirez\_92

[https://twitter.com/cramirez\\_92](https://twitter.com/cramirez_92)

