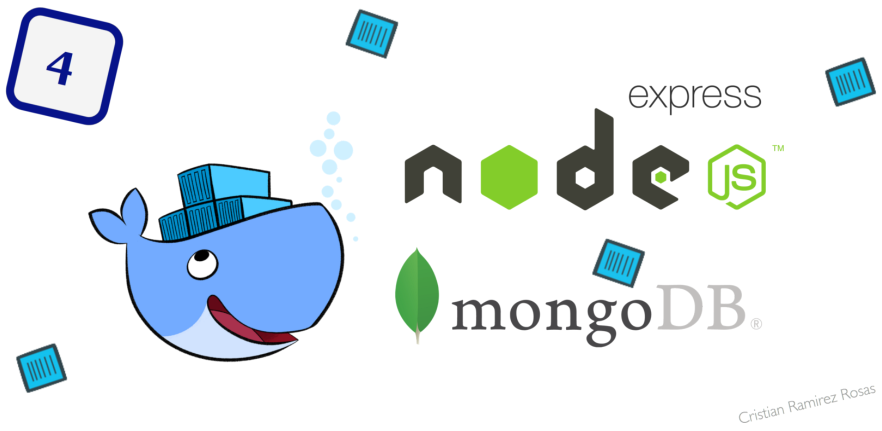
Cristian Ramirez Follow
FullStack MEAN Engineer with some Docker knowledge Linkedin: https://www.linkedin.com/in/cristian-r...
Feb 13 · 15 min read

Build a NodeJS cinema API Gateway and deploying it to Docker (part 4)



Images founded on google—cover made by me :D

> This is the 🏛 fourth article from the series "Build a NodeJS cinema microservice". This series of articles demonstrates how to ***design, build, and deploy microservices*** with expressjs using ES6, ¿ES7 …8?, connected to a MongoDB Replica Set, also this articles demonstrate how to deploy it into a docker container and simulate how this microservices will run in a cloud environment.

## A quick recap from our previous chapters

- The first article introduces the **Microservices Architecture pattern** and discusses the **benefits** and **drawbacks** of using microservices.
- The second article we talked about **microservices security** with the **HTTP/2 protocol.**
- The third article in the series describe **different** aspects of **communication** within a **microservices** architecture, and we explain about **design patterns** in **NodeJS** like **Dependency Injection, Inversion of Control** and **SOLID principles.**
- we have made already 3 API's, and run it into a **Docker Container**

- we have made **unit, integration** and **stress testing.**

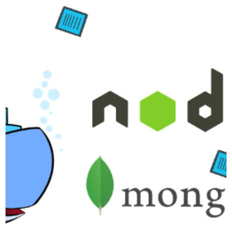
if you haven't read the previous chapters, you're missing some great stuff 🤘, i will put the links below, so you can give it a look 👀.

# Build the cinema microservice (part 1)

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

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

medium.com

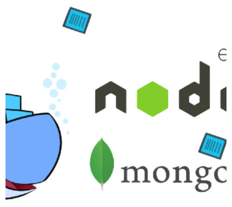


# Build the cinema microservice (part 2)

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

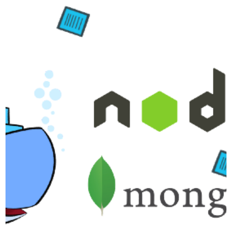


# Build the cinema booking microservice (part 3)

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

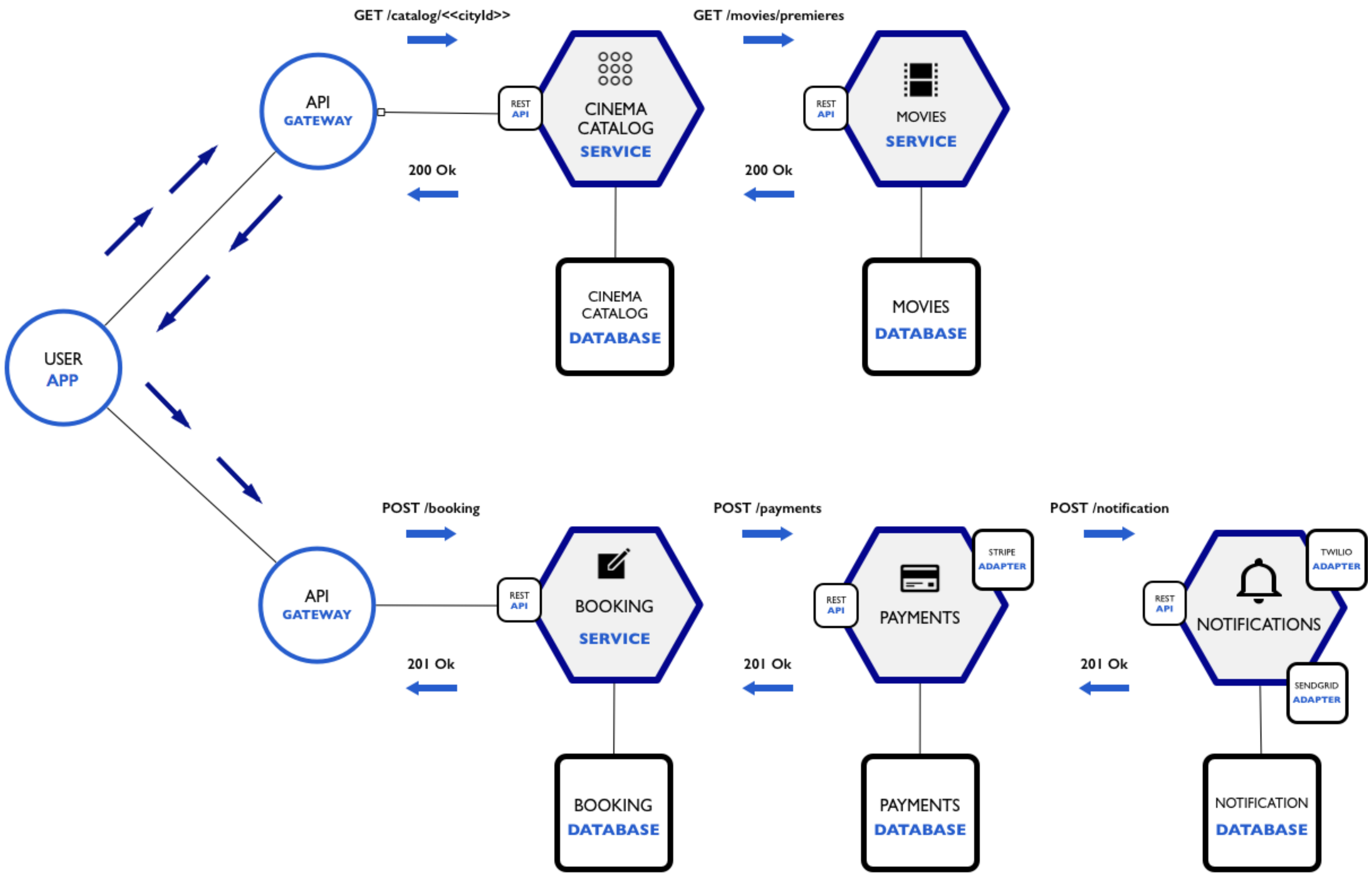
Hello community this is the 🏰 third article from the series "Build a NodeJS cinema microservice". This series of…

medium.com



. . .

Ok so at this point we are going to finish the next diagram:



Sub cinema microservice architecture

What is left to build for us is the **payments service** and the **notifications service,** that we will develop very fast this time to focus on something that we haven't talked about in this architecture an that has been present since the beginning, the **API Gateway**, so stay with me and let's start making some fun stuff 😎.

What we are going to use for this article is:

- NodeJS version 7.5.0 (installed locally)
- MongoDB 3.4.1
- Docker for Mac 1.13.0 (installed, 1.13.1 break things)

Prerequisites to following up the article:

- Have completed the examples from the last chapter.

If you haven't, i have uploaded a github repository, so you can be up to date, repo link at branch **step-3.**

Payment and Notification Services

Since this article is for building the **API Gateway**, this time i wont spent to much time in describing the next services, i will only highlight the interesting parts. For this services, we will continue using the same project and app structure, there will be a slightly change on this ones, so let's see how are this services composed 👀.

## Payment Service

To have a payment service working, as you may know there are a bunch of libraries for node to make credit card charges, in this time i will use a library called **stripe**, but before building our **payment service**, you should go to the **stripe website,** and create an account to be able to use this library, because we will need a stripe token for making payment tests.

```
# Then we need to install stripe in our project

cinema-microservice/payment-service $ npm i -S stripe --
silent
```

So how do we use stripe, first let's register our **stripe dependency** in our `di.js` file:

```
const { createContainer, asValue } = require('awilix')
const stripe = require('stripe')

// here we include the stripeSettings
function initDI ({serverSettings, dbSettings, database, mod
  mediator.once('init', () => {
    mediator.on('db.ready', (db) => {
      const container = createContainer()

      container.register({
        database: asValue(db),
        validate: asValue(models.validate),
        ObjectID: asValue(database.ObjectID),
        serverSettings: asValue(serverSettings),
        // and here we register our stripe module
```

Next up, we will see how is our `api/payment.js` file:

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

module.exports = ({repo}, app) => {
  app.post('/payment/makePurchase', (req, res, next) => {
    const {validate} = req.container.cradle

    validate(req.body.paymentOrder, 'payment')
      .then(payment => {
        return repo.registerPurchase(payment)
      })
      .then(paid => {
        res.status(status.OK).json({paid})
      })
      .catch(next)
  })
```

and finally let’s check our `repository.js` file:

```

// this the function that makes the charge, when it's don
// returns the charge object returned by stripe

const makePurchase = (payment) => {
  return new Promise((resolve, reject) => {
    // here we retrieve or stripe dependecy
    const {stripe} = container.cradle

    // we create the charge
    stripe.charges.create({
      amount: Math.ceil(payment.amount * 100),
      currency: payment.currency,
      source: {
        number: payment.number,
        cvc: payment.cvc,
        exp_month: payment.exp_month,
        exp_year: payment.exp_year
      },
      description: payment.description
    }, (err, charge) => {
      if (err && err.type === 'StripeCardError') {
        reject(new Error('An error occuered procesing pay
      } else {
        const paid = Object.assign({}, {user: payment.use
        resolve(paid)
      }
    })
  })
}

// this the function that our API calls first
const registerPurchase = (payment) => {
  return new Promise((resolve, reject) => {

    // and here we call the function to execute stripe
```

I want to highlight something at the `repository.js` here we are using some guidelines as good developers as we are, in the functions `registerPurchase()` and `makePurchase()` , the guidelines are:

> **Do One Thing (DOT)**
> *"Each function should do only one thing, and do that one thing as well as it can."*

> **Less Is More**
> *"Functions should be as short as possible: If they run much longer, consider breaking out subtasks and data into separate functions and objects."*

> *— from book Programming Apps with Javascript, Eric Elliott*

Notification Service

Ok so now, in our **notification service,** again there are some very good libraries for sending emails, sms , mms, etc., you can give it a look to the **twilio** or **sendgrid,** to dive deeper on notification services, but this time i will show you a very simple service using, **nodemailer.**

```
# So we need to install nodemailer in our project

notification-service$ npm i -S nodemailer nodemailer-smtp-
transport --silent
```

Now let's see how is our js files, first will be our `api/notification.js` then our `repository.js`

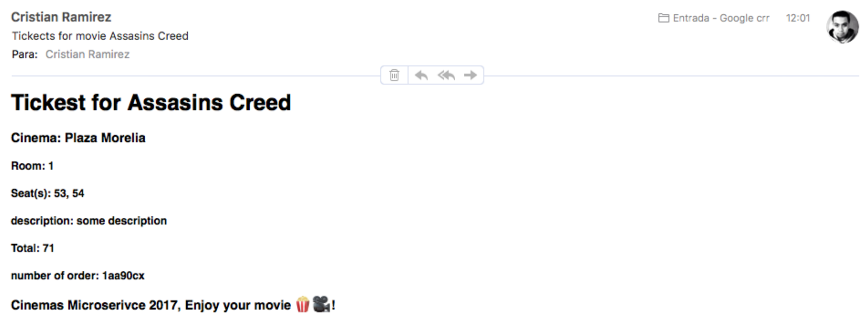
```
module.exports = ({repo}, app) => {
  // this our endpoint where is going to validate our email
  app.post('/notifiaction/sendEmail', (req, res, next) => {
    const {validate} = req.container.cradle

    validate(req.body.payload, 'notification')
      .then(payload => {
        return repo.sendEmail(payload)
      })
      .then(ok => {
        res.status(status.OK).json({msg: 'ok'})
      })
      .catch(next)
  })
}
```

notification.api.js hosted with ♡ by GitHub view raw

```
const sendEmail = (payload) => {
    return new Promise((resolve, reject) => {
      const {smtpSettings, smtpTransport, nodemailer} = con

      const transporter = nodemailer.createTransport(
        smtpTransport({
          service: smtpSettings.service,
          auth: {
            user: smtpSettings.user,
            pass: smtpSettings.pass
          }
        }))

      const mailOptions = {
        from: '"Do Not Reply, Cinemas Company 👥" <no-repl
        to: `${payload.user.email}`,
        subject: `Tickects for movie ${payload.movie.title}
        html: `
            <h1>Tickest for ${payload.movie.title}</h1>

```

To see the full configuration as always you are welcome to check the **cinemas microservice repo** at github at branch **step-4.**

If we have setup everything ok, and run the integration test, our **notification-service** can send an emails like the image below:


Cristian Ramirez
Tickects for movie Assasins Creed
Para: Cristian Ramirez
Entrada - Google crr 12:01

Tickest for Assasins Creed

Cinema: Plaza Morelia

Room: 1

Seat(s): 53, 54

description: some description

Total: 71

number of order: 1aa90cx

Cinemas Microserivce 2017, Enjoy your movie 🍿🎥!

# Conclusion Payment and Notification services

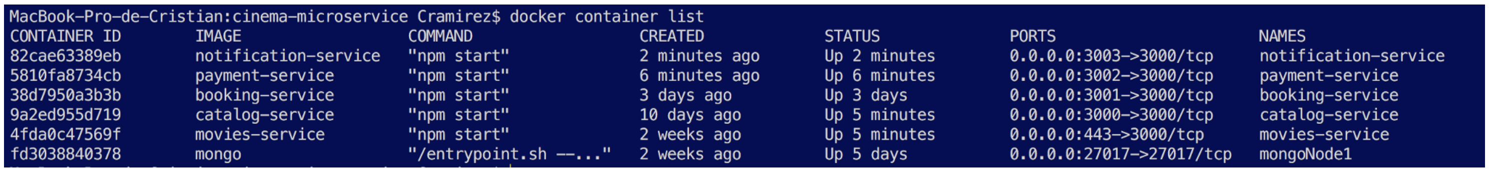
If you think that i went to fast with this two services as always you're welcome to send me a tweet or just put a comment below 🤓 so we can discuss in more detail, what is happening here.

There is something important that i didn't mention, is that when we are treating with money (credit cards, accounts), **we need to ensure that our data is encrypted** just to **add** another layer of **security**, and inside the **payment-service** we need to decrypt the user information, to proceed with **stripe** checkout**.**

Finally at the repo and on each service there is our `start_service.sh` bash file, if we execute it, we will put our service into a docker container.

```
$ bash < start_service.sh
```

If we run it, in our docker-machine **Manager 1**, we should have something like this:



```
MacBook-Pro-de-Cristian:cinema-microservice Cramirez$ docker container list
CONTAINER ID        IMAGE                  COMMAND                  CREATED           STATUS           PORTS                      NAMES
82cae63389eb        notification-service   "npm start"              2 minutes ago     Up 2 minutes     0.0.0.0:3003->3000/tcp     notification-service
5810fa8734cb        payment-service        "npm start"              6 minutes ago     Up 6 minutes     0.0.0.0:3002->3000/tcp     payment-service
38d7950a3b3b        booking-service        "npm start"              3 days ago        Up 3 days        0.0.0.0:3001->3000/tcp     booking-service
9a2ed955d719        catalog-service        "npm start"              10 days ago       Up 5 minutes     0.0.0.0:3000->3000/tcp     catalog-service
4fda0c47569f        movies-service         "npm start"              2 weeks ago       Up 5 minutes     0.0.0.0:443->3000/tcp      movies-service
fd3038840378        mongo                  "/entrypoint.sh --..."   2 weeks ago       Up 5 days        0.0.0.0:27017->27017/tcp   mongoNode1
```

docker container list === docker ps :D

# API Gateway

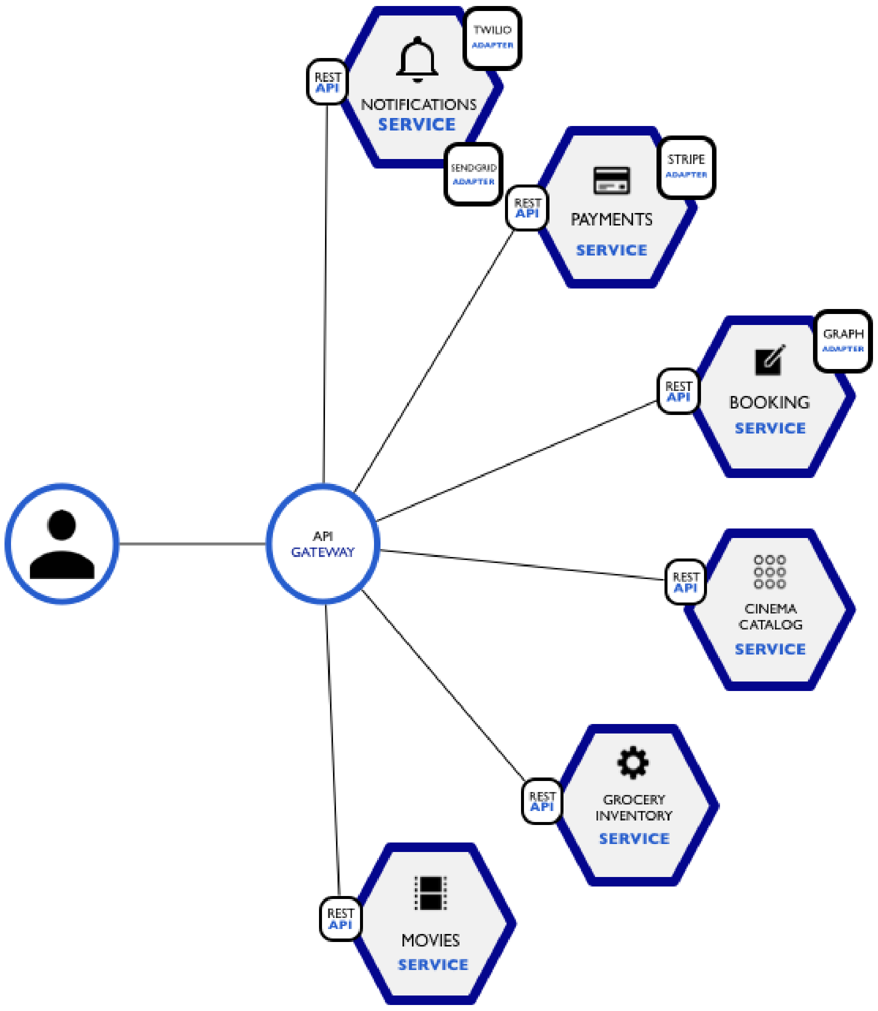
> One of the **biggest changes** in application design and architecture when moving to **microservices** is using the **network to communicate between functional components of the application**. In monolithic apps, application components communicate in memory. In a microservices app, that communication happens over the network, so network design and implementation become critically important.— *@Nginx MRA document*

But first ¿ what is an API Gateway ? ¿ and DO WE NEED IT ?

> *An API Gateway is a server that is the single entry point into the system. It is similar to the Facade pattern from object oriented design.—Chris Richardson*

The **API Gateway encapsulates** the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

The following diagram show us how can an **API Gateway** fit into our cinema architecture:



Example of the cinema microservice API Gateway

The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice.

¿ Why do we need it ? (Benefits & Drawbacks)

Because using an API Gateway encapsulates our internal structure of the application, where this reduces the number of round trips between the client and application, and also simplifies our code. Implementing an **API Gateway** handles requests in one of two ways. Some requests are simply **proxied/routed to the appropriate service**. It handles other requests by fanning out to multiple services.

But the API Gateway also has some drawbacks. It is yet another "**highly available component that must be developed, deployed, and managed**". There is also **a risk that the API Gateway becomes a development bottleneck**, so as good developers we must update the API Gateway, it is important that the process for updating the API Gateway be as lightweight as possible.

> ***Cross-cutting concerns*** *must be implemented in a way such that microservices need not deal with details regarding problems outside their specific scope. For instance, authentication can be implemented as part of any API gateway or proxy.—@authO*

A list of common concerns handled by API gateways:

- Authentication
- Transport security
- Load-balancing
- Request dispatching (including fault tolerance and service discovery)
- Dependency resolution
- Transport transformations

# Building the microservice

Ok so now that we are soaked and we have read 📖 on what does an **API Gateway**, let's start building our **cinema microservice API Gateway** 👩🔬👨🔬

Until know we have been building our microservices with the **http/2 protocol,** and to get up and running this protocol, we need to satisfy some rules, and we would need to create some **ssl certificates,** and for simplicity we create **self-signed-certificates**, but this will become an issue to proxy our routes in our **api-gateway.** There's a recommendation from the **http-proxy** nodejs module, that tell us the following:

> *You can activate the validation of a secure SSL certificate to the target connection (avoid self signed certs), just set* `secure: true` *in the options.*

we can not make the proxy to self-signed-certificates, we will need to rollback our microservices **http/2** protocol to the previous one, but not everything is bad news, as the **api-gateway** has many concerns, we will use now the **http/2 protocol**, only in the **api-gateway**, and this how we activate the **transport security** concern to our gateway.

So let's get started, and let's get our hands 🖐 dirty with some `< coding />` 👩💻👨💻, but first we need to make some review and give a look to our microservice definitions, and that's in our `raml files` then we need to check our `api files` and corroborate that our `api routes` are well defined. This is important because, those routes are the ones that we are going to proxy.

So let's begin with the `raml files`

```
#%RAML 1.0
title: Booking Service
version: v1
baseUri: /booking

/:
  type:   { POST: {item : Booking, item2 : User, item3: Tic

  /verify/{orderId}:
    type:  { GET: {item : Ticket} }
```

booking.raml hosted with ♡ by GitHub view raw

```
#%RAML 1.0
title: Cinema Catalog Service
version: v1
baseUri: /cinemas

/:
  type:  { GET: {item : Cinemas } }

  /{cinema_id}:
    type:  { GET: {item : Movies } }

  /{city_id}/{movie_id}:
      type:  { GET: {item : Schedules } }
```

catalog.raml hosted with ♡ by GitHub view raw

```
#%RAML 1.0
title: Movies Service
version: v1
baseUri: /movies

/:
  /premieres:
    type:  { GET: {item : MoviePremieres } }

  /{id}:
    type:  { GET: {item : Movie } }
```

So now that we have our endpoints well defined, it's time to refactor our **microservices**, with the http protocol only and check our **api endpoints** in code.

Now behold 🤴👸 the most interesting part is yet to come 😁

Since we have been dockerizing all of our microservices, `docker-machine manger1` has the knowledge of what containers are running, and if we issue the following command:

```
$ docker inspect <containerName | containerId >
```

This will return us, information about what that container does, if only if we specify that information at the container creation moment, if you have been following this series, you have noticed that each service has a `start-service.sh` so now we need to stop and remove our services, why because we are going to add a new flag to the `docker run command.` the **label flag**, like the following:

```
$ docker run
 --name {service}
 -l=apiRoute='{route}'
 -p {host-port}:{container-port}
 --env-file env
 -d {service}
```

Let's see why do we need this flag, and for that my friends let's go deep inside the **api-gateway** source code, so the first file we are going to see 👀 is the `api-gateway/config.js` **:**

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

const serverSettings = {
  port: process.env.PORT || 8080,
  ssl: require('./ssl')
}

const machine = process.env.DOCKER_HOST
const tls = process.env.DOCKER_TLS_VERIFY
const certDir = process.env.DOCKER_CERT_PATH

if (!machine) {
  throw new Error('You must set the DOCKER_HOST environment
}
if (tls === 1) {
  throw new Error('When using DOCKER_TLS_VERIFY=1 you must
}
if (!certDir) {
  throw new Error('You must set the DOCKER_CERT_PATH enviro
}

const dockerSettings = {
```

Here we are setting the **docker-settings** variable, because we are going to talk with our `manager1 docker-machine` but to be able to talk with that machine we need to set our environment correctly to the **manger1 docker-machine**, and we can do that in our terminal exectuing the following command:

```
$ eval `docker-machine env manager1
```

Once set our environment we are going to **connect** to our **docker-machie** directly from **nodejs**, ¿ why ? because we are getting the information of our running containers and be able to correctly **proxy the request** from our **api-gateway** to the microservices.

So how do we connect to our **docker-machine from nodejs**, well first we need to install a nodejs module called `dockerode` to our project like the following:

```
$ npm i -S dockerode --silent
```

Before we continue viewing the code of the **api-gateway,** first of all let's figure out what is a **Proxy.**

> *In computer networks, a **proxy server** is a server that acts as an intermediary for requests from clients seeking resources from other servers.—wikipedia*

Ok so now let's see what is an **ES6 proxy.**

> *Proxies are an interesting and powerful feature coming in ES6 that act as intermediaries between API consumers and objects. In a nutshell, you can use a `Proxy` to determine the desired behavior whenever the properties of an underlying `target` object are accessed. A `handler` object can be used to configure traps for your `Proxy`, which define and restrict how the underlying object is accessed—from book Practical ES6, author: Nicolás Bevacqua*

So now that we know what a proxy means either in computer network and as ES6 objects let's see the **`docker.js`**

```
'use strict'
const Docker = require('dockerode')

const discoverRoutes = (container) => {
  return new Promise((resolve, reject) => {
    // here we retrieve our dockerSettings
    const dockerSettings = container.resolve('dockerSetting

    // we instatiate our docker object, that will communica
    const docker = new Docker(dockerSettings)

    // function to avoid registering our database route and
    const avoidContainers = (name) => {
      if (/mongo/.test(name) || /api/.test(name)) {
        return false
      }
      return true
    }

    // here we register our routes in our ES6 proxy object
    const addRoute = (routes, details) => {
      routes[details.Id] = {
        id: details.Id,
        name: details.Names[0].split('').splice(1).join('')
        route: details.Labels.apiRoute,
        target: getUpstreamUrl(details)
      }
    }

    // we generate the container url to be proxy
    const getUpstreamUrl = (containerDetails) => {
      const {PublicPort} = containerDetails.Ports[0]
      return `http://${dockerSettings.host}:${PublicPort}`
    }

    // here we list the our running containers
    docker.listContainers((err, containers) => {
      if (err) {
        reject(new Error('an error occured listing containe
      }

```

```
        const routes = new Proxy({}, {
```

In our `docker.js` file, is happening a lot of magic 🔮✨ so let's see what is going on.

So first, we instantiate our `docker` object to be able to communicate to our **docker-machine**, then we create our `proxy routes object` to store all the routes that our `docker` object discover and list it, then we loop through the discovered containers and we register our `route` object with the container details, and here is why we need to add the **label flag when we start a docker container,** because, it can provide us more information and this os one way of adding information to a container, and so for us, it help us to know what is the purpose of the container.

So finally we resolve or routes object to be available in the `server.js`. You may be asking why i am using **ES6 Proxy** object, well it's because i thought that it could be a good example of using ES6 proxy object 🤓 😜 (because a i haven't seen to many examples of ES6 proxy), we could use any kind of object to store our routes, but proxy object's can help us to do much more things, we can see the proxies like middleware in javascript objects, but this is out of scope to this article.

So now let's look to our `server.js` file and see how do we implement our routes:

```
'use strict'
const express = require('express')
const proxy = require('http-proxy-middleware')
const spdy = require('spdy')
const morgan = require('morgan')
const helmet = require('helmet')
const cors = require('cors')
const status = require('http-status')

const start = (container) => {
  return new Promise((resolve, reject) => {
    const {port, ssl} = container.resolve('serverSettings')
    const routes = container.resolve('routes')

    if (!routes) {
      reject(new Error('The server must be started with rou
    }
    if (!port) {
      reject(new Error('The server must be started with an
    }

    const app = express()
    app.use(morgan('dev'))
    app.use(bodyparser.json())
    app.use(cors())
    app.use(helmet())
    app.use((err, req, res, next) => {
      reject(new Error('Bad Gateway!, err:' + err))
      res.status(status.BAD_GATEWAY).send('url not found!')
      next()
    })

    for (let id of Reflect.ownKeys(routes)) {
```

Here what we are doing is creating an `express app` then we loop our routes, and register it to the app as middleware, and in that middleware we are applying another middleware called `http-proxy-middleware` that will proxy the request to the correct **microservice**, and finally we start our server.

So now we are ready to run our **api-gateway** locally and run a super integration test, where we are going to call all our endpoints declared in our raml files, and we are calling in it through the **api-gateway.**

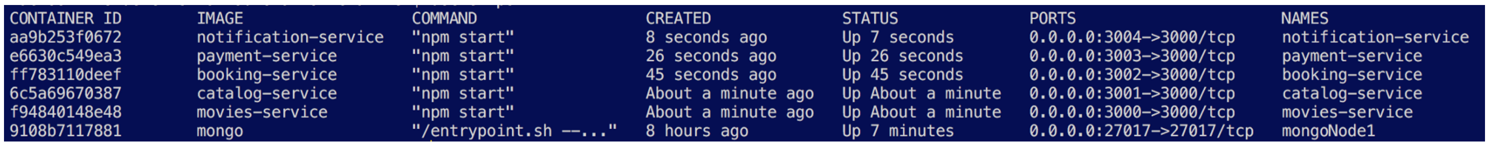
But first let's re create our containers and for that, let's create an automated script called `start_all_microservices.sh` that do all that job for us like the following:

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

eval `docker-machine env manager1`

array=('./movies-service'
  './cinema-catalog-service'
  './booking-service'
  './payment-service'
  './notification-service'
)

# we go to the root of the project
cd ..

```

But we need to have modified each `start-service.sh` of every microservice with the new label flag like the following:

```
docker run --name booking-service -l=apiRoute='/booking' -p
3002:3000 --env-file env -d booking-service

docker run --name catalog-service -l=apiRoute='/cinemas' -p
3001:3000 --env-file env -d catalog-service

// and so on

# and once we have it, let's run the command
$ bash < start_all_microservice.sh
```

And we need to have something like this

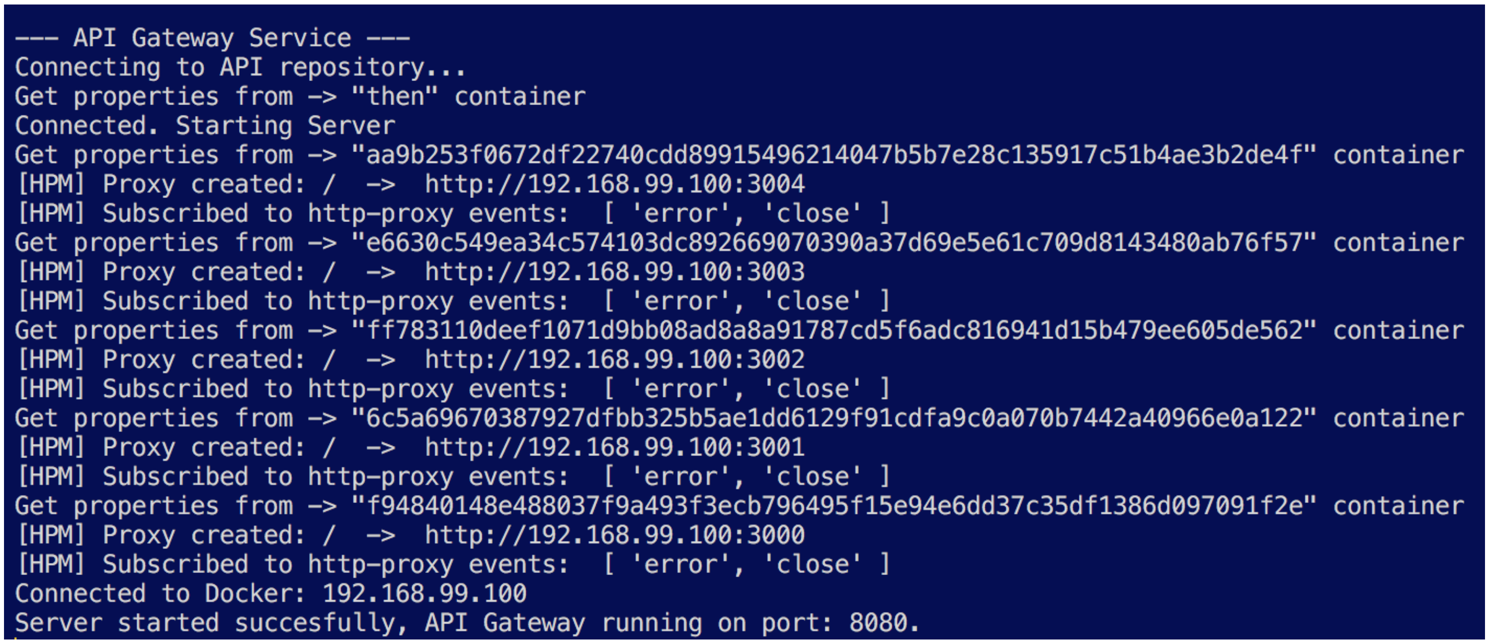


```
CONTAINER ID        IMAGE                  COMMAND                  CREATED              STATUS              PORTS                      NAMES
aa9b253f0672        notification-service   "npm start"              8 seconds ago        Up 7 seconds        0.0.0.0:3004->3000/tcp     notification-service
e6630c549ea3        payment-service        "npm start"              26 seconds ago       Up 26 seconds       0.0.0.0:3003->3000/tcp     payment-service
ff783110deef        booking-service        "npm start"              45 seconds ago       Up 45 seconds       0.0.0.0:3002->3000/tcp     booking-service
6c5a69670387        catalog-service        "npm start"              About a minute ago   Up About a minute   0.0.0.0:3001->3000/tcp     catalog-service
f94840148e48        movies-service         "npm start"              About a minute ago   Up About a minute   0.0.0.0:3000->3000/tcp     movies-service
9108b7117881        mongo                  "/entrypoint.sh --..."   8 hours ago          Up 7 minutes        0.0.0.0:27017->27017/tcp   mongoNode1
```

Fresh and newly created containers with the new label flag, to be able to discover it, in our nodejs api-gateway, so now **let's run our api-gateway,** with the following command:

```
$ npm start
```

This will start a server in our "**there is no place like**": **127.0.0.1:8080, (**our beloved localhost 💞**)** And we should see something like this:



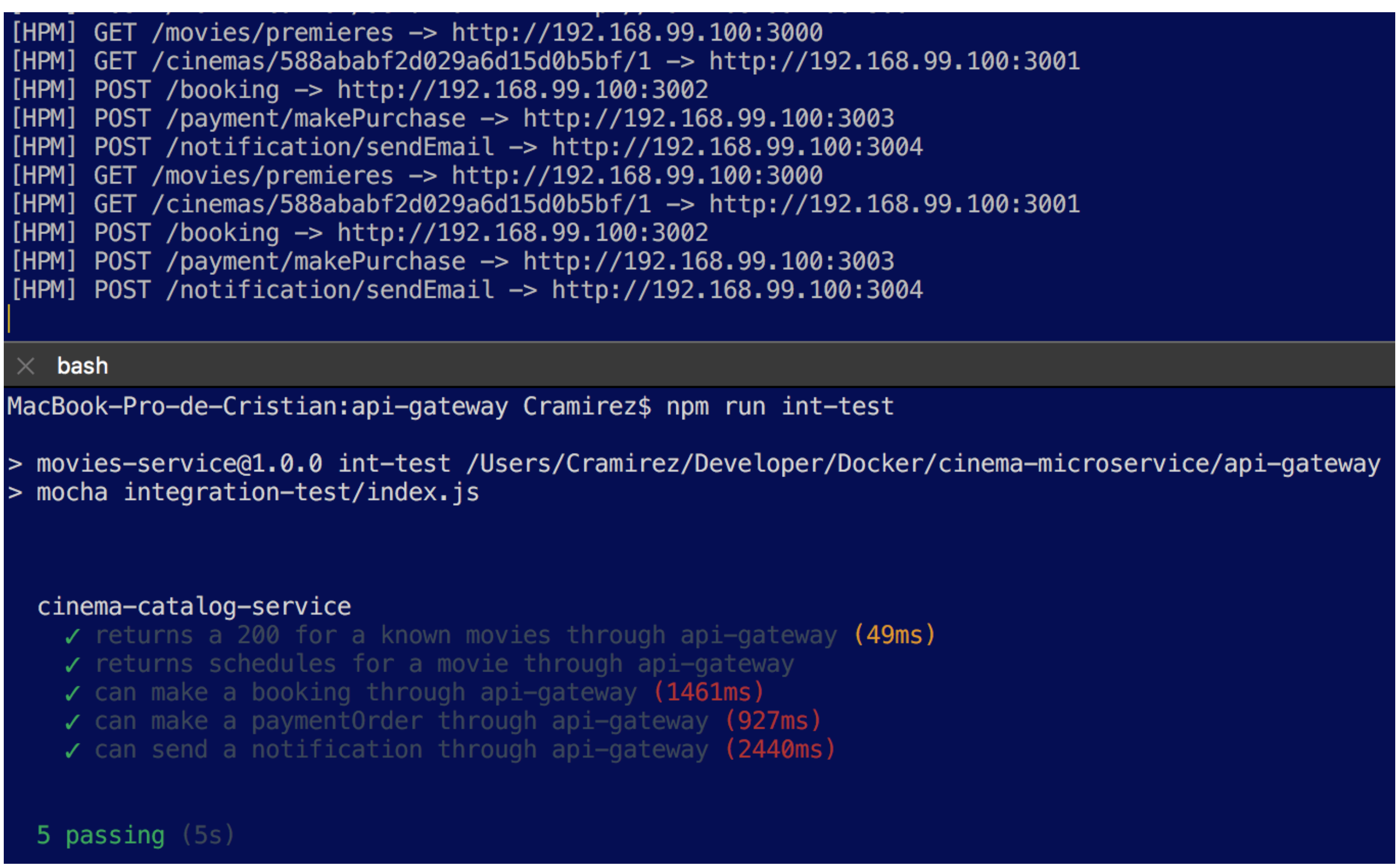
```
--- API Gateway Service ---
Connecting to API repository...
Get properties from -> "then" container
Connected. Starting Server
Get properties from -> "aa9b253f0672df22740cdd89915496214047b5b7e28c135917c51b4ae3b2de4f" container
[HPM] Proxy created: /  ->  http://192.168.99.100:3004
[HPM] Subscribed to http-proxy events:  [ 'error', 'close' ]
Get properties from -> "e6630c549ea34c574103dc892669070390a37d69e5e61c709d8143480ab76f57" container
[HPM] Proxy created: /  ->  http://192.168.99.100:3003
[HPM] Subscribed to http-proxy events:  [ 'error', 'close' ]
Get properties from -> "ff783110deef1071d9bb08ad8a8a91787cd5f6adc816941d15b479ee605de562" container
[HPM] Proxy created: /  ->  http://192.168.99.100:3002
[HPM] Subscribed to http-proxy events:  [ 'error', 'close' ]
Get properties from -> "6c5a69670387927dfbb325b5ae1dd6129f91cdfa9c0a070b7442a40966e0a122" container
[HPM] Proxy created: /  ->  http://192.168.99.100:3001
[HPM] Subscribed to http-proxy events:  [ 'error', 'close' ]
Get properties from -> "f94840148e488037f9a493f3ecb796495f15e94e6dd37c35df1386d097091f2e" container
[HPM] Proxy created: /  ->  http://192.168.99.100:3000
[HPM] Subscribed to http-proxy events:  [ 'error', 'close' ]
Connected to Docker: 192.168.99.100
Server started succesfully, API Gateway running on port: 8080.
```

API gateway console output

Next we need to open another terminal and position it in our **cinemas-microservice** project and run the following command to **execute our super mega integration test** 👩‍🔬👨‍🔬

```
$ npm run int-test
```

And we would have some output like the following:



```
[HPM] GET /movies/premieres -> http://192.168.99.100:3000
[HPM] GET /cinemas/588ababf2d029a6d15d0b5bf/1 -> http://192.168.99.100:3001
[HPM] POST /booking -> http://192.168.99.100:3002
[HPM] POST /payment/makePurchase -> http://192.168.99.100:3003
[HPM] POST /notification/sendEmail -> http://192.168.99.100:3004
[HPM] GET /movies/premieres -> http://192.168.99.100:3000
[HPM] GET /cinemas/588ababf2d029a6d15d0b5bf/1 -> http://192.168.99.100:3001
[HPM] POST /booking -> http://192.168.99.100:3002
[HPM] POST /payment/makePurchase -> http://192.168.99.100:3003
[HPM] POST /notification/sendEmail -> http://192.168.99.100:3004
```

```
MacBook-Pro-de-Cristian:api-gateway Cramirez$ npm run int-test

> movies-service@1.0.0 int-test /Users/Cramirez/Developer/Docker/cinema-microservice/api-gateway
> mocha integration-test/index.js



  cinema-catalog-service
    ✓ returns a 200 for a known movies through api-gateway (49ms)
    ✓ returns schedules for a movie through api-gateway
    ✓ can make a booking through api-gateway (1461ms)
    ✓ can make a paymentOrder through api-gateway (927ms)
    ✓ can send a notification through api-gateway (2440ms)


  5 passing (5s)
```

In the upper console we can see how the proxy is dispatching and redirecting to the corresponding url:port, and in the lower console, we can see how all tests has been passed correctly.

There is one more thing that we need to do to our **api-gateway**, and that is to **dockerize** our **api-gateway**, but let me tell you that i couldn't dockerize it, i'm still figuring out how to talk from a container to the docker-machine(host), if you can figured it out, you are welcome to post a comment below, share us what you find and applied to dockerize our **api-gateway**, so in the meantime, i am still researching how to accomplish this task, so thats why we made our **integration test** in our

“there is no place like 127.0.0.1” address and not to the docker-machine ip address, so stay tuned and i will tell you what i have figure out.

# Update 14/02/17 Solution for api-gateway to dockerize

Hello again, well let me tell you what did i find, and how i could solve this issue, so let`s start to dockerize our **api-gateway.**

So following our pattern let’s look at our `start-service.sh` inside the **api-gateway** folder and this file should contain somenthing like this:

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

eval `docker-machine env manager1`

docker rm -f api-gateway-service

docker rmi api-gateway-service

docker image prune

docker volume prune

docker build -t api-gateway-service .

docker run --name api-gateway-service -v
/Users/Cramirez/.docker/machine/machines/manager1:/certs --
net='host' --env-file env -d api-gateway-service
```

Here we are adding some flags so let’s give them a look.

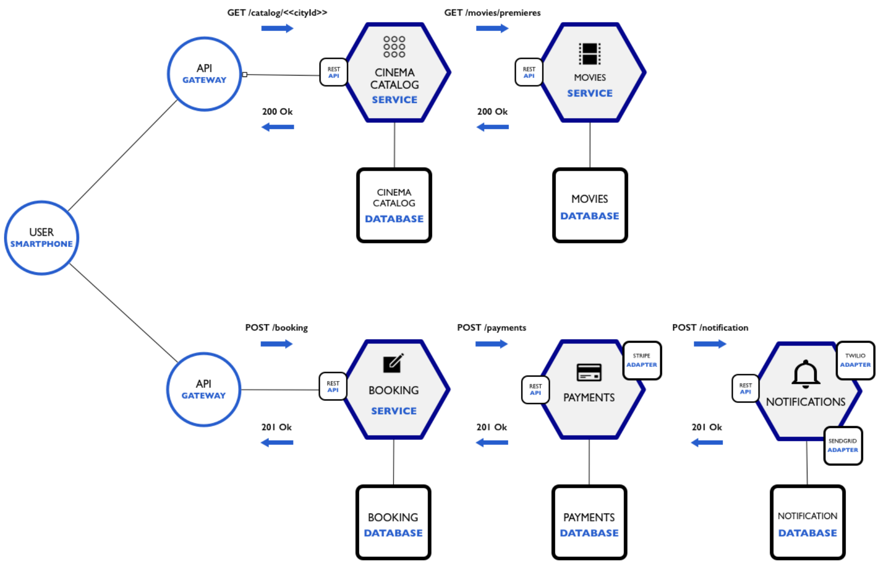
- volume flag `-v` here we are binding our docker-machine certificates folder to our container so it can be able to read those certificates
- net flag `--net` here we are telling the container to attach the host network adapter, and **this will bind the docker-machine ip to our api-gateway container,** the only thing we need to care about, is for ports available so we can run our **api-gateway**, in my case i am executing the api-gateway on port 8080

So now we can call again our super duper test, but know with our **docker-machine ip + api-gateway port,** for example `https://192.168.99.100:8080`

# Time for a recap

What we have done…

We have concluded the following diagram:



We just build 3 services in this article the **payment service,** the **notification service** and the super duper **api-gateway 😎.**

We have integrated all the functionality of our microservices, so now we can make, **GET** and **POST** requests, and in those request some of our microservices has data validation, database interaction, third party services interaction, etc… in summary we have done a lot, but let me tell you dear readers, this is not the end of the series, maybe this could start to see like the **game of thrones,** where there is no ending coming (**winter is coming ❄️,** to much tv shows for me), but there is a lot more to do with our **cinema-microservice system,** so stay tuned for some fun stuff.

What do we have achieved with our **api-gateway 🤔 ?**

we deliver our microservice security, to be handled only in our **api-gateway server,** and this is called **Transport security ✔**

we made a semi dynamic, container service discovery, why semi ? because we are hardcoding our routes with the **label flag** added to our containers, and we create our container one by one, but this help us to redirect the request to the correct target, and this is called **Request dispatching** ✔ and **Dependency resolution** ✔

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 for a little bit more advance programming. 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

we will start to create a **Docker Swarm Cluster**, and our microservices will become **Docker Services**, and and and …. our **api-gateway will become more dynamically** and we will going to have a very robust and dynamic microservice system, also there is something left for our microservice that we havent talked about and applied, the famously **Twelve-Factor App for microservices,** we will see how to accomplish those twelve factors, so stay tuned, stay curious 👀🕵🕵♀.

. . .

# Thanks for reading

Let me remember you, this article is part of "***Build a NodeJS cinema microservice***" series so, If you want to keep going on the series there is the link below of the 5th chapter.

Deploy Nodejs microservices to a Docker Swarm Cluster [Docker from zero to hero]

This is the 🖐 fifth article from the series "Build a NodeJS cinema microservice". This series of articles…

medium.com

I hope you've enjoyed this article, i'm currently still exploring the NodeJS and Microservices world, so as always i am open to accept feedback or contributions.

Do recommend this if it helped you. In-case you have questions on any aspect of this series or need my help in understanding something, feel free to tweet me or leave a comment below, recommend it to a friend, share it or read it again 📖.

So in the meantime, stay tuned, stay curious ✌️👨‍💻👩‍💻

¡ Hasta la próxima ! 😁

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

. . .

# 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

# References

- Building Microservices: Using an API Gateway
- Pattern: API Gateway