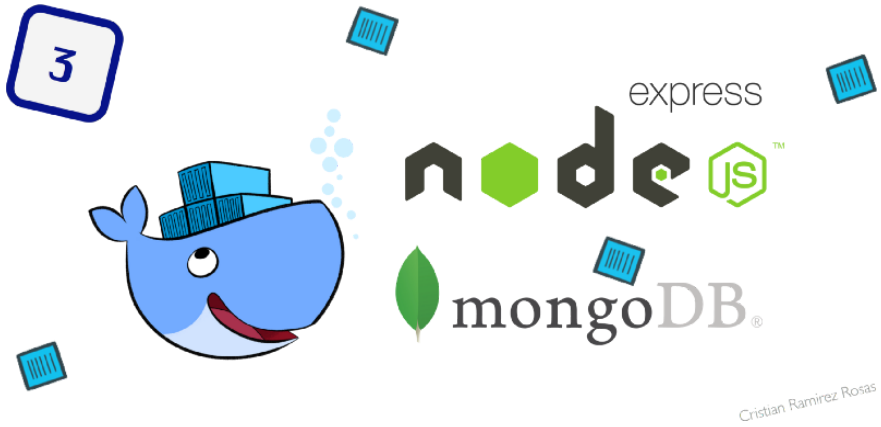
Cristian Ramirez Follow
FullStack MEAN Engineer with some Docker knowledge Linkedin: https://www.linkedin.com/in/cristian-r...
Feb 7 · 12 min read

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



images from the web—cover made by me

> Hello community this is the third article from the series “Build a NodeJS cinema microservice”. This series of articles demonstrate how to build API’s 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

- we talk about **what is a microservice,** we saw what are the **benefits** and **drawback** of **microservices**.
- we define our **cinema microservice architecture.**
- we design and developed our **movies service** and **cinema-catalog service.**
- we made an API for each service and made **unit testing** to our API’s.

- we compose our API's to make it a service and run it into a **Docker** container.
- we made an **integration tests** to our **services** running on **Docker.**
- we talk about **microservices security** and we implement the **HTTP/2 protocol.**
- we made a **stress test** to the **cinema-catalog service**.

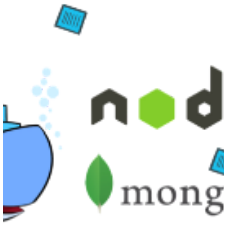
if you haven't read the previous chapters, you're missing some fun stuff 🤘, i will put the links below, so you can give it a look 👀.

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

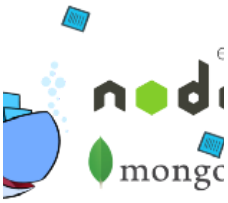


Build the cinema microservice (part 2)

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

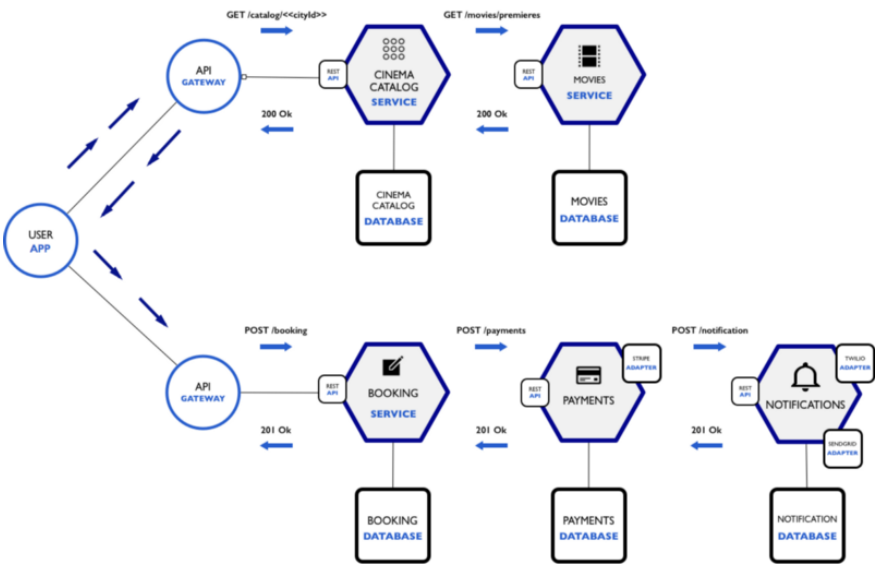
This is the ✌ second article from the series "Build a NodeJS cinema microservice".

medium.com



. . .

In the previous chapters we have fulfilled the superior sub architecture from the following diagram and we are going to start to develop the inferior sub architecture in this chapter.



At this point a final user already can see what movie premieres are available at a cinema and can select a cinema and request a booking, so in this article, we will continue building the **cinema architecture** and we are going to see what is happening inside the **booking service**, so follow up 😎 and let's learn some interesting things.

What we are going to use for this article is:

- NodeJS version 7.5.0
- 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 a github repository, so you can be up to date, repo link at branch **step-2.**

Dependency Injection in NodeJS

Until here we have built 2 API's for our microservices, but in those microservices we haven't do so much configurations and so much development, because of it's nature and simplicity, but the moment has come folks, in our **booking microservice**, we are going to see a little bit more interactions with other services, and for that we will need

more dependencies to fulfill the task assigned to this microservice, but to not start making some 💩 spaghetti code, as good developers we are going to follow up some development design patterns, and for that we will see what it is **"Dependency Injection".**

> To achieve excellent design patterns we have to understand very well and applied the **S.O.L.I.D. Principles**, i have made an article about this using javascript, so you can give it a look 🤓, and see what this principles are and how can we benefit from them.

S.O.L.I.D The first 5 principles of Object Oriented Design with JavaScript

I've found a very good article explaining the S.O.L.I.D. principles, if you are familiar with PHP, you can read the...

medium.com

Before we start talking about dependency injection, If you are not familiar with it, you can watch the following video before going on.

Dependency Injection basics- Fun Fun Function

Dependency injection is a software design pattern in which one or more dependencies (or services) are injected, or passed by reference, into a dependent object.

Why is it important to understand what is dependency injection ?, it's important because it give us 3 major points in development patterns like the following up:

- **Decoupling:** Dependency injection makes our modules less coupled and with that achieved we gain major maintability.
- **Unit testing:** With dependency injection, we can make better unit testing for every module, also our code would be less buggy.
- **Faster development:** With dependency injection, after the interfaces are defined it is easy to work without any merge conflicts.

So until now in our microservices we already have made **dependency injection(DI)** at the `index.js`

```
1  // more code
2
3  mediator.on('db.ready', (db) => {
4    let rep
5    // here we are making DI to the repository
6    // we are injecting the database object and the ObjectID
7    repository.connect({
8      db,
9      ObjectID: config.ObjectID
10   })
11   .then(repo => {
12       console.log('Connected. Starting Server')
13       rep = repo
14       // here we are also making DI to the server
15       // we are injecting serverSettings and the repo objec
16       return server.start({
17         port: config.serverSettings.port,
18         ssl: config.serverSettings.ssl,
19         repo
20       })
```

What we made at `index.js` files was manually **DI** because we don't need to do more, but know in the **booking service**, we will need to make a better approach of **DI**, let's see why we need that, so again

before we start building our API, let's figure out what the **booking service** needs to do.

- The booking service needs a booking object and a user object, and after making the booking action first we need to validate those objects.
- Once validate we are able to continue the process for start making the purchase of the tickets.
- The booking service needs the user credit card information to make the purchase of the tickets, via the **payment service.**
- when the charge is made successfully we need to send a notification, via the **notification service.**
- Also we need to generate the tickets for the user, a send the tickets and the purchase orderId code back to the user.

So here our development task has been incremented a little, so the code will too, and that's why we need to make a single source of truth for **DI,** since we are going to be doing more functionality.

Building the microservice

Ok so first let's see how is going to be our **RAML** file, for the **booking service.**

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

types:
  Booking:
    properties:
      city: string
      cinema: string
      movie: string
      schedule: datetime
      cinemaRoom: string
      seats: array
      totalAmount: number


  User:
    properties:
      name: string
      lastname: string
      email: string
      creditcard: object
      phoneNumber?: string
      membership?: number

  Ticket:
    properties:
      cinema: string
      schedule: string
      movie: string
      seat: string
      cinemaRoom: string
      orderId: string


resourceTypes:
  GET:
    get:
      responses:
        200:
```

```
42            body:
43              application/json:
44                type: <<item>>
```

We define 3 model objects, that are **Booking, User** and **Ticket,** so since this is our first **POST** request that we see in the series, there is one NodeJS **best practice** that we haven't made use of, **data validation.** There is a good quote that i read from the article "Build beautiful node API's" that said the following:

> *Always, alway, always validate the incoming (and also outgoing) data. There are modules like joi and express-validator to help you sanitize the data elegantly.— Azat Mardan*

So we are going to start building our **booking service** from here. As the previous chapter, we are still going to use the same project structure, but we are going to make this time a little bit more modifications. So let's stop talking 🗣 about theory and let the **hunger games** begin, sorry again, so let the fun begin let's do some ¡coding! 👩‍💻👨‍💻.

First we need to create a new folder under the `/src` folder called `models`

```
booking-service/src $ mkdir models

# Now let's move to the folder and create some files

booking-service/src/models $ touch user.js booking.js
ticket.js

# Now is moment to install a new npm package for data
validation

npm i -S joi --silent
```

Ok now that we are set is moment to start coding our schema validation objects, **MongoDB** also has built in a validation object, but here what we need to validate is that the object is complete that's why i choose joi,

also joi allow us to validate the data at the same time too, so lets begin with `booking.model.js` then with the `ticket.model.js` and finally with the `user.model.js`

```
const bookingSchema = (joi) => ({
  bookingSchema: joi.object().keys({
    city: joi.string(),
    schedule: joi.date().min('now'),
    movie: joi.string(),
    cinemaRoom: joi.number(),
    seats: joi.array().items(joi.string()).single(),
    totalAmount: joi.number()
  })
})

module.exports = bookingSchema
```

booking.model.js hosted with ♡ by GitHub view raw

```
const ticketSchema = (joi) => ({
  ticketSchema: joi.object().keys({
    cinema: joi.string(),
    schedule: joi.date().min('now'),
    movie: joi.string(),
    seat: joi.array().items(joi.string()).single(),
    cinemaRoom: joi.number(),
    orderId: joi.number()
  })
})

module.exports = ticketSchema
```

If you don't know about `joi` you can check their github documentation here: link-to-documentation.

Now let's code the model `index.js` to expose a validate function like the following:

```
const joi = require('joi')
const user = require('./user.model')(joi)
const booking = require('./booking.model')(joi)
const ticket = require('./ticket.model')(joi)

const schemas = Object.create({user, booking, ticket})

const schemaValidator = (object, type) => {
  return new Promise((resolve, reject) => {
    if (!object) {
      reject(new Error('object to validate not provided'))
    }
    if (!type) {
      reject(new Error('schema type to validate not provide
    }

    const {error, value} = joi.validate(object, schemas[typ
```

So what we have made, we have applied the **single responsibility**, from the **solid principles** where every model has its own validation, we also applied **open-close principle**, where the **schema validator** function has the ability to validate as many models as we declare, so let's see how is our test file for this models.

```
/* eslint-env mocha */
const test = require('assert')
const {validate} = require('./')

console.log(Object.getPrototypeOf(validate))

describe('Schemas Validation', () => {
  it('can validate a booking object', (done) => {
    const now = new Date()
    now.setDate(now.getDate() + 1)

    const testBooking = {
      city: 'Morelia',
      cinema: 'Plaza Morelia',
      movie: 'Assasins Creed',
      schedule: now,
      cinemaRoom: 7,
      seats: ['45'],
      totalAmount: 71
    }

    validate(testBooking, 'booking')
      .then(value => {
        console.log('validated')
        console.log(value)
        done()
      })
      .catch(err => {
        console.log(err)
        done()
      })
  })

  it('can validate a user object', (done) => {
    const testUser = {
      name: 'Cristian',
      lastName: 'Ramirez',
      email: 'cristiano@nupp.com',
      creditCard: '1111222233334444',
      membership: '7777888899990000'
    }
```

```
42
43      validate(testUser, 'user')
44        .then(value => {
45          console.log('validated')
46          console.log(value)
47          done()
48        })
```

The next file to review is going to be the `api/booking.js` at this point, we are starting to get into much trouble, **¿ why ?,** because here we are going to be interacting with **two external services**, the **payment service** and the **notification service**, and this kind of interactions can lead us to rethink the architecture of the microservice, there is something called **Event Driven Data Management** and **CQRS,** but those topics are going to be saved for further chapters on the series and to not making this chapter to long and complicated, so in the meantime, let's made our interactions with the services simple for this chapter.

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

module.exports = ({repo}, app) => {
  app.post('/booking', (req, res, next) => {

    // we grab the dependencies need it for this route
    const validate = req.container.resolve('validate')
    const paymentService = req.container.resolve('paymentSe
    const notificationService = req.container.resolve('noti

    Promise.all([
      validate(req.body.user, 'user'),
      validate(req.body.booking, 'booking')
    ])
    .then(([user, booking]) => {
      const payment = {
        userName: user.name + ' ' + user.lastName,
        currency: 'mxn',
        number: user.creditCard.number,
        cvc: user.creditCard.cvc,
        exp_month: user.creditCard.exp_month,
        exp_year: user.creditCard.exp_year,
        amount: booking.amount,
        description: `
          Tickect(s) for movie ${booking.movie},
          with seat(s) ${booking.seats.toString()}
          at time ${booking.schedule}`
      }

      return Promise.all([
        // we call the payment service
        paymentService(payment),
        Promise.resolve(user),
        Promise.resolve(booking)
      ])
    })
    .then(([paid, user, booking]) => {
```

As you can see here, we are making use of the expressjs **middleware**, and we are making use of the **container** where we register our

dependencies from a single source of truth.

But where is coming the **container** of **DI ?**

Well we have made a little change to our project structure, mostly at the `config` folder and now is like the following:

```
.
|-- config
|   |-- db
|   |   |-- index.js
|   |   |-- mongo.js
|   |   `-- mongo.spec.js
|   |-- di
|   |   |-- di.js
|   |   `-- index.js
|   |-- ssl
|   |   |-- certificates
|   |   `-- index.js
|   |-- config.js
|   |-- index.spec.js
|   `-- index.js
```

At the `config/index.js` file we are including mostly all the configurations as well as the **DI** services:

```
const {dbSettings, serverSettings} = require('./config')
const database = require('./db')
const {initDI} = require('./di')
const models = require('../models')
const services = require('../services')

const init = initDI.bind(null, {serverSettings, dbSettings,
database, models, services})

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

In the code above we are seeing something a little bit rare, and let me zoom it for you again:

```
initDI.bind(null, {serverSettings, dbSettings, database,
models, services})
```

What are we doing here ?, i said that we were configuring **DI,** but here we are making something called **Inversion of control**, yes yes i know that this is to much technical words, and might sound bloated, but it is easy to understand, once you get it, if you haven't heard about **IoC**, i recommend you to watch the following video:

So our **DI** function, doesn't need to know where our dependencies are coming from, it only needs to register our dependencies to be available at our application, so our `di.js` file looks like the following:

```
const { createContainer, asValue, asFunction, asClass } = r

function initDI ({serverSettings, dbSettings, database, mod
  mediator.once('init', () => {
    mediator.on('db.ready', (db) => {
      const container = createContainer()

      // loading dependecies in a single source of truth
      container.register({
        database: asValue(db).singleton(),
        validate: asValue(models.validate),
        booking: asValue(models.booking),
        user: asValue(models.booking),
        ticket: asValue(models.booking),
        ObjectID: asClass(database.ObjectID),
        serverSettings: asValue(serverSettings),
        paymentService: asValue(services.paymentService),
        notificationService: asValue(services.notificationS
      })

      // we emit the container to be able to use it in the
      mediator.emit('di.ready', container)
    })
```

As you can see, we are using a npm package called `awilix` for the dependency injection, awilix fulfills the mechanism of dependency injection in nodejs (i am currently evaluating this library, but i use it here to make the examples clear), so to install it we need to to execute next command:

```
npm i -S awilix --silent
```

To comprehend more how does awilix work, you can check out this dependency injection series of articles that the author wrote at the following link: series of di, and awilix documentation.

Now or main `index.js` file will look something like this:

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

console.log('--- Booking Service ---')
console.log('Connecting to movies repository...')

process.on('uncaughtException', (err) => {
  console.error('Unhandled Exception', err)
})

process.on('uncaughtRejection', (err, promise) => {
  console.error('Unhandled Rejection', err)
})

mediator.on('di.ready', (container) => {
  repository.connect(container)
    .then(repo => {
      container.registerFunction({repo})
```

As you can see now, we are only using one single source of truth, that has every dependency we need, available to request it via the container, so how do we set it up to the expressjs **middleware**, like a commented it before, well it's just a couple of lines of code:

```
const express = require('express')
const morgan = require('morgan')
const helmet = require('helmet')
const bodyparser = require('body-parser')
const cors = require('cors')
const spdy = require('spdy')
const _api = require('../api/booking')

const start = (container) => {
  return new Promise((resolve, reject) => {

    // here we grab our dependencies needed for the server
    const {repo, port, ssl} = container.resolve('serverSett

    if (!repo) {
      reject(new Error('The server must be started with a c
    }
    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) => {
      if (err) {
        reject(new Error('Something went wrong!, err:' + er
        res.status(500).send('Something went wrong!')
      }
      next()
    })

    // here is where we register the container as middlewar
```

So basically we are appending the **container** object to the expressjs **req object** and this is how we have it available through all the expressjs routes. If you want to read deeper how the middleware works with

expressjs you can go at this link and check the expressjs documentation.

Well there is a saying that, the better comes last, finally we are going to review the `repository.js` file:

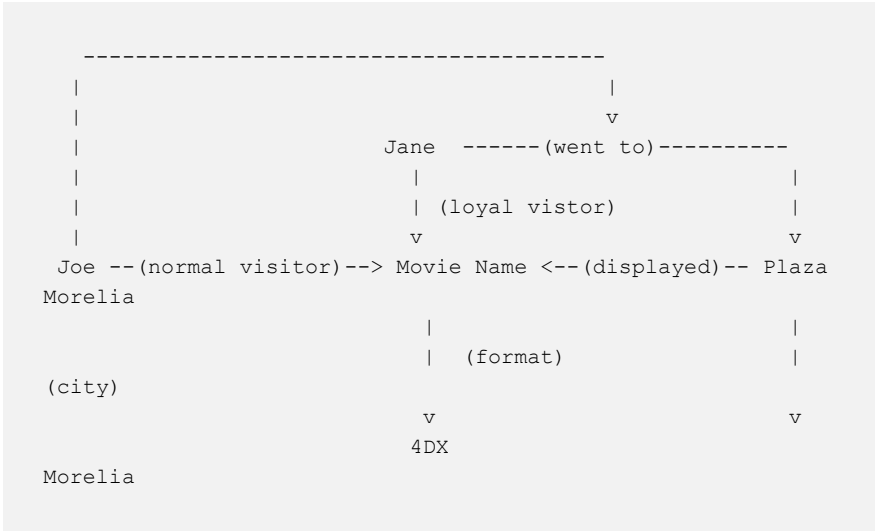
```
'use strict'
const repository = (container) => {
  // we get the db object via the container
  const {db} = container.resolve('database')

  const makeBooking = (user, booking) => {
    return new Promise((resolve, reject) => {
      // payload to be insterted to the booking collection
      const payload = {
        city: booking.city,
        cinema: booking.cinema,
        book: {
          userType: (user.membership) ? 'loyal' : 'normal',
          movie: {
            title: booking.movie.title,
            format: booking.movie.format,
            schedule: booking.schedule
          }
        }
      }

      db.collection('booking').insertOne(payload, (err, boo
        if (err) {
          reject(new Error('An error occuered registring a
        }
        resolve(booked)
      })
    })
  }

  const generateTicket = (paid, booking) => {
    return new Promise((resolve, reject) => {
      // payload of ticket
      const payload = Object.assign({}, {booking, orderId:
      db.collection('tickets').insertOne(payload, (err, tic
        if (err) {
          reject(new Error('an error occured registring a t
        }
        resolve(ticket)
      })
    })
```

```
42    }
43
44    const getOrderById = (orderId) => {
45      return new Promise((resolve, reject) => {
46        const ObjectID = container.resolve('ObjectID')
47        const query = {_id: new ObjectID(orderId)}
48        const response = (err, order) => {
49          if (err) {
50            reject(new Error('An error occuered retrieving a
```

Ok so there is not too much relevance at our `repository.js` maybe could be that we are using for the first time in the series the `insertOne()` method, but there is one thing i want to point in this file, specially at the `makeBooking()` method, if you see the payload object, this is the collection data model schema, but why ?, why we would be using that approach, if we use it, aren't we will be repeating a lot of information ?

Well yes, we will be repeating information and that is not a best practice, but there is a reason to this, and i won't tell you until the next time 🐾, why because there is something very interesting to come on the series …

If you want a hint i will leave this for you're curiousiness 😁



If you can discover what is coming, you're welcome to put a comment in the comments section.

Well let's continue, we have commented that we are interacting with two external services, for simplicity let's see what do we need from this external services

```
# for the payment service we will need to implement
something like the following
```

```
module.exports = (paymentOrder) => {
  return new Promise((resolve, reject) => {
    supertest('url to the payment service')
      .get('/makePurchase')
      .send({paymentOrder})
      .end((err, res) => {
        if (err) {
          reject(new Error('An error occured with the
payment service, err: ' + err))
        }
        resolve(res.body.payment)
      })
  })
}
```

```
# since we haven't made the payment service yet, let's make
something simple to fulfill the article example, like the
following
```

```
module.exports = (paymentOrder) => {
  return new Promise((resolve, reject) => {
    resolve({orderId: Math.floor((Math.random() * 1000) +
1)})
  })
}
```

```
# for the notification service, at the moment we don't need
any information from this service we will not implement it,
this service will have the task for sending an email, sms or
another notification, but we will make this service in the
next chapter.
```

Well we are done for the building of this microservice so, now is time to execute the file inside the repo, with the following command:

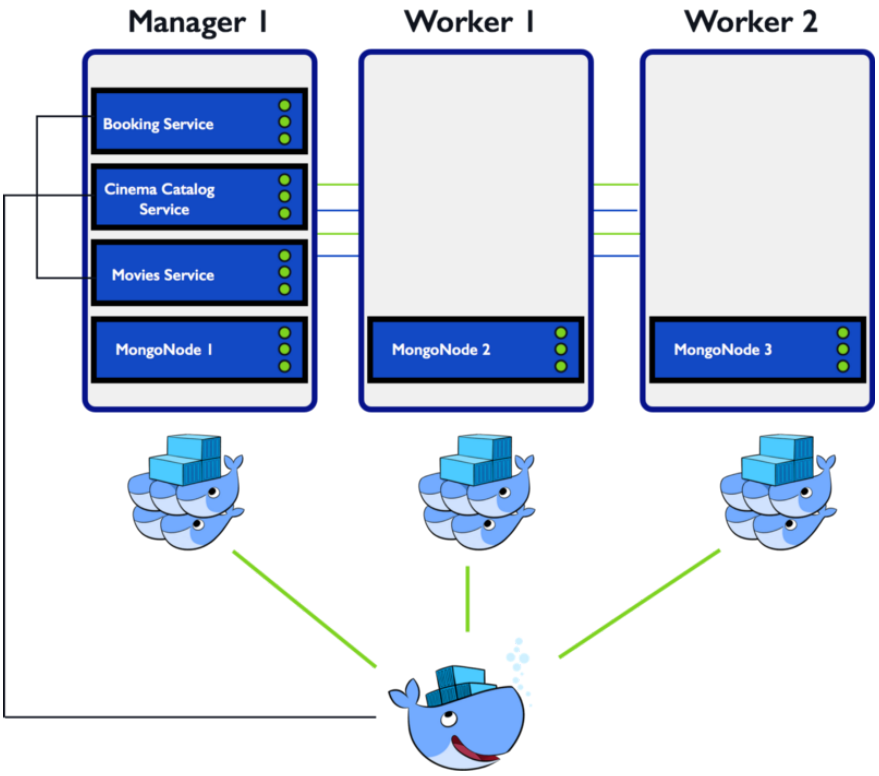
```
$ bash < start_service
```

To have our microservice ready and fully functional into a docker container, and start to make our **integration test**.

Time for a recap

What we have done…

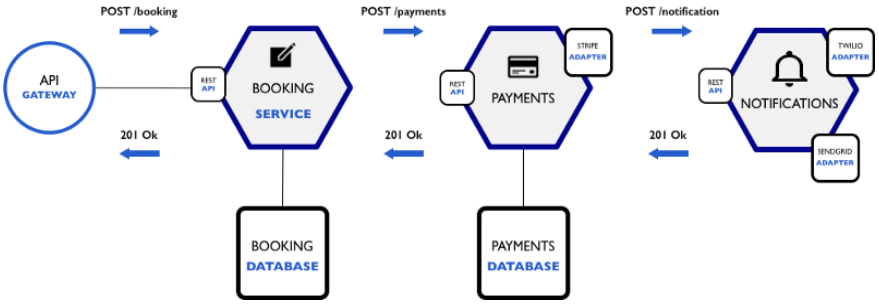
If you have followed my previous chapters we have a system architecture like the following:



Cinema System Architecture

If you noticed our system is starting to take shape, but there's something that doesn't make us feel right, well is that in the **worker 1** and **worker 2**, we don't have any microservice running, and that's because we haven't created it any service in those `docker-machines` but we will do it soon.

Now in the **cinema microservice architecture**, we almost complete the following diagram:



We just build the **booking service** and we make a simple implementation of the **payment service** and the **notification service.**

So what we made in this chapter ¿ 🤔 ?, we learned about **Dependency Injection,** we saw a little bit of **SOLID principles** and **Inversion of Control,** using **NodeJS**, we also make our first **POST request** in our microservice, and we also learned how to validate objects and data with the **joi** library.

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

In the next episodes we will create and finish the implementation of our **Payment Service** and the **Notification Service,** but that is not the interesting part, the interesting part is that we will create our **API Gateway**, since our **cinema microservice** is starting to grow and the **microservices** have the necessity to communicate between each other. But there's so much things left to have a very robust microservice system, and later chapters, we are going to see how to **Adapt the Twelve-Factor App for microservices.**

One more thing ...

There is one more thing … that i want to tell you, and where we are heading to, since our system is starting to grow, and that we already have a Docker ecosystem, we will start to create a **Docker Swarm Cluster**, and our microservices will become **Docker Services**, so stay tuned, stay curious 👀🕵🕵♀.

. . .

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

Microservices and NodeJS patterns

- Fundamental Node.js Design Patterns
- Dependency Injection in Node.js
- Event-Driven Data Management for Microservices
- CQRS Explained—Node.js at Scale
- MongoDB Document Validation

. . .

Final thoughts about the author

This is the third episode of “Building a cinema microservice”, where we have changed a little bit the project structure of our microservices, it will be great to know what you think about this **DI** approach, also i would be great to know until now, what you think about the series, about my writing, and what could i can improve so the series can be better 👼✌👦💻👩💻

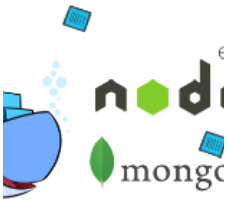
. . .

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 4th chapter:

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

This is the fourth article from the series “Build a NodeJS cinema microservice”. This series of articles…

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