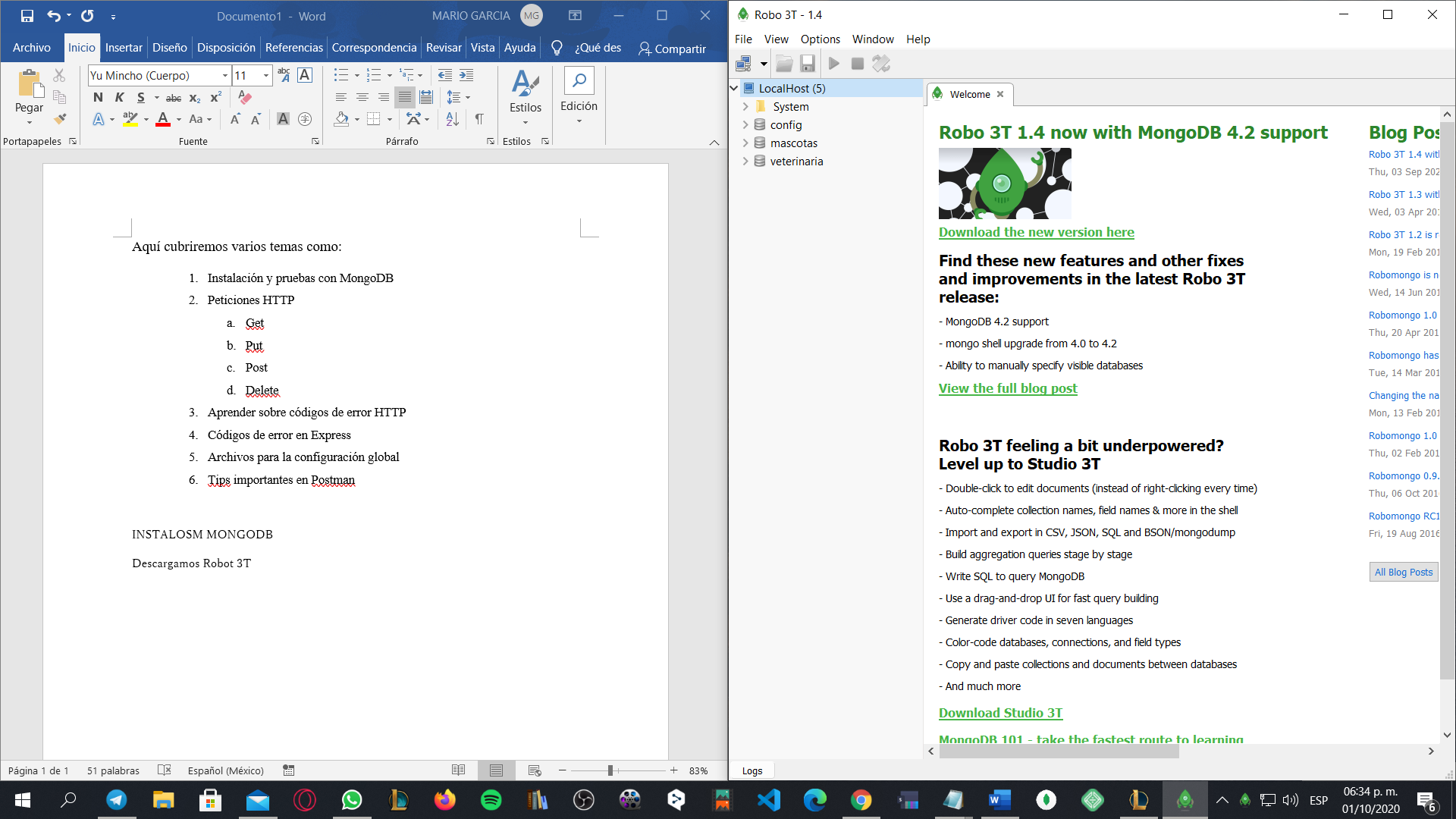
Aquí cubriremos varios temas como:

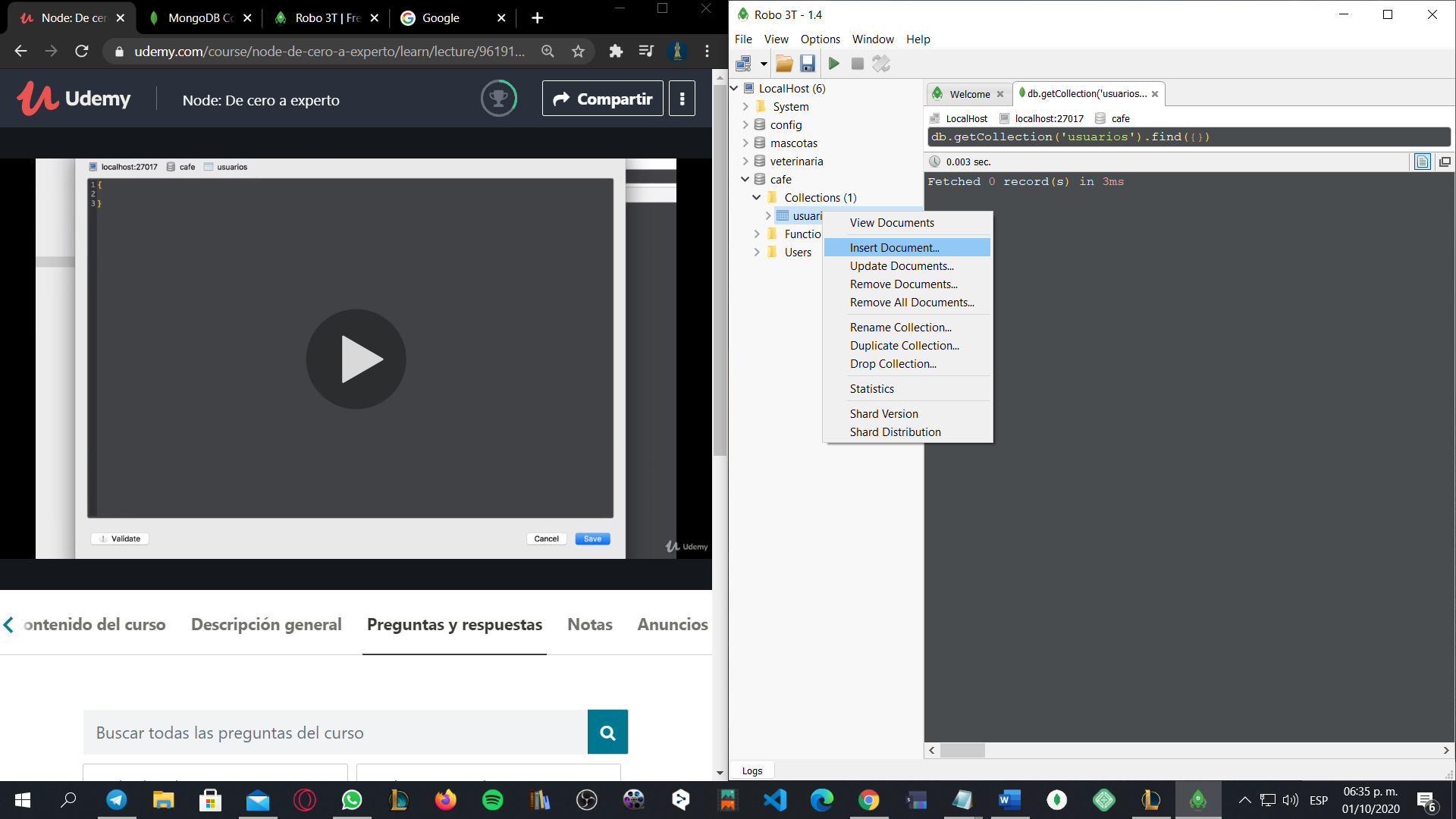
1. Instalación y pruebas con MongoDB
2. Peticiones HTTP
   1. Get
   2. Put
   3. Post
   4. Delete
3. Aprender sobre códigos de error HTTP
4. Códigos de error en Express
5. Archivos para la configuración global
6. Tips importantes en Postman

INSTALOSM MONGODB

Descargamos Robot 3T, creamos una nueva conexión en local y nos aparecen nuestra Bases de datos

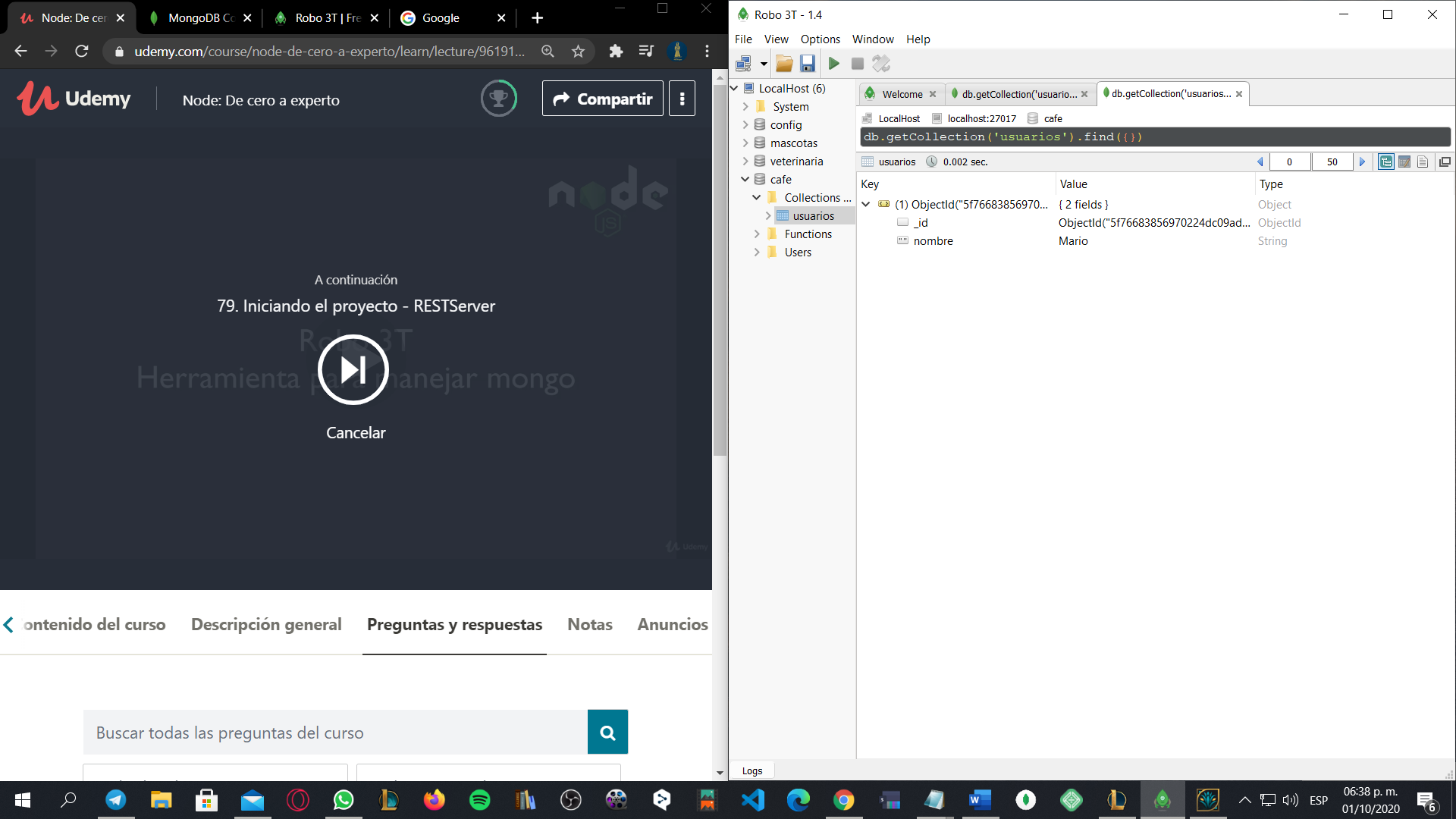


Creamos una colección y click izquiero podemos insertar los documentos



Ya adentro el jason ejemplo

“nombre”:”Mario”



**INICIANDO PROYECTO REST SERVER**

Npm init

Trabajaremos con express

npm install express –save

creamos capreta nueva server con el archivo serve.js

para ejecutarlo node server/server por que esta dentro de la carpeta

const express = require('express')

const app = express()

app.get('/', function(req, res) {

    res.send('Hello World')

})

app.listen(3000, () => {

    console.log('Escuchando el puerto 3000');

})

Como vamos a trabajar con Json el método send lo cambiamos por json

Mejor usar nodemon server/server para que este atento a los cambios

**PETICIONES HTTP - GET -PUT -POST -DELETE**

Vamos a ahacer las 4 peticiones básicas que necesitaremos

const express = require('express')

const app = express()

app.get('/usuario', function(req, res) {

    res.json('get usuario')

})

app.post('/usuario', function(req, res) {

    res.json('post usuario')

})

app.put('/usuario', function(req, res) {

    res.json('put usuario')

})

app.delete('/usuario', function(req, res) {

    res.json('delete usuario')

})

app.listen(3000, () => {

    console.log('Escuchando el puerto 3000');

})

Como nosotros vamos a modificar un usuario en especifico necesitamos poner el comid para que acepte cualquier id, y agregamos la variable id que sale del URL

app.put('/usuario/:id', function(req, res) {

    let id= req.params.id;

    res.json({

        id

    })

})

Trabajamos con el post, tenemos que obtener las variables que mande el cliente..Existe un paquete que se llama body-parser

npm install body-parser –sabe

const bodyParser = require('body-parser');

e implemantamos los middleware

const bodyParser = require('body-parser');

// parse application/x-www-form-urlencoded

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

// parse application/json

app.use(bodyParser.json())

con una variable que es la que,va a tiener toda la información que necesitamos, las peticviones se guardaran en esa variable

app.post('/usuario', function(req, res) {

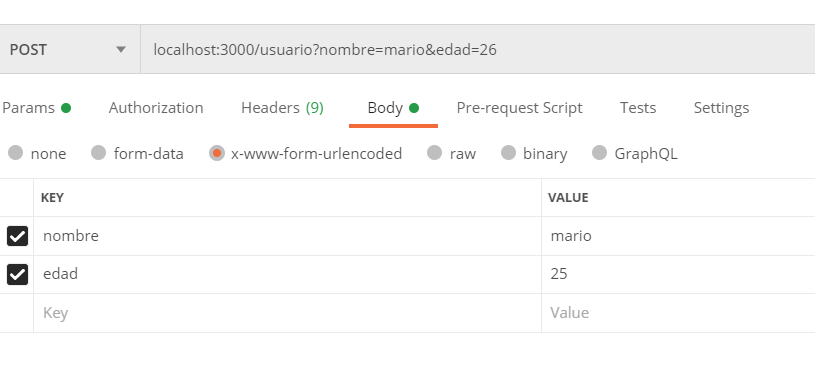
    let body = req.body;

    res.json({

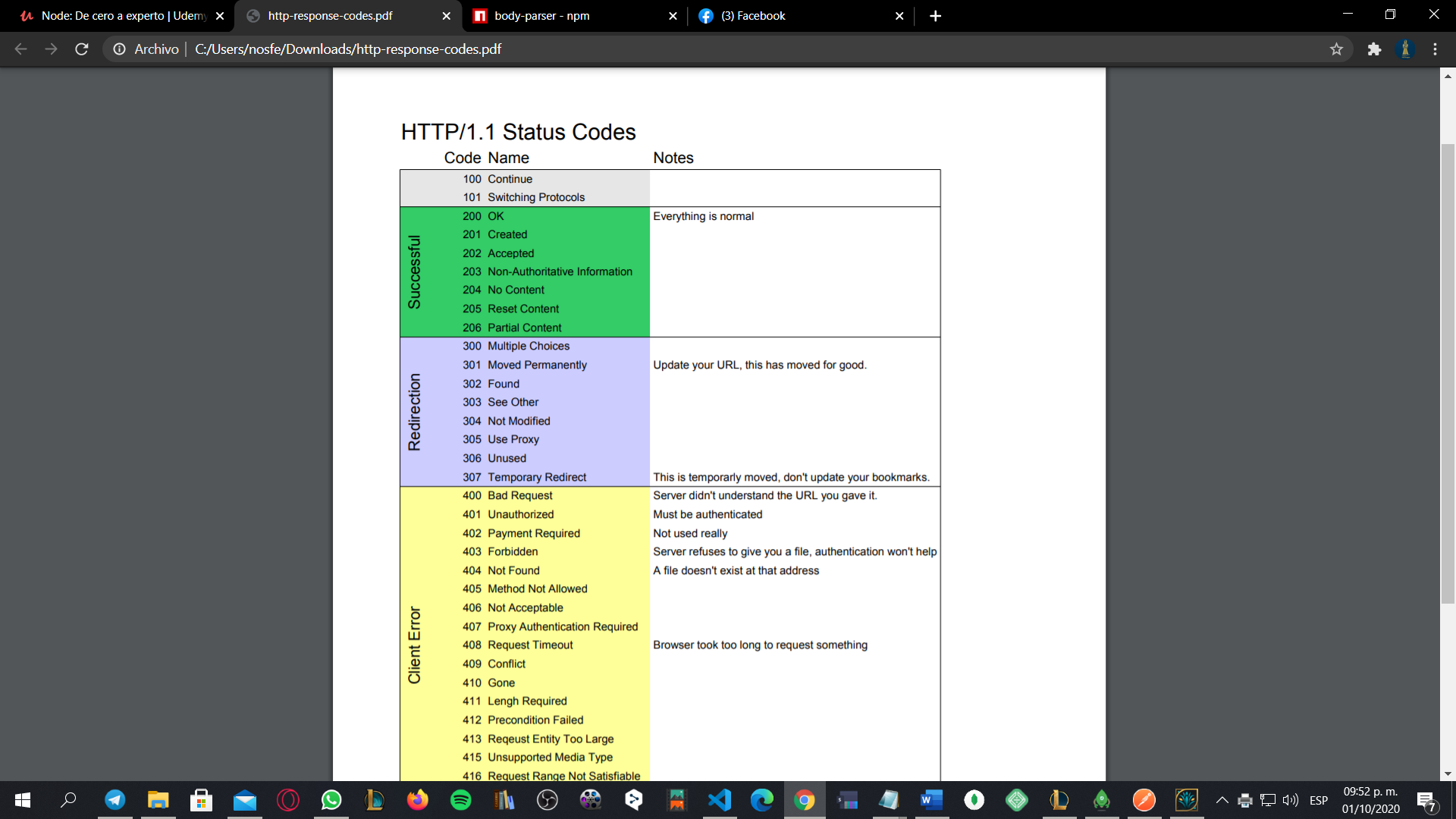
        body

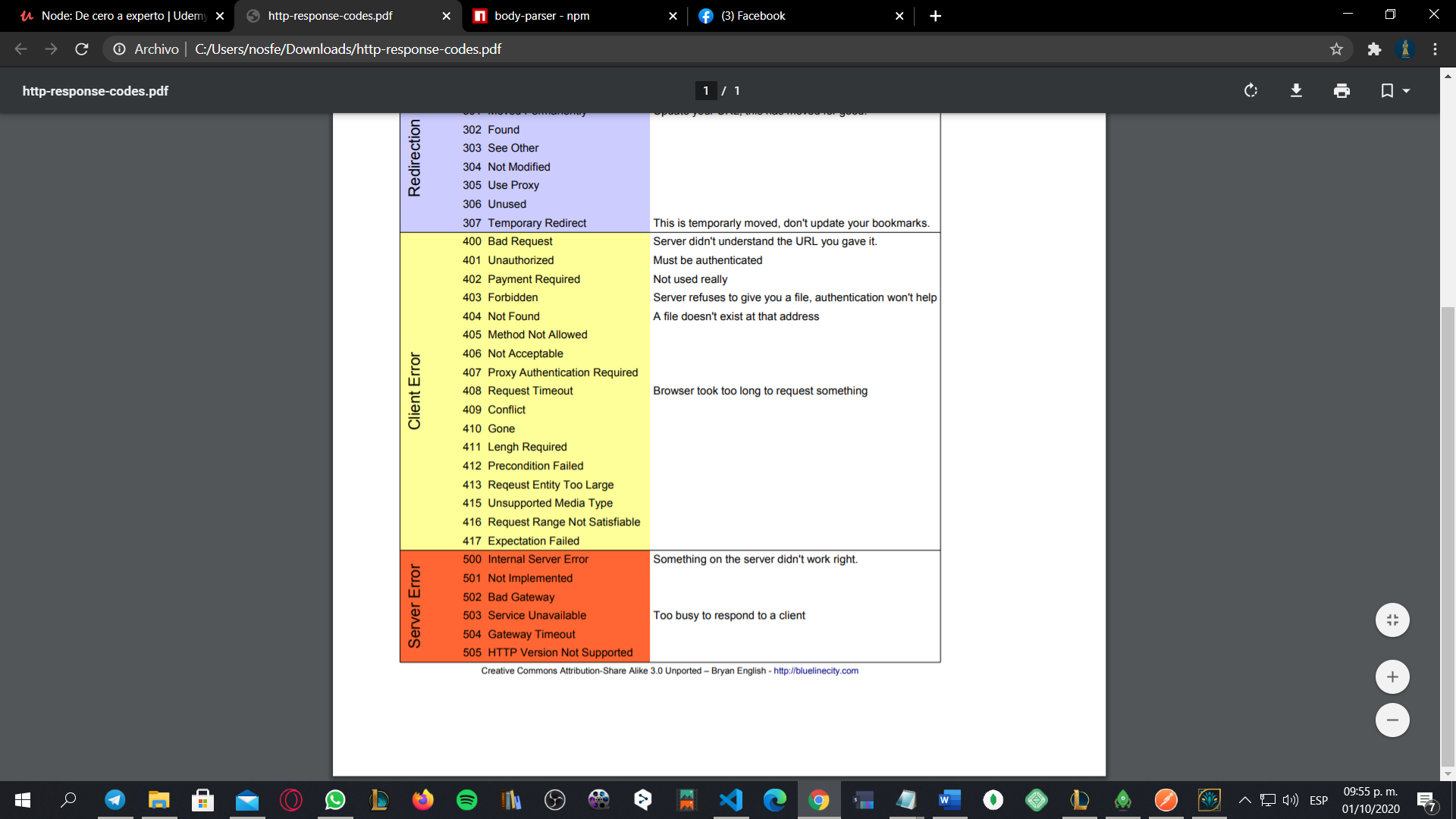
    })

Ya salió la petición pero por x-www-form-urlenconde



**CODIGOS DE RESPUESTAS HTTP**





Quedaría un post con el estatus 400

})

**USANANDO CODIGO DE RESPUESTA HTTP**

**400 para cuando el cliente no introduce un dato**

app.post('/usuario', function(req, res) {

    let body = req.body;

    if (body.nombre === undefined) {

        res.status(400).json({

            ok: false,

            mensaje: 'El nombre es necesario'

        })

    } else {

        res.json({

            persona: body

        })

    }

})

**Creando un archivo de configuración global**

Mas adelante nuestro proyecto tendrá 2 partes “desarrollo y producción”, El puerto que estamos enviando es el 300, cuando subamos a producción no sabremos que puerto tendrá, vamos a crear un archivo para estar entre producción y desarrollo sin necesidad de meter mano al código.

Hacemos una capreta en el server que se llame config y dentro el archivo config.js, esto va a reemplazar el path por que que ponga nustro servidor.

///PUERTO

process.env.PORT = process.env.PORT || 3000;

importamos en el archive principal server,js

require('./config/config');

const express = require('express')

luego donde va estar escuchando ya no será el puerto 3000, va a hacer en ese puerto que ponga el path.

app.listen(process.env.PORT, () => {

    console.log('Escuchando el puerto xxx');

})

RESPALDO DEL REST SERVER A GITHUB

Lo vamos a subir.. ya trabajamos esto

**SUBIRLO A HEROKU (RECOMIENDA LA FORMA QUE HICIMOS ANTERIOR MENTE)**

Podemos hacerlo desde la terminal esta vez.

**Heroku create**

Esto asigna un nombre en automatico del proyecto

Para subirlo

**Git push heroku master**

Y abrirlo en el explorador

**Heroku open**

Tenemos que agregar en el **script del package.json**

**“start” “node server/server.js”**

Temenos que Volver a actualizar el git antes de subirlo por los cambios ya echcos

Git add .

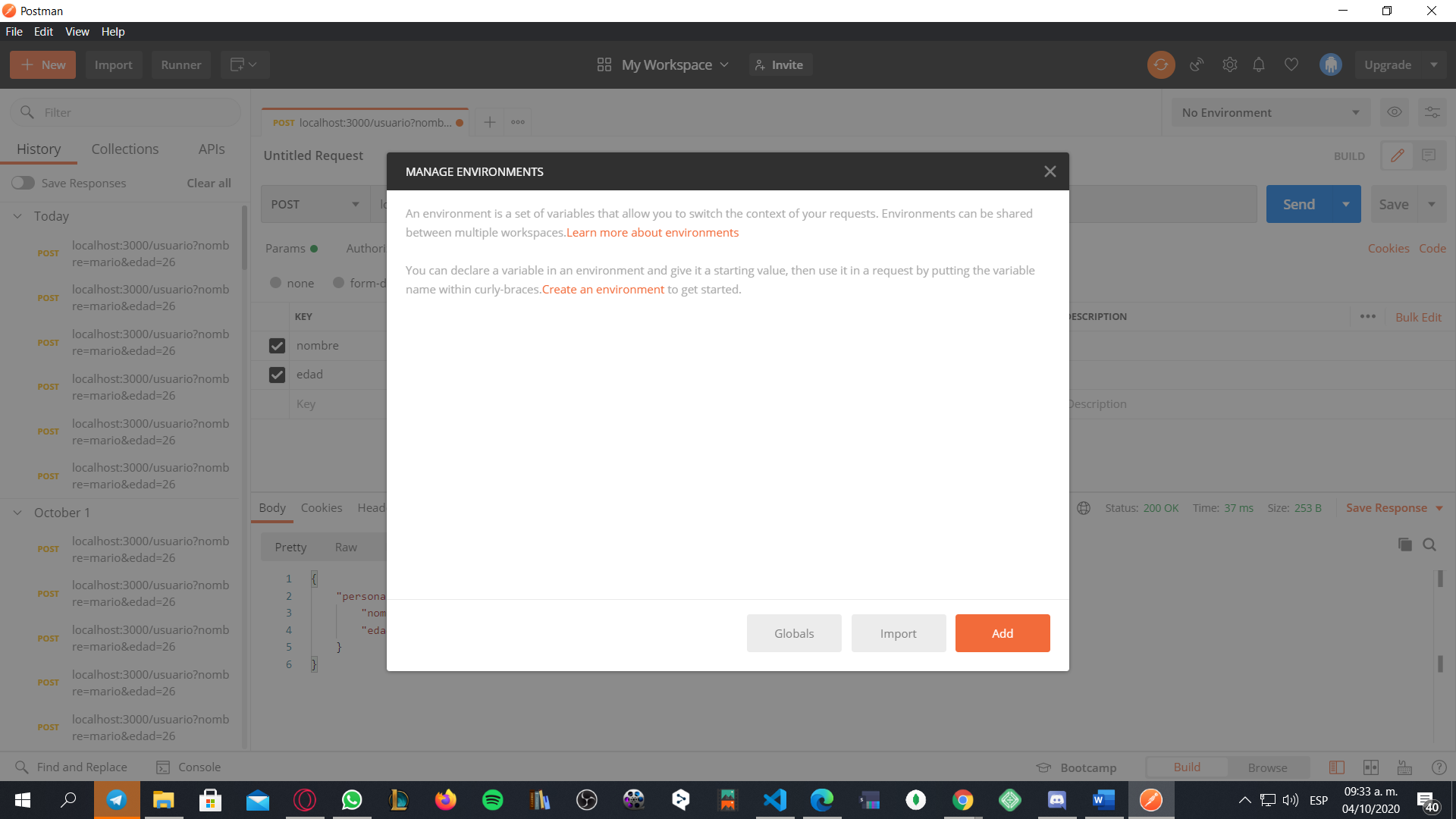
O al archivo que se modifico

**Git commit -am “Npm start listo”** //agrega commit automaticamente y agrega archivos

Y ya hacemos subimos

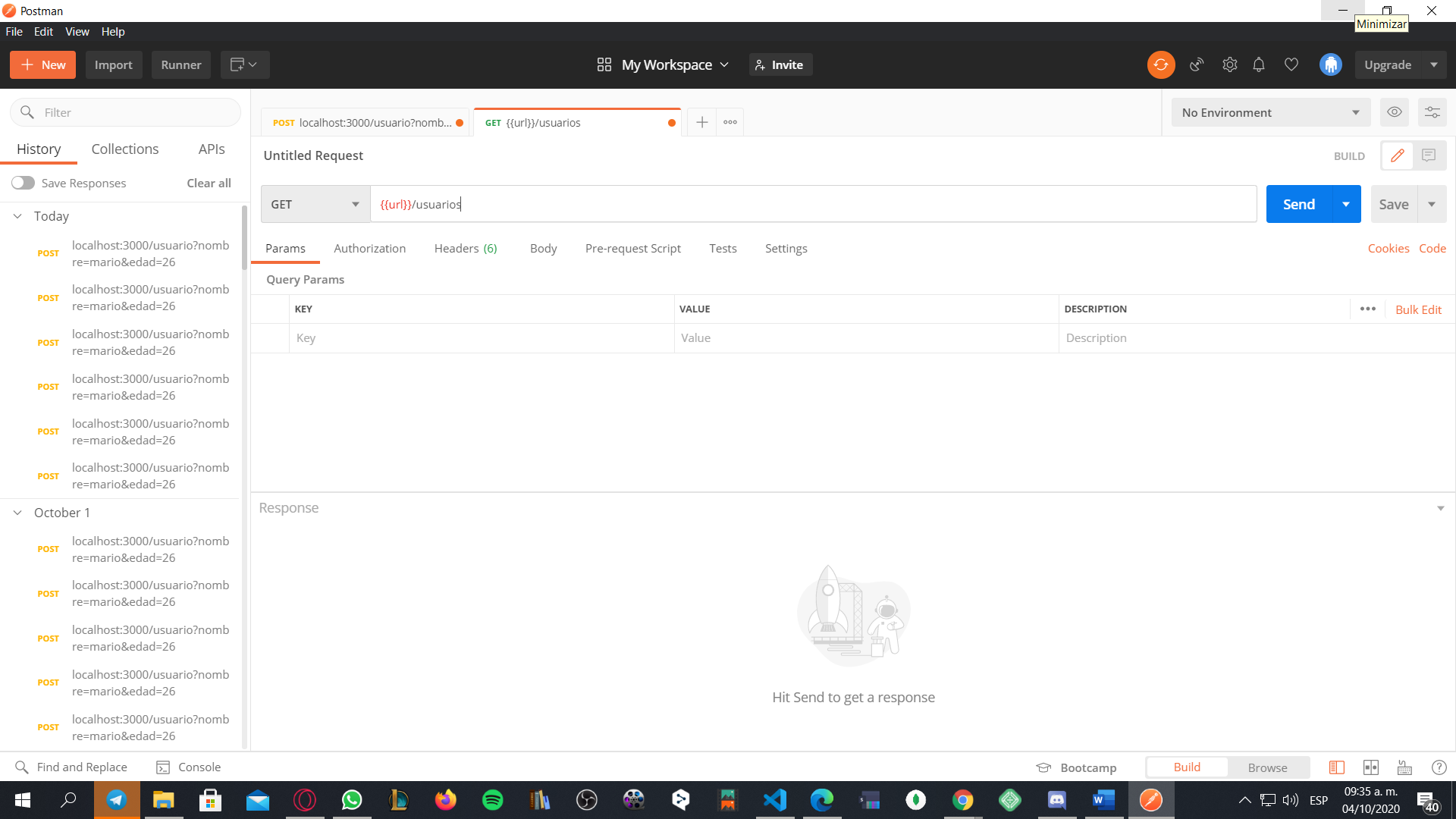
**Git push heroku master**

**TIP AMBIENTE PRODUCCION Y DESARROLLO EN POSTMAN**



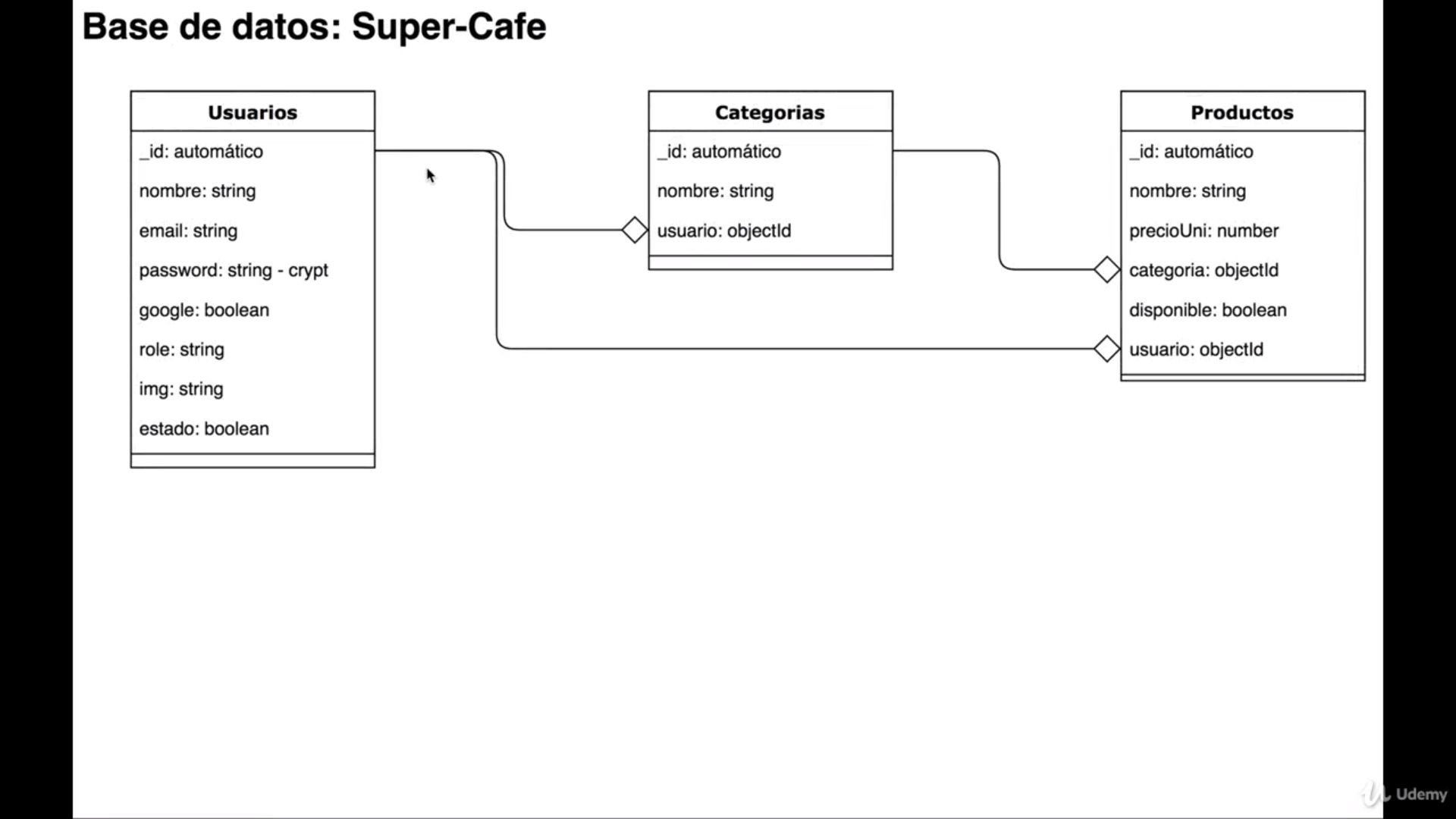
Creamos el ambiente de producción y el otro de desarrollo, el primero con la url que nmos da el servido y el otro con el localhost:3000

Para poner en la url y seleccionarlos para que reconozca los valores que introducimos {{url}} y seleccionar el desarrollo o producción.



Aquí cubriremos varios temas como:

1. Definir los alcances de nuestro RESTServer
2. CRUD
3. Encriptación de contraseñas
4. Validaciones personalizadas
5. Creación de roles
6. Conexiones con MLAB
7. Despliegue de base de datos en la nube
8. Conexión con Robo 3T con base de datos en la nube
9. Configuración de variables de entorno
10. Borrado de archivos
    1. Eliminado físico de la base de datos
    2. Eliminación por estado en un campo de la colección



Vamos a usar la API de Google de sign in para iniciar sesión. En la imagen subida y descargar de archivos, etc. La contraseña usaremos hash de una sola via.

El programa de diagrama es draw.io

CONECTARNOS A LA BASE DE DATOS

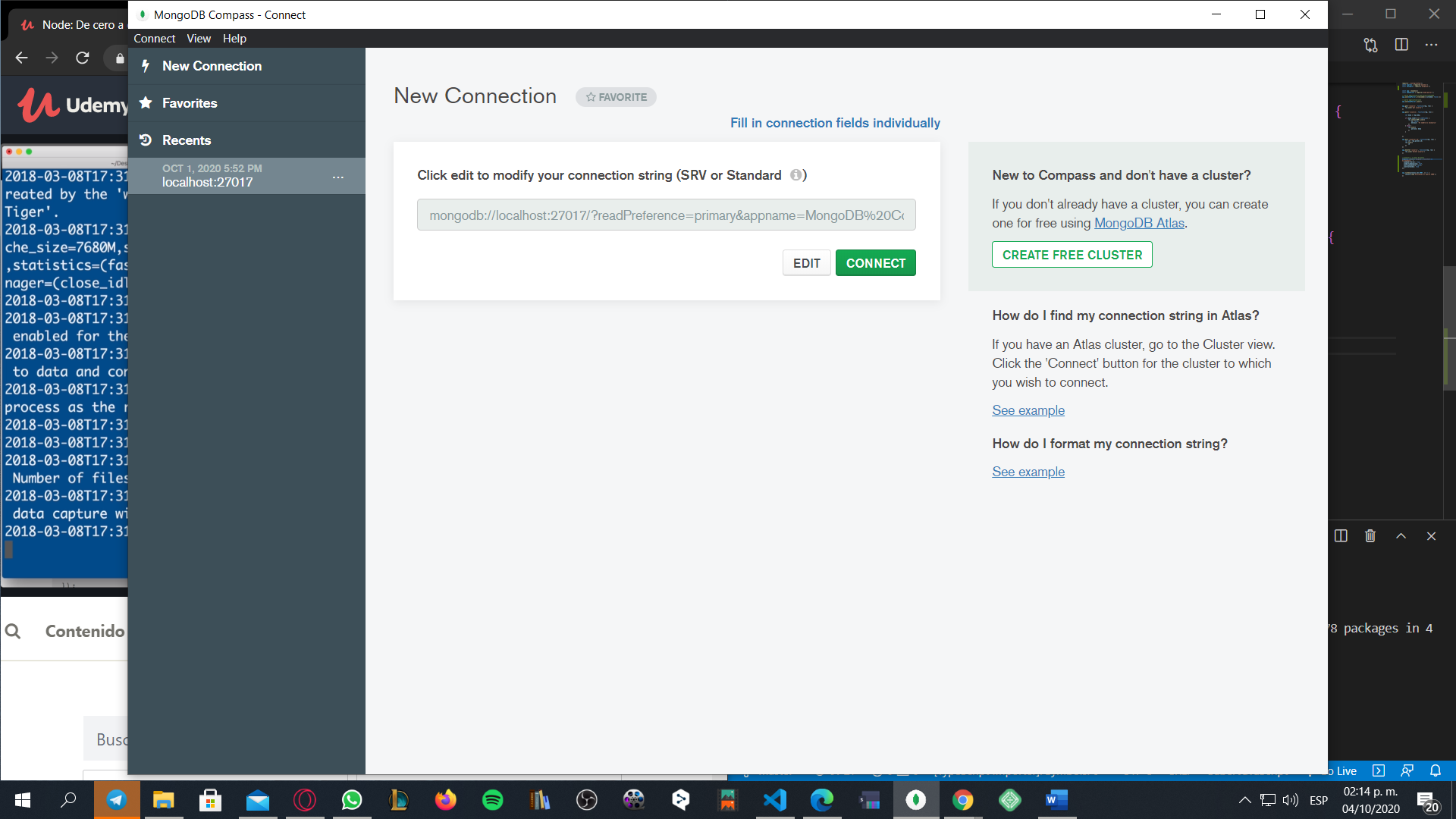
Instalamos la librería en el proyecto npm install mongoose –save

Lo exportamos en el server

const express = require('express')

const moongose = require('mongoose');

Saber el Puerto en el que estamos trabajando



//conectar a la base de datos

mongoose.connect('mongodb://localhost:27017/cafe',

    { useNewUrlParser: true, useUnifiedTopology: true },

    (err, res) => {

        if (err) throw err;

        console.log(`Bas de datos ONLINE`);

    });

27017 puerto de la base de datos / nombre de la base

Y ponemos un callback si hay un error y la respuesta con un clg

Aunque no exista la base de datos funciona

**ORDENAR RUTAS DE USUARIO**

Hay mucho código en nuestro archivo server, vamosa crear otra capeta dentro de server con el nombre de rutas o controladores, nosotros le ponemos usuario.js

Movimos las peticiones con sus respectivas exportaciones y al fina expotar la app.

const express = require('express')

const app = express()

app.get('/usuario', function(req, res) {

    res.json('get usuario')

})

app.post('/usuario', function(req, res) {

    let body = req.body;

    if (body.nombre === undefined) {

        res.status(400).json({

            ok: false,

            mensaje: 'El nombre es necesario'

        })

    } else {

        res.json({

            persona: body

        })

    }

})

app.put('/usuario/:id', function(req, res) {

    let id = req.params.id;

    res.json({

        id

    })

})

app.delete('/usuario', function(req, res) {

    res.json('delete usuario')

})

module.exports = app;

lo importamos al archivo del server

app.use(bodyParser.json())

app.use( require('./usuario/usuario'));

como va a crecer mucho las rutas debemos de tener un mejor control optimizando esta parte.

**MODELO DE USUARIO**

Un modelo nos permite realizar inserciones, actualizaciones, usar funciones demongo etc. Es un objeto. Creamos una capeta dentro de serve, models, con el archivo usuario.js

const mongoose = require('mongoose');

let Schema = mongoose.Schema;

let usuarioSchema = new Schema({

    nombre: {

        type: String,

        required: [true, 'el nombre es necesario']

    },

    email: {

        type: String,

        required: [true, 'el correo es necesario']

    },

    password: {

        type: String,

        required: [true, 'la contraseña es obligatoria']

    },

    img: { //no obligatoria

        type: String,

        required: false

    },

    role: { //default: 'USER\_ROLE'

type: String,

        default: 'USER\_ROLE',

    },

    estado: { //boolean

        type: Boolean,

        default: true

    },

    google: { //boolean

        type: Boolean,

        default: false

    }

});

module.exports = mongoose.model('Usuario', usuarioSchema);

cómo se llama el modelo

**POST CREANDO UN USUARIO EN LA COLECCIÓN**

**Hacemos la importación al server.js**

const app = express()

const Usuario = require('../models/usuario');

creamos instancia de ese esquema en metodo post

app.post('/usuario', function(req, res) {

    let body = req.body;

    let usuario = new Usuario({

        nombre: body.nombre,

        email: body.email,

        password: body.password,

        role: body.role

    });

Con los nombres de las variables

app.post('/usuario', function(req, res) {

    let body = req.body;

    let usuario = new Usuario({

        nombre: body.nombre,

        email: body.email,

        password: body.password,

        role: body.role

    });

//metodo save para guardarlo en la base de datos

    usuario.save((err, usuarioDB) => {

        if (err) {

            return res.status(400).json({

                ok: false,

                err

            });

        }

        res.json({

            ok: true,

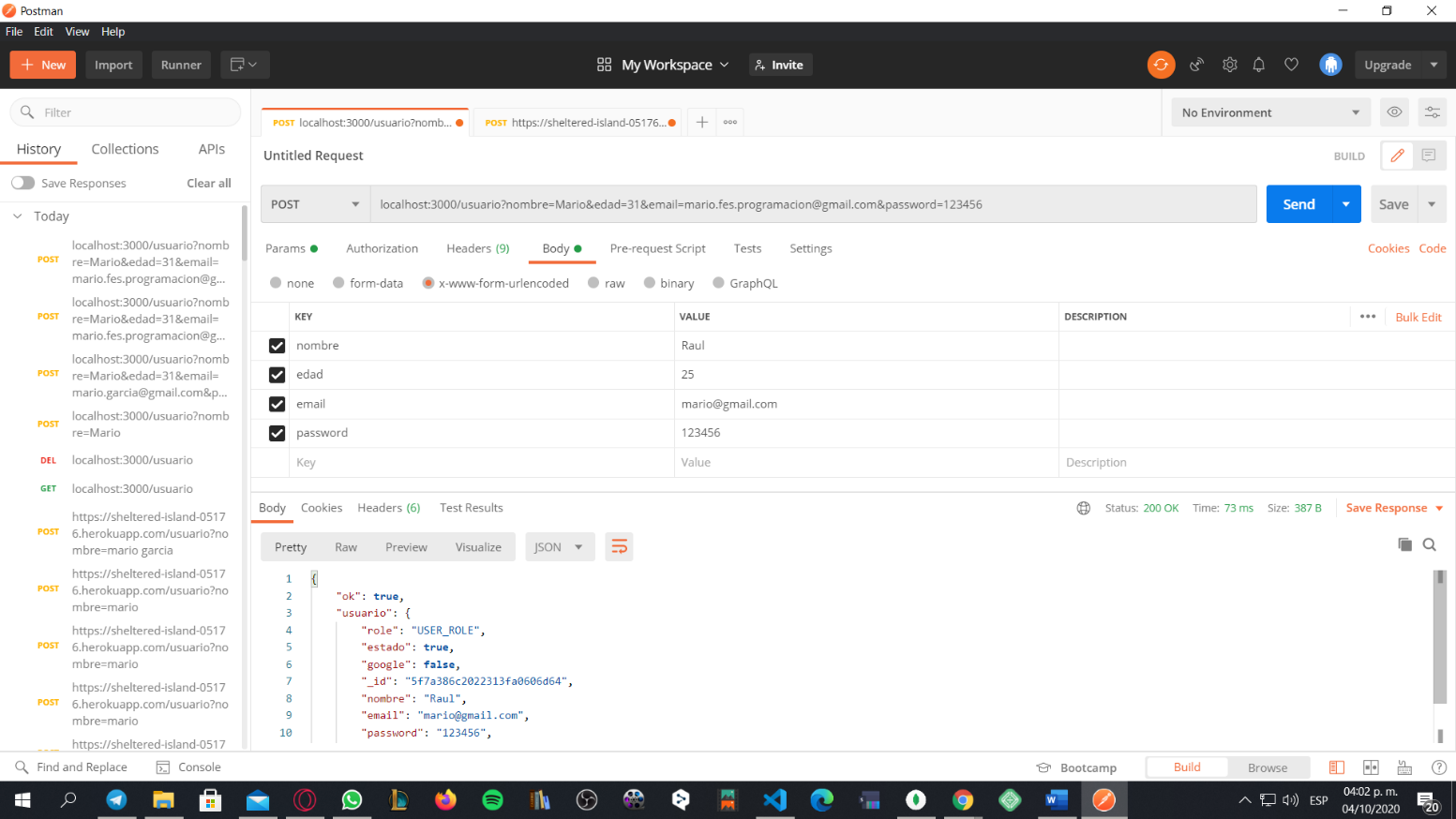
            usuario: usuarioDB

        });

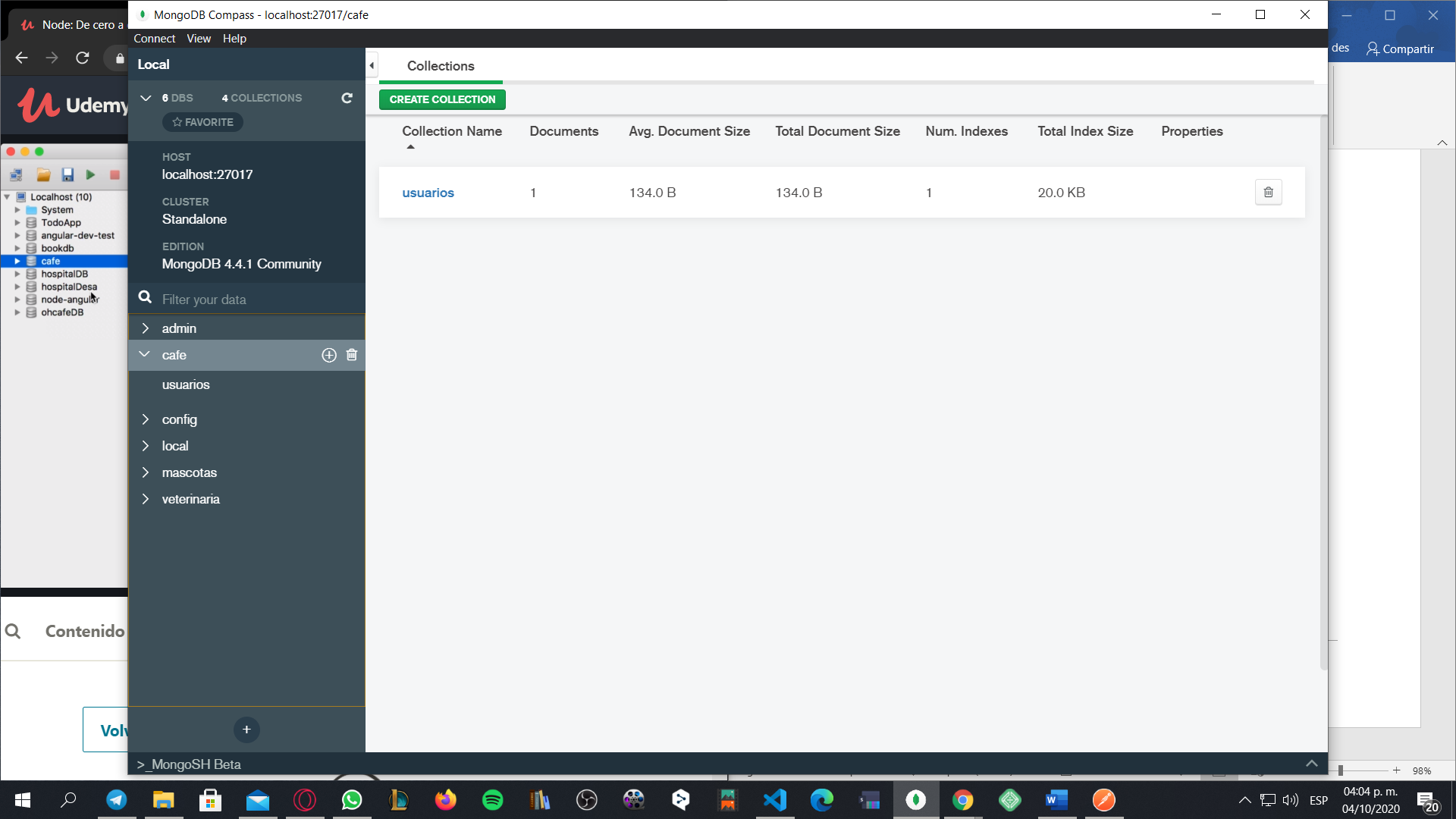
    })

})

**Probamos en postman todo ok**



**Ya se creo la base de datos**



**VALIDACIONES PERSONALIZAADAS EMAIL Y ROLE**

El problema es que si enviamos varias veces la petición la BD permite ese registo, tenemos que validad que el email sea diferente, único para cada registro. Con el campo **unique**

email: {

        type: String,

        unique: true,

        required: [true, 'el correo es necesario']

    },

Si volvemos a enviar el usuario el mensaje de error no es user friendly, tenemos que instalar un paquete **npm mongoogse-unique-validator**

Es un plugin

Lo importamos en el modelo

const mongoose = require('mongoose');

const uniqueValidator = require('mongoose-unique-validator');

y hacemos uso del pluging

usuarioSchema.plugin(uniqueValidator, {

    message: '{PATH} debe de ser único'

})

module.exports = mongoose.model('Usuario', usuarioSchema);

el resultado

{

    "ok": **false**,

    "err": {

        "errors": {

            "email": {

                "name": "ValidatorError",

                "message": "email debe de ser único",

                "properties": {

                    "message": "email debe de ser único",

                    "type": "unique",

AHORA SOLO NOS INTERESA EL AMDIN ROLE Y EL USER ROLE, no queremos que se ingrese roles que no existen.

En el modelo creamos una variable objetos con los valores y el mensaje de error, dentro de la llaves VALUE es lo que el usuario escriba

const uniqueValidator = require('mongoose-unique-validator');

let rolesValidos = {

    values: ['ADMIN','USER'],

    message : '{VALUE} no es un rol válido'

};

Y en el role con la propiedad enum solo nos permitirá esos valorea que tiene el objeto

role: { //default: 'USER\_ROLE'

        type: String,

        default: 'USER\_ROLE',

        enum: rolesValidos

ya quedo la validación(no enviamos nombre, correo y escribirmos un role diferente) de los campos y esto lo podemos usar en el front.

  "ok": **false**,

    "err": {

        "errors": {

            "nombre": {

                "name": "ValidatorError",

                "message": "el nombre es necesario",

                "properties": {

                    "message": "el nombre es necesario",

                    "type": "required",

                    "path": "nombre"

                },

                "kind": "required",

                "path": "nombre"

            },

            "email": {

                "name": "ValidatorError",

                "message": "el correo es necesario",

                "properties": {

                    "message": "el correo es necesario",

                    "type": "required",

                    "path": "email"

                },

                "kind": "required",

                "path": "email"

            },

            "role": {

                "name": "ValidatorError",

                "message": "SUPER\_ROLE no es un rol válido",

                "properties": {

                    "message": "SUPER\_ROLE no es un rol válido",

                    "type": "enum",

                    "enumValues": [

                        "ADMIN",

                        "USER"

                    ],

                    "path": "role",

                    "value": "SUPER\_ROLE"

                },

                "kind": "enum",

                "path": "role",

                "value": "SUPER\_ROLE"

            }

        },

        "\_message": "Usuario validation failed",

        "message": "Usuario validation failed: nombre: el nombre es necesario, email: el correo es necesario, role: SUPER\_ROLE no es un rol válido"

    }

**NOTA DE ACTUALIZACION BCRYPT**

En linux y OSX no hay ningún inconveniente, pero para usuarios de Windows necesitarán instalarlo con la última versión disponible en Windows así:

npm install bcrypt@3.0.2

**ENCRIPTANDO LA CONTRASEÑA**

Instalamos el paquete

**npm i bcrypt --save**

lo importamos en el archivo del server

const app = express()

const bcrypt = require('bcrypt');

y en el modelo uso la función hashSync como primer parámetro la información y segundo el numero de veces que dará el ciclo para su encriptación.

 let usuario = new Usuario({

        nombre: body.nombre,

        email: body.email,

        password: bcrypt.hashSync(body.password,10),

        role: body.role

    });

Si enviamos la petición ya sale encriptada.

No nos interesa ahora imprimir o sacar el pass. Lo hacemos con null cuando guardamos el usuario

usuario.save((err, usuarioDB) => {

        if (err) {

            return res.status(400).json({

                ok: false,

                err

            });

        }

        usuarioDB.password = null;

**PUT ACTUALIZAR INFORMACIÓIN DEL USUARIO**

Usamos el metodo findByIdAndUpdate del modelo, nos pide como parámetros el id, la data en eset caso el body, yn callback con el error y la informaicon que regresa. Pusimo el error del estatus 400 si no la regresa.

app.put('/usuario/:id', function(req, res) {

    let id = req.params.id;

    let body = req.body;

    Usuario.findByIdAndUpdate(id, body, (err, usuarioDB) => {

        if (err) {

            return res.status(400).json({

                ok: false,

                err

            });

        }

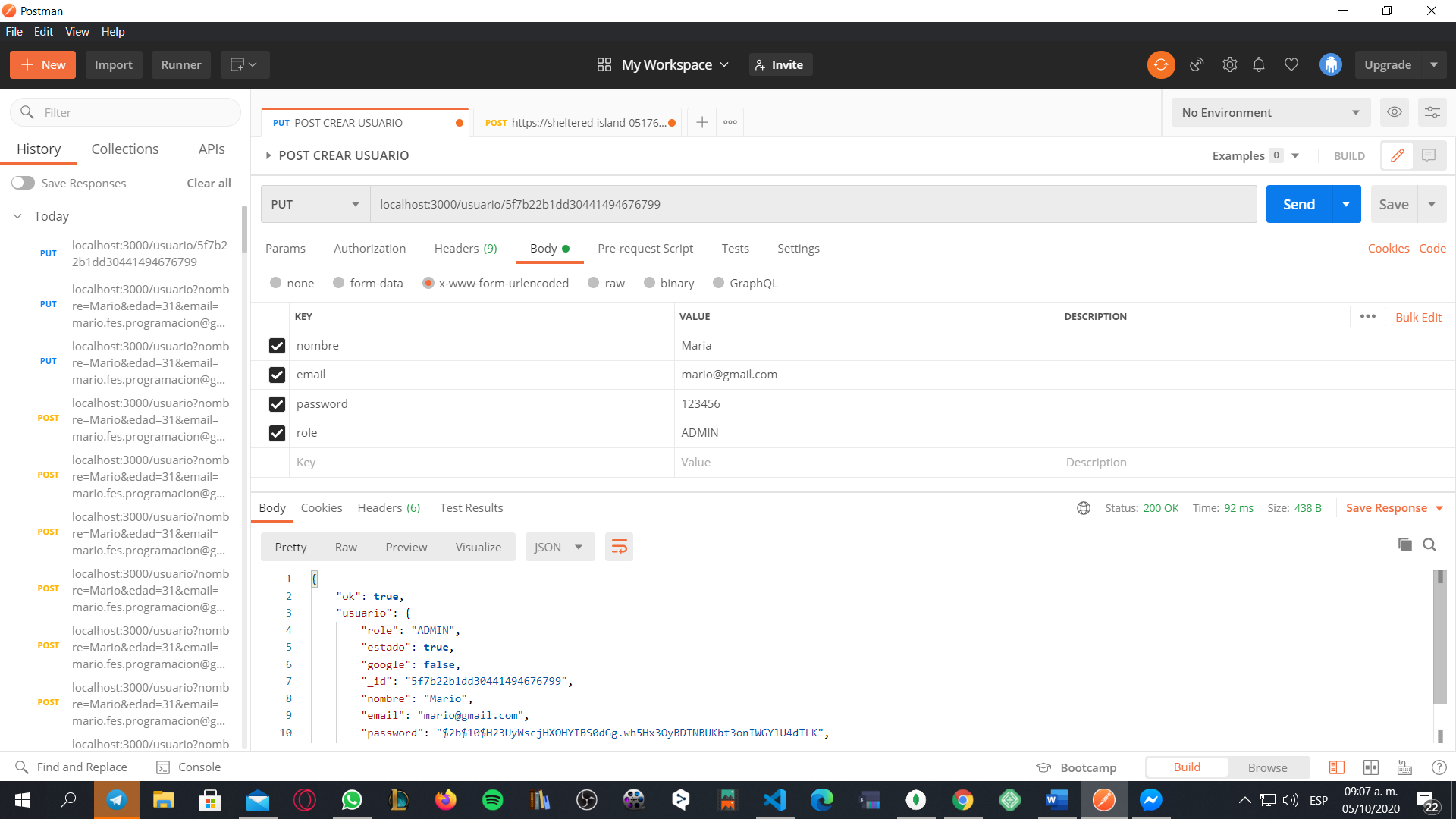
        res.json({

            ok: true,

            usuario: usuarioDB

        })

    })



Pero no actualiza en la vista solamente ne la base de datos

En la documentación de moongose en el metodo, hay un tercer parametro opciones, uno de ellos es new que nos regresa el documento modificado en vez del original.

Usuario.findByIdAndUpdate(id, body, { new: true }, (err, usuarioDB) => {

        if (err) {

El problema con put que tenemos es que el usuario puede modificar el estado de inicio de sesión de Google a true, no debemos permitir que se haga eso. Por que nosotros en el body enviamos todo el objeto con lo datos, hay que aplicar una serie de validaciones.

**VALIDACIONES ADICIONALES EN EL PUT**

Hay otro metodo en la documentación de moongose que ejecuta las validaciones ‘runValidators’ que definimos en el schema, por que si modificamos hasta los roles los cambia.

 Usuario.findByIdAndUpdate(id, body, { new: true, runValidators: true },

Ahora para que no alteren los demás campos como el de Google, la contraseña, etc vamos a usar la librería

**Underscore.js**

Npm install underscore

Hay un objeto que se llama pick que hace una copia del objeto y filtrta los valores que necesitamos.

Hacemos la importación en el archivo de las rutas

const bcrypt = require('bcrypt');

const \_ = require('underscore');

mando los campos del modleo que si quiero que se actualicen

app.put('/usuario/:id', function(req, res) {

    let id = req.params.id;

    let body =\_.pick(req.body, ['nombre','email','img','role','estado']);

**GET OBETENER TODOS LOS USUARIOS**

Creamos varios usuarios para hacer pruebas…

El método find del modelo filtra mediante un objeto, y dentro de este objeto podemos poner las reglas de lo que queremos obtener, y el metodo exec ejecuta ese find.. Manejamos el error de estatus 400 y obtenerlos la repsuesta por el callback de usuarios.

app.get('/usuario', function(req, res) {

    Usuario.find({})

        .exec((err, usuarios) => {

            if (err) {

                return res.status(400).json({

                    ok: false,

                    err

                });

            }

            res.json({

                ok: true,

                usuarios

            })

        })

})

Poniendo filtros de obtención de datos

Usuario.find({})

        .limit(5)

**PAGINACION**

Trabajando un poco en la paginación

Creamos dos variables que va a contener desde que registros queremos hasta el limite de registros que queremos que se muestren, el metodo query tiene todos los registros que se obtieen de la respuesta de get. Y tranformamos esa variable a Numero.

app.get('/usuario', function(req, res) {

    let desde = req.query.desde || 0;

    desde = Number(desde);

    let limite = req.query.limite || 5;

    limite = Number (limite);

    Usuario.find({})

        .skip(desde)

        .limit(limite)

**RETORAR NÚMERO TOTAL DE REGISTROS EN UNA COLECCIÓN**

.exec((err, usuarios) => {

            if (err) {

                return res.status(400).json({

                    ok: false,

                    err

                });

            }

            Usuario.count({},(err,conteo)=>{

                res.json({

                    ok: true,

                    usuarios,

                    cuantos: conteo

                })})})

El metodo que se usa es count que regularmente tiene los mismos filtros que usemos en find para obtenerlos.

**FILTRANDO LOS CAMPOS DE LOS RESULTADOS DE UN GET**

Solo nos interesa regresar ciertos datos por ejemplo, nombre,email y si es de Google por ejemplo…

Podemos poner el nombre de los campos que queremos que retorne cuando ejecutamos get, (deben ser los nombres tal cual están en el esquema o modelo.

 Usuario.find({},'nombre email')

**DELETE BOORRANDO UN USUARIO EN LA BASE DE DATOS**

Hay dos tipos de eliminación, borrado físico de la base de datos o eliminación virtual. Por eso en el modelo de datos usamos el estado true para indicar que existen en la base de datos.

app.delete('/usuario/:id', function(req, res) {

    let id = req.params.id;

    //borrado fisico

    Usuario.findOneAndRemove(id, (err, usuarioBorrado) => {

        if (err) {

            return res.status(400).json({

                ok: false,

                err

            });

        }

        res.json({

            ok: true,

            usuario: usuarioBorrado

        })

    })

})

Borrado virtual

app.delete('/usuario/:id', function(req, res) {

    let id = req.params.id;

    // Usuario.findByIdAndRemove(id, (err, usuarioBorrado) => {

    let cambiaEstado = {

        estado: false

    };

    Usuario.findOneAndDelete(id, cambiaEstado, { new: true }, (err, usuarioBorrado) => {

        if (err) {

            return res.status(400).json({

                ok: false,

                err

            });

        };

        if (!usuarioBorrado) {

            return res.status(400).json({

                ok: false,

                err: {

                    message: 'Usuario no encontrado'

                }

            });

        }

        res.json({

            ok: true,

            usuario: usuarioBorrado

        });

    });

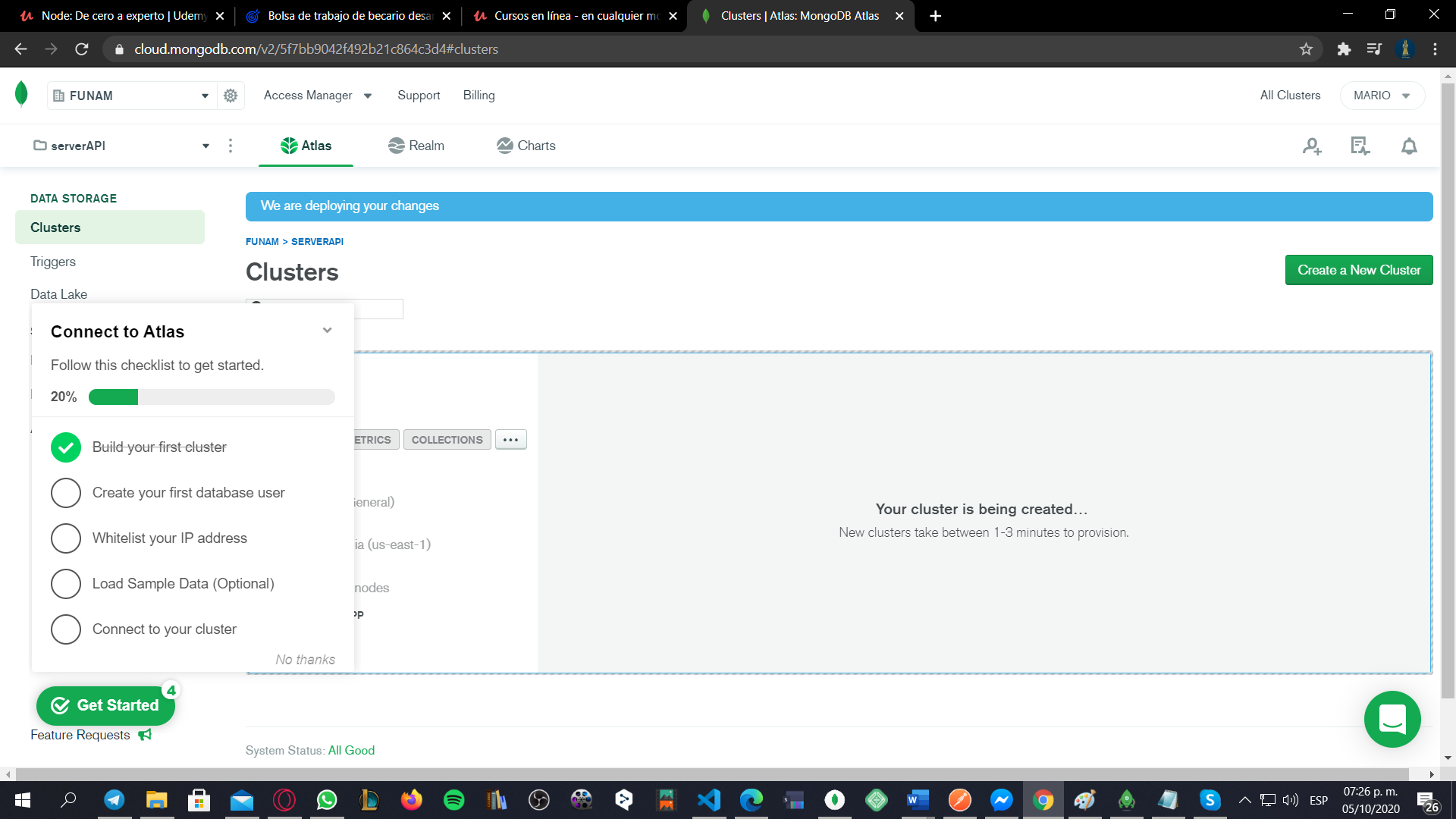
});

**MLAB: UN SERVICIO DE MONGODB EN LA NUBE**

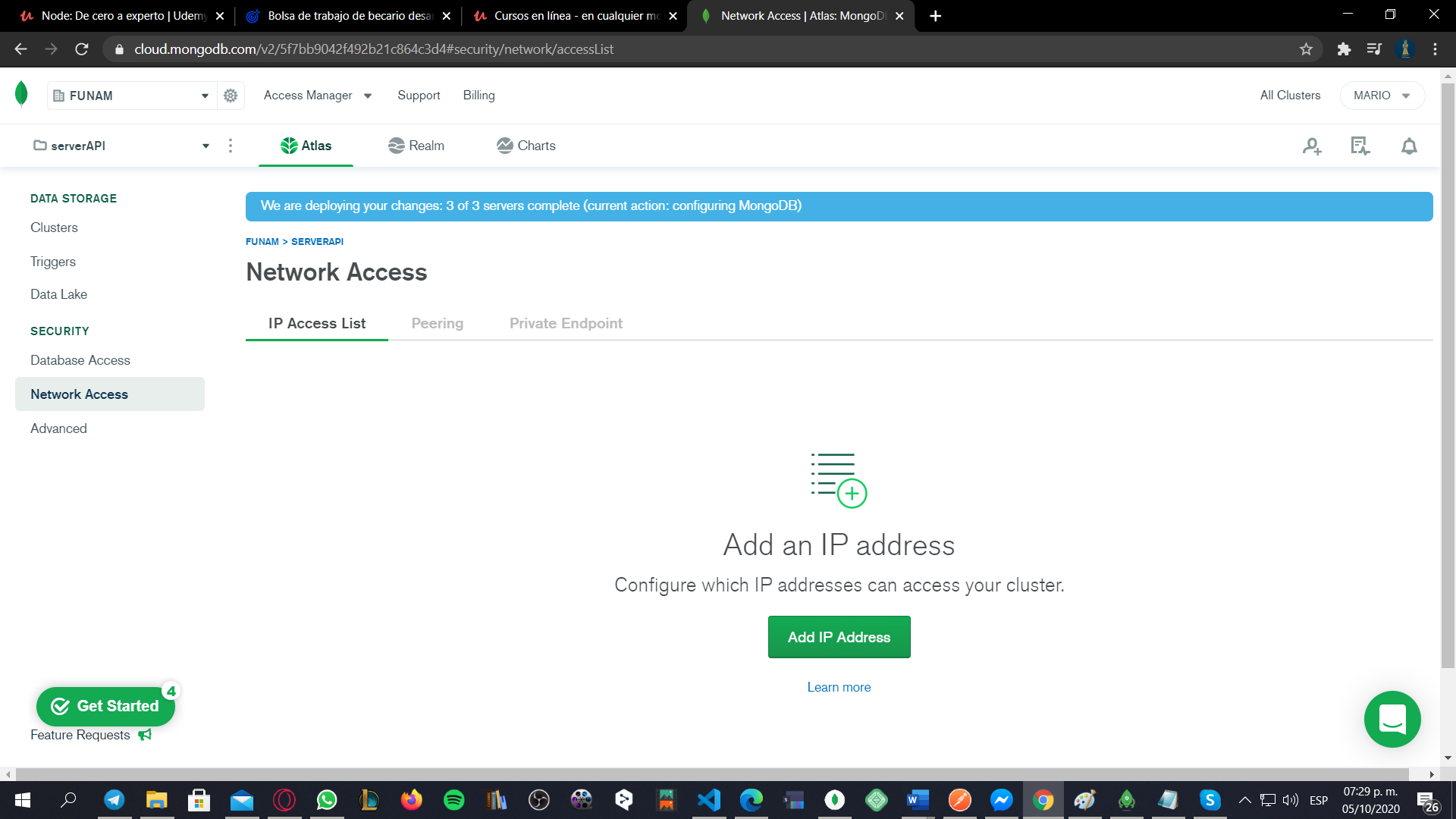
Estamos conectados a una base de datos que esta conectada localmente conectada en la maquina, existe un serviio de datos que se llama mlab ofrece servicios gratuitos hasta 500 mb

Pero ya se traslado a mongo Atlas hay que iniciar sesión ahí.

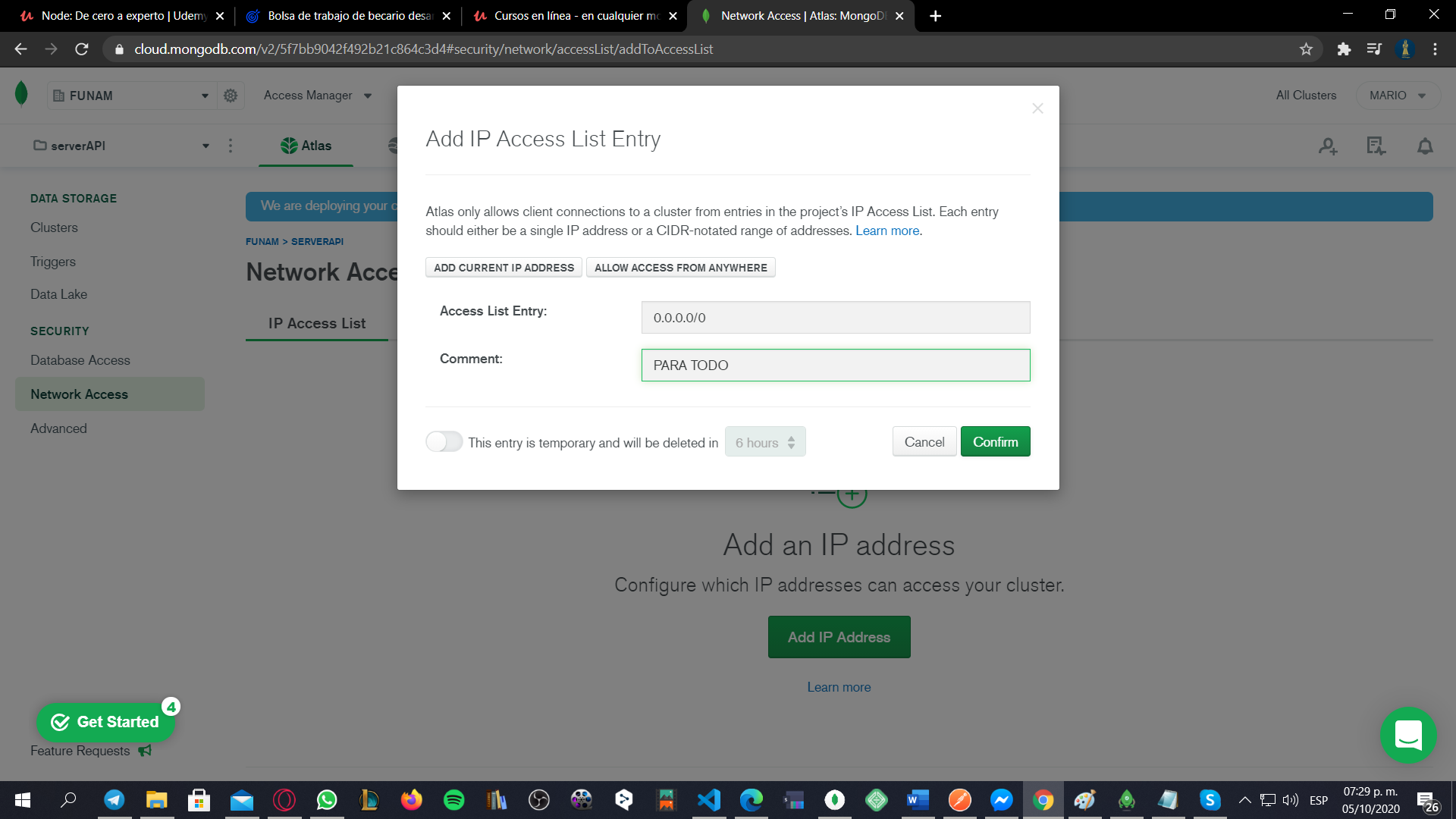
Configuración gratis



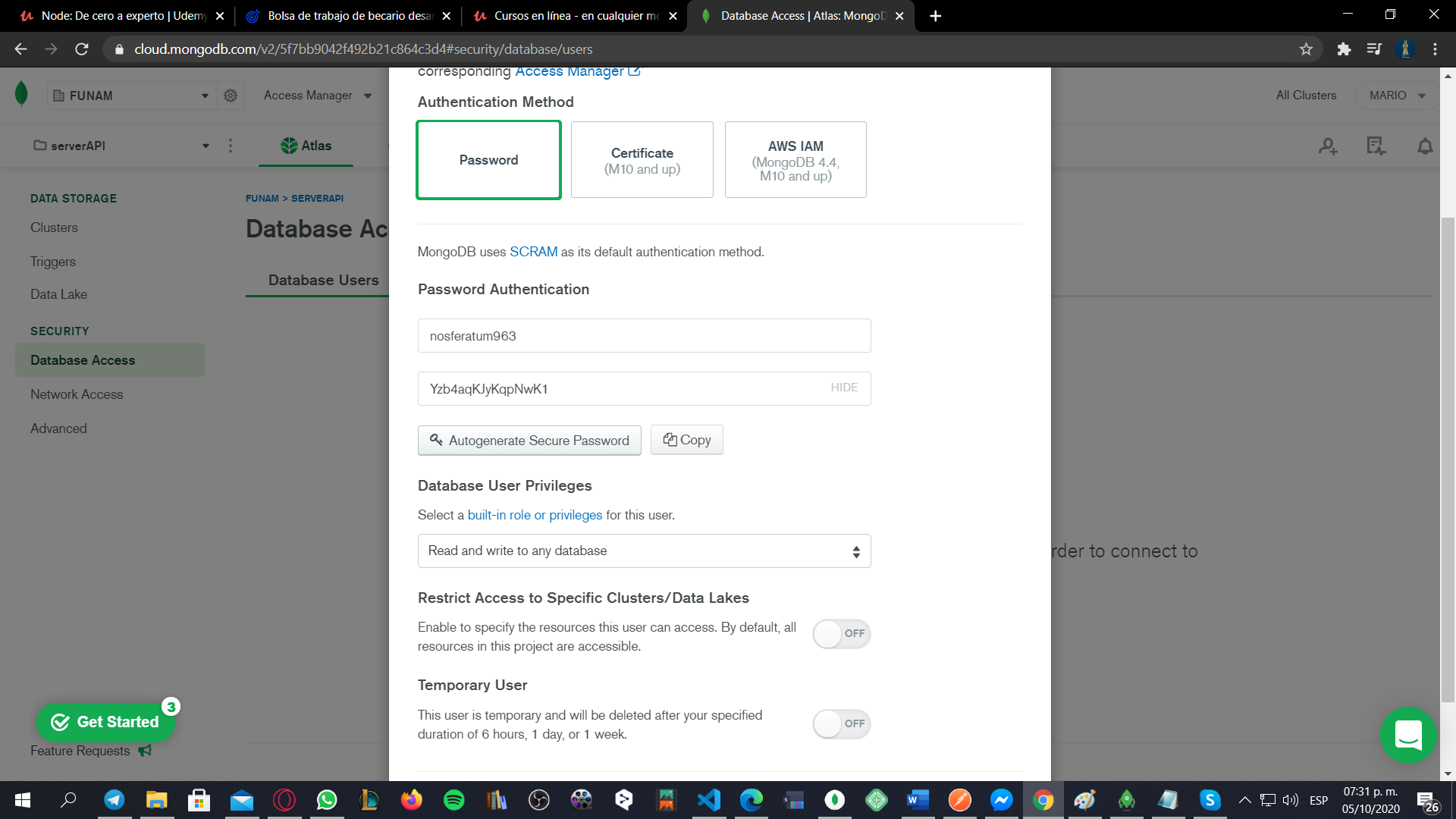
AGREGAMOS UNA IP SEGURA



Desde cualquier lugar



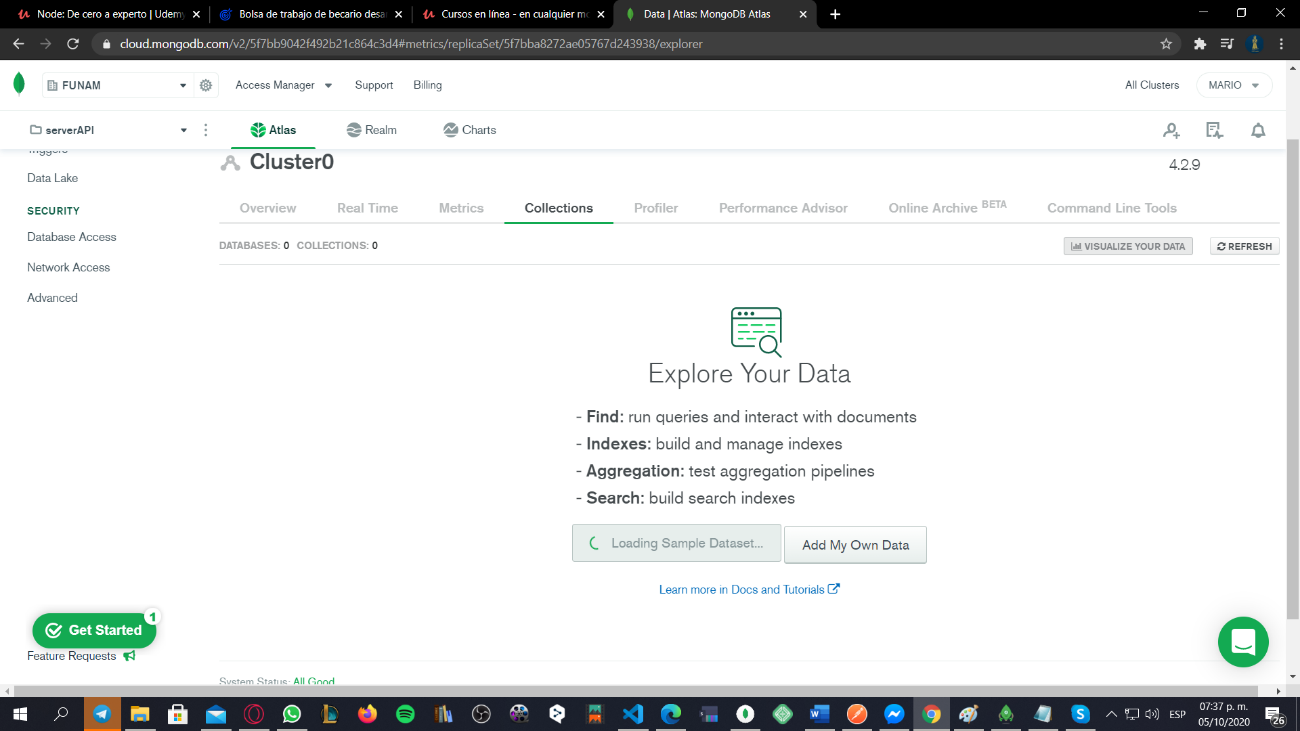
Genero un nuevo usuario



nosferatum963

Yzb4aqKJyKqpNwK1

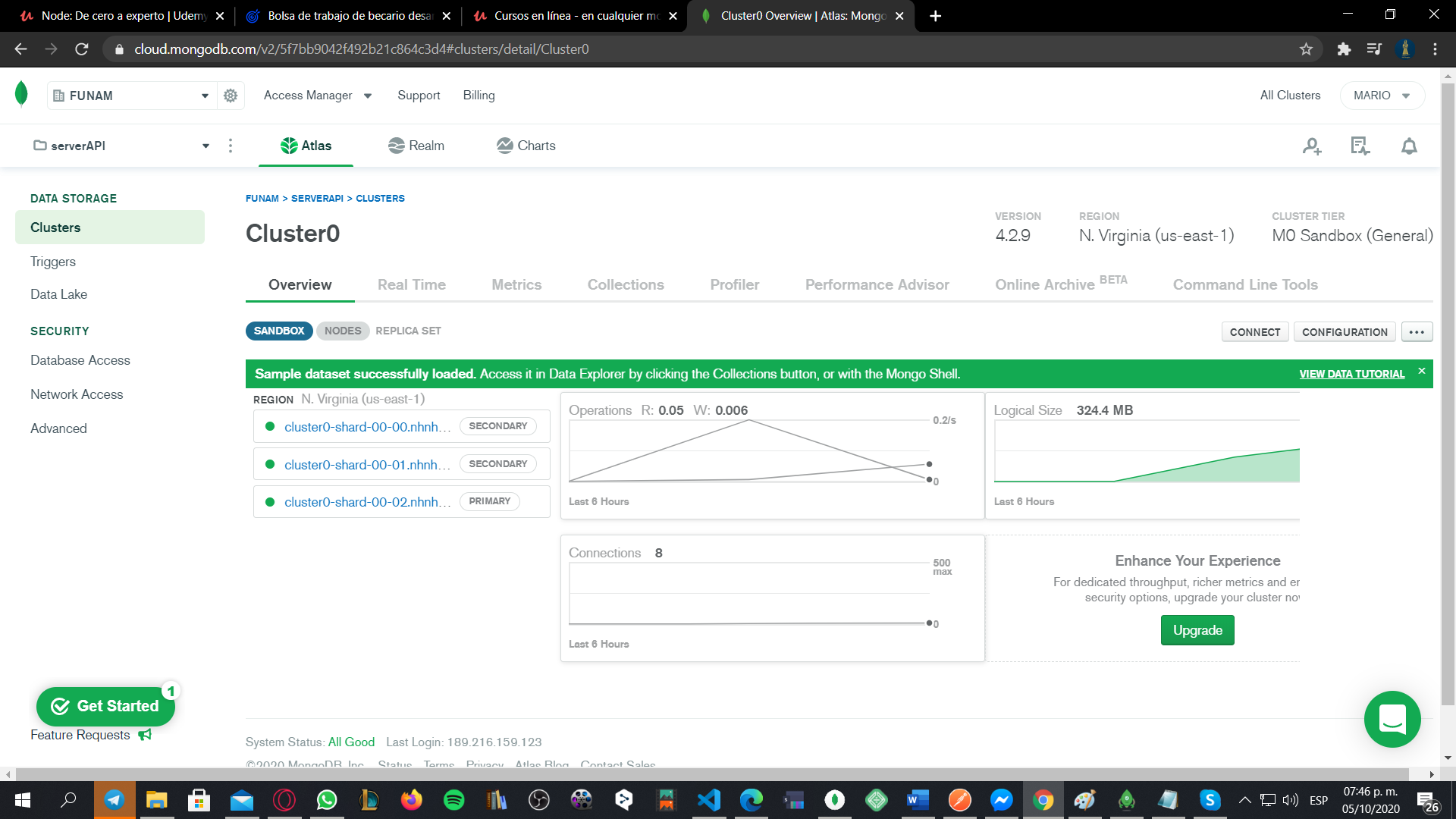
Tipo: atlas admin

**Vamos a hacer una colección para probarla localmente «primero creando una BD**

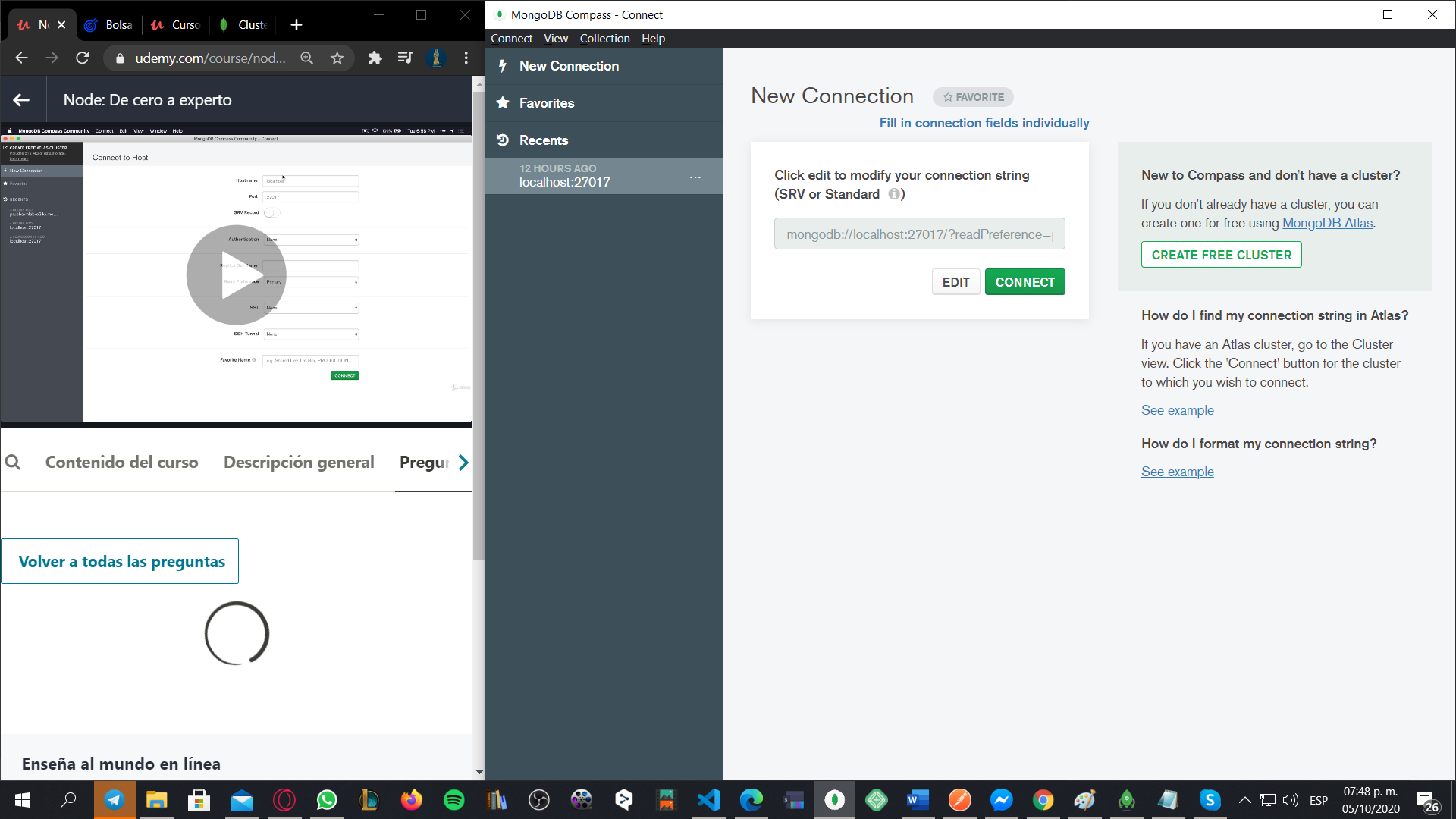
***MONGO COMPASS UI PARA EL MANEJO DE MONGODB***

Compass es mejor para bases de datos locales y atlas

Al inicio en la opción conectar elegimos db compass



En compass seleccionamos nueva conexión y hacemos las configuraciones respectivas



Tenemos que regresas a atlas nos da el link para conectarnos

mongodb+srv://nosferatum963: Yzb4aqKJyKqpNwK1@cluster0.nhnhj.mongodb.net/cafe

tenemos que poner la contraseña y la base de datos

cluster0.nhnhj.mongodb.net es el host para compass

llenamos los datos respectivos y listo!

Ya los dos están sincronizados

Vamos a ver si es compatible la ultima actualización de moongose para ver si soporta la autentificación SRV

Y actualizar todos los paquetes de Node