NODE MICROSERVICIOS GRAPHQL - 01 Introducción a Microservicios

- Tengo tres servers
    - Products
    - Sales
    - Users
- Ejemplo de products:
- src/controllers/products.controller.ts

```
import { Request, Response } from "express";

export class ProductsController{

    constructor(){}

    public getAllProducts = async(req:Request,res:Response)=>{
        return res.status(200).json({
            message: "OK",
            data:{
                products:{
                    name: "Piedra de Móstoles",
                    price: 20000,
                    quantity: 1
                }
            }
        })
    }
}
```

- src/presentation/routes.ts

```
import { Router } from 'express';
import { ProductsController } from '../controllers/products.controller';



export class ProductRoutes {


  static get routes(): Router {
```

```
    const router = Router();
    const productsController= new ProductsController()


    router.get('/all', productsController.getAllProducts)

    return router;
  }


}
```

- products/src/presentation/server.ts

```
import express, { Router } from 'express';


interface Options {
  port: number;
  routes: Router;
  public_path?: string;
}


export class Server {

  public readonly app = express();
  private serverListener?: any;
  private readonly port: number;
  private readonly publicPath: string;
  private readonly routes: Router;

  constructor(options: Options) {
    const { port, routes, public_path = 'public' } = options;
    this.port = port;
    this.publicPath = public_path;
    this.routes = routes;
  }



  async start() {


    //* Middlewares
    this.app.use( express.json() ); // raw
    this.app.use( express.urlencoded({ extended: true }) ); // x-www-form-
urlencoded

    //* Public Folder
    this.app.use( express.static( this.publicPath ) );
```

```
    //* Routes
    this.app.use( "/api/v1/products", this.routes );



    this.serverListener = this.app.listen(this.port, () => {
      console.log(`Server running on port ${ this.port }`);
    });

  }

  public close() {
    this.serverListener?.close();
  }

}
```

- products/src/app.ts

```
import { envs } from './config/envs';
import { ProductRoutes } from './presentation/routes';
import { Server } from './presentation/server';


(async()=> {
  main();
})();


function main() {

  const server = new Server({
    port: envs.PORT,
    routes: ProductRoutes.routes,
  });

  server.start();
}
```

- src/config/envs.ts

```
import 'dotenv/config';
import { get } from 'env-var';


export const envs = {

  PORT: get('PORT').required().asPortNumber(),
```

```
}
```

.env

```
PORT=3001
```

- Creo un api-gateway (otro módulo sin servicio, solo app, server y controller)
- **NOTA**: cada microservicio es un módulo ajeno al otro, con su propio json.config, .env, etc y su propio puerto de escucha
    - Haremos que ninguno se expongan al exterior, que el punto de entrada sea el api-gateway
- Creo un método post en el server de api-gateway
- gateway/src/presentation/server.ts

```
  async start() {


    //* Middlewares
    this.app.use( express.json() ); //para trabajar con json en el body
    this.app.use( express.urlencoded({ extended: true }) ); // x-www-form-
urlencoded
    this.app.use(cors())

    //* Public Folder
    this.app.use( express.static( this.publicPath ) );

    //* Routes
    this.app.use( "/api/v1", this.routes );

    //* SPA /^\/(?!api).*/  <== Únicamente si no empieza con la palabra api
    this.app.get('*', (req, res) => {
      const indexPath = path.join( __dirname + `../../../${ this.publicPath
}/index.html` );
      res.sendFile(indexPath);
    });


    //ESTE SERÁ MI MICROSERVICIO
    this.app.post("/api/v1", gateWayController.getAll)

    this.serverListener = this.app.listen(this.port, () => {
      console.log(`Server running on port ${ this.port }`);
    });

  }
```

- En el gateway.controller evalúo que venga el event

- Si el evento coincide con el string de PRODUCTS_GET_ALL uso axios para hacer la petición get al microservicio de products
- Si no capturo el error con un try catch (importante usar un try catch con axios en typeScript!)

```
import axios from "axios";
import { Request, Response } from "express";

export class gatewayController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event} = req.body

        if(!event){

            return res.status(400).json({message: "Event is required"})
        }

        if(event.trim()=== 'PRODUCTS_GET_ALL'){

            try {
            const {data} = await
axios.get('http://localhost:3001/api/v1/products/all')
            res.status(200).json({
                message: "Success!",
                data

            })
            } catch (error) {
              console.log(error)
              return res.status(500).json({
                message: "Error with axios"
              })
            }
            }

    }
}
```

- En la petición con POSTMAN/THUNDERCLIENT apunto al gateway

```
http://localhost:3000/api/v1/all
```

- En el body coloco esto

```
{
  "event": "PRODUCTS_GET_ALL"
}
```

- Me devuelve esto (apuntando al al endpoint del api-gateway donde hago un get con axios al microservcicio de products)

```
{
  "message": "Success!",
  "data": {
    "message": "OK",
    "data": {
      "products": {
        "name": "Toyota",
        "price": 2000,
        "quantity": 1
      }
    }
  }
}
```

- El gateway sirve para verificar que traemos el evento y posiblemente la data
- Si hubiera que hacer todas estas verificaciones (con trim, etc) para cada evento sería muy tedioso
- Para ello usaremos un **Event Broker** donde recibir el PRODUCT_GET_ALL y similares y redirigir la petición al microservicio adecuado tan solo cambiando una parte de la url con axios

---

Event Broker

- Creamos una carpeta independiente (fuera del API gateway y los microservicios)
- Puedo copiar el mismo api-gateway y renombrarlo a event-broker
- Le pongo el puerto 3001 y cambio el de products a 3002, y el resto
- El api-gateway se va a comunicar directamente con el event-broker
- Cambio el endpoint /all a events
- event-broker/src/presentation/server.ts

```
import { Router } from 'express';
import { Eventontroller } from '../controllers/gateway.controller';



export class AppRoutes {


  static get routes(): Router {
```

```
    const router = Router();

    router.use('/events', EventBrokerController.getAll)


    return router;
  }

}
```

- En el controller desestructuro el event y la data (que puede o no venir)
- Creo un archivo enum para los GET_ALL_PRODUCTS y demás
- event-broker/src/enums/products.enum.ts

```
export enum ProductsEvent{
    CREATE_PRODUCT = 'CREATE_PRODUCT',
    UPDATE_PRODUCT = 'UPDATE_PRODUCT',
    DELETE_PRODUCT = 'DELETE_PRODUCT',
    GET_PRODUCT = 'GET_PRODUCT',
    GET_PRODUCTS = 'GET_PRODUCTS'
}
```

- Ahora puedo usar el enum para las peticiones en el event-broker
- Creo un products.controller.ts en controllers/event-broker/src/controllers

```
import axios from "axios"


const productsApi = axios.create({
    baseURL: 'http://localhost:3002/products'
})

export const getAllProducts = async()=>{

    try {
        const {data} = await productsApi('/all')
        return data

    } catch (error) {
        console.log(error)
    }

}
```

- Por el momento no hago validaciones de la data
- En el event-broker.controller.ts llamo a la función

```
import axios from "axios";
import { Request, Response } from "express";
import { ProductsEvent } from "../enums/products.enum";
import { getAllProducts } from "./products.controller";

export class EventBrokerController {

    constructor(){}

    static async getAll(req: Request, res: Response){
        const {data} = await getAllProducts()


    }
}
```

- En el api-gateway ahora debo hacer una petición post y pasarle la data y el event al event-broker
- Renombro data a requestData para evitar conflictos

```
import axios from "axios";
import { Request, Response } from "express";

export class gatewayController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data: requestData} = req.body

        if(!event){

            return res.status(400).json({message: "Event is required"})
        }

          try {
            const {data} = await axios.post("http://localhost:3001/events",{
              requestData,
              event
            })
            return res.status(200).json({
              message: "Success!!",
              data
            })

          } catch (error) {
            return res.status(500).json({
              message: "Error",
              error
            })
          }
```

```
        }
}
```

- La validación/elección del event ocurre en el event-broker.controller

```
import { Request, Response } from "express";
import { getAllProducts } from "./products.controller";
import { ProductsEvent } from "../enums/products.enum";

export class EventBrokerController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data}= req.body

        console.log(event)

        if(event === ProductsEvent.GET_PRODUCTS){

                const products = await getAllProducts()
                return res.status(200).json({
                    products
                })

            }

        res.status(404).json({
            message: "Event not found!"
        })



    }
}
```

- getAllProducts es donde llamo desde products.controller de api-gateway/src/controllers/products.controller.ts con axios al endpoint /all de products.controller (del microservicio products) (con un POST)

```
import axios from "axios"


export const productsApi = axios.create({
    baseURL: 'http://localhost:3002/products'
})

export const getAllProducts = async()=>{
```

```
        const {data} = await productsApi.get('/all') //aqui uso get apuntando a
products con un get

        return data
}
```

- Apunto con un POST a http://localhost:3000/api/v1/all
- Este es el endpoint de tipo post del api-gateway que llama al gatewayController
- api-gateway/src/presentation/routes.ts

```
import { Router } from 'express';
import { gatewayController } from '../controllers/gateway.controller';



export class AppRoutes {


  static get routes(): Router {

    const router = Router();

    router.post('/all', gatewayController.getAll)



    return router;
  }
}
```

- EN el gateway-controller apunto al event-broker
- El event-broker solo tiene el endpoint events que es de tipo post

```
import axios from "axios";
import { Request, Response } from "express";

export class gatewayController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data: requestData} = req.body

        if(!event){

            return res.status(400).json({message: "Event is required"})
```

```
        }

          try {
            const {data} = await axios.post("http://localhost:3001/events",{
              requestData,
              event
            })
            return res.status(200).json({
              message: "Success!!",
              data
            })

          } catch (error) {
            return res.status(500).json({
              message: "Error",
              error
            })
          }
        }
}
```

- event-broker/src/presentation/routes.ts

```
import { Router } from 'express';
import { EventBrokerController } from '../controllers/event-broker.controller';



export class AppRoutes {


  static get routes(): Router {

    const router = Router();

    router.post('/events', EventBrokerController.getAll)



    return router;
  }
}
```

- Este endpoint apunta al event-broker.controller, que es dónde evalúo el event y sirvo los productos llamando al getAllProducts del controlador de products que tengo en el event-broker

```
import { Request, Response } from "express";
import { getAllProducts } from "./products.controller";
```

```
import { ProductsEvent } from "../enums/products.enum";

export class EventBrokerController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data}= req.body

        console.log(event)

        if(event === ProductsEvent.GET_PRODUCTS){

                const products = await getAllProducts()
                return res.status(200).json({
                    products
                })

            }

        res.status(404).json({
            message: "Event not found!"
        })

    }
}
```

- event-broker/src/controllers/products.controller.ts

```
import axios from "axios"


export const productsApi = axios.create({
    baseURL: 'http://localhost:3002/products'
})

export const getAllProducts = async()=>{

        const {data} = await productsApi.get('/all') //llamo al endpoint de
products con la instancia de axios

        return data
}
```

Enumeración Eventos

- Generaremos el evento de usuarios y después crearemos la API
- En el event-broker/src/enums/users.enum.ts

```
export enum UserEvent{
    CREATE_USER = 'CREATE_USER',
    UPDATE_USER = 'UPDATE_USER',
    DELETE_USER = 'DELETE_USER',
    GET_USER = 'GET_USER',
    GET_USERS = 'GET_USER'
}
```

- Creo también el controlador de users en el event-broker
- Usaremos funciones de flecha en vez de clases para los controllers
- El api-gateway está expuesto en http://localhost:3000/api/v1/all, desde aquí llamo al endpoint events del event-broker
- api-gatway.controller

```
import axios from "axios";
import { Request, Response } from "express";

export class gatewayController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data: requestData} = req.body

        if(!event){

            return res.status(400).json({message: "Event is required"})
        }

          try {
            const {data} = await axios.post("http://localhost:3001/events",{
              requestData,
              event
            })

            return res.status(200).json({
              message: "Success!!",
              data
            })

          } catch (error) {
            return res.status(500).json({
              message: "Error gateway",

            })
          }
        }
}
```

- este all es el que expongo como POST en api-gateway/routes.ts

```
import { Router } from 'express';
import { gatewayController } from '../controllers/gateway.controller';



export class AppRoutes {


  static get routes(): Router {

    const router = Router();

    router.post('/all', gatewayController.getAll)




    return router;
  }

}
```

- En el event-broker hago las validaciones del event y llamo al controlador que se requiere

```
import { Request, Response } from "express";
import { getAllProducts } from "./products.controller";
import { ProductsEvent } from "../enums/products.enum";
import { UserEvent } from "../enums/users.enum";
import { getAllUsers } from "./users.controller";

export class EventBrokerController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data}= req.body



        if(event === ProductsEvent.GET_PRODUCTS){

                const products = await getAllProducts()
                return res.status(200).json({
                    products
                })
```

```
            }

        if(event === UserEvent.GET_USERS){
            const users = await getAllUsers()

            return res.status(200).json({
                users
            })

        }
        res.status(500).json({
            message: "Internal Server Error - users"
        })
    }
}
```

- event-broker/src/controllers/user.controller.ts

```
import axios from 'axios'


export const usersApi = axios.create({
    baseURL: 'http://localhost:3004/users'
})


export const UsersController =async ()=>{
     const getAllUsers = async()=>{

        const {data} = await usersApi.get('/all')

        return data
}

}
```

- En el server de users (por ejemplo), marco la ruta cuando ejecuto this.routes en el microservicio de users

```
  this.app.use( "/users", this.routes );
```

- Y el /all lo marco en el routes.ts del microservicio de users

```
router.post('/all', gatewayController.getAll)
```

- Conviene usar una función para pasar el event todo a mayúsculas antes de evaluarlo

```
import axios from "axios";
import { Request, Response } from "express";

export class gatewayController {

    constructor(){}

    static async getAll(req: Request, res: Response){

        const {event, data: requestData} = req.body

        if(!event){

            return res.status(400).json({message: "Event is required"})
        }

          try {
            const {data} = await axios.post("http://localhost:3001/events",{
              requestData,
              event: event.toUpperCase() //lo paso a mayúsculas
            })

            return res.status(200).json({
              message: "Success!!",
              data
            })

          } catch (error) {
            return res.status(500).json({
              message: "Error gateway",

            })
          }
        }
}
```

Microservicio Sales

- Creo en event-broker/src/enums/sales.enum.ts

```
export enum SalesEvent{
    CREATE_SALES = 'CREATE_SALES',
    UPDATE_SALES = 'UPDATE_SALES',
    DELETE_SALES = 'DELETE_SALES',
    GET_SALE = 'GET_SALE',
    GET_SALES = 'GET_SALES'
}
```

- A parte de crear una ruta para todos, quiero crear una ruta para crear una venta
- sales/src/presentation/routes.ts

```
import { Router } from 'express';
import { SalesController } from '../controllers/sales.controller';


const salesController = new SalesController()

export class SalesRoutes {
     get routes(): Router {

    const router = Router();
    const productsController= new SalesController()


    router.get('/all', salesController.getAll)
    router.post('create', salesController.createSale)

    return router;
  }

}
```

- Apunto a http://localhost:3003/sales/create
- sales/src/controller/sales.controller.ts

```
import { Request, Response } from "express";


interface Sale{
    user: Object,
    product: Object,
    quantity: number,
    price: number
}


export class SalesController{



    public  getAll= async(req:Request,res:Response)=>{
        return res.status(200).json({
            message: "OK",
            data:{
                sales:{
                    user: {},
                    product: {},
```

```
                    quantity: 0,
                    price: 0
                }
            }
        })
    }

    public createSale(req: Request, res: Response){
        const sales = []
        const {data}: any = req.body

        const {uid, product_id, quantity} = data

        const sale: Sale = {
            user: {},
            product: {},
            quantity,
            price: 0

        }
        sales.push(sale)

        return res.status(200).json({
            message: "OK!",
            sale
        })


    }
}
```

- En el sales server tengo

```
import express, { Router } from 'express';
import cors from 'cors'

interface Options {
  port: number;
  routes: Router;
  public_path?: string;
}


export class Server {

  public readonly app = express();
  private serverListener?: any;
  private readonly port: number;
  private readonly publicPath: string;
  private readonly routes: Router;

  constructor(options: Options) {
```

```
    const { port, routes, public_path = 'public' } = options;
    this.port = port;
    this.publicPath = public_path;
    this.routes = routes;
  }



  async start() {


    //* Middlewares
    this.app.use(cors())
    this.app.use( express.json() ); // raw
    this.app.use( express.urlencoded({ extended: true }) ); // x-www-form-
urlencoded

    //* Public Folder
    this.app.use( express.static( this.publicPath ) );

    //* Routes
    this.app.use( "/sales", this.routes );



    this.serverListener = this.app.listen(this.port, () => {
      console.log(`Server running on port ${ this.port }`);
    });

  }

  public close() {
    this.serverListener?.close();
  }

}
```

Comunicar varios microservicios

- Instalo axios en sales
- Genero una instancia de axios como eventBroker dentro de sales.controller
- Este código es algo complejo porque estoy usando la db en memoria
- **NOTA**: este código da error. No usaremos un broker de esta manera, en memoria...usaremos RabbitMQ

```
import { Request, Response } from "express";
import axios from 'axios'



const eventBroker = axios.create({
```

```
    baseURL: 'http://localhost:3001'
})


export class SalesController{



    public  getAll= async(req:Request,res:Response)=>{
        return res.status(200).json({
            message: "OK",
            data:{
                sales:{
                    user: {},
                    product: {},
                    quantity: 0,
                    price: 0
                }
            }
        })
    }

    public async createSale(req: Request, res: Response){
        const sales = []
        const {data}: any = req.body

        const {uid, product_id, quantity} = data

        try {

            const {data: user} = await eventBroker.post(`/events`,{
                event: "GET_USERS",

            })

            const {data: product} = await eventBroker.post('/events',{
                event:'GET_PRODUCTS'
            })

            const sale = {
                user: user.users[0],
                product: product.products[0],
                quantity,
                price: {
                    unit: product[0]?.products.price, //puede no existir
                    total: product[0]?.products.price * quantity
                }

            }
            sales.push(sale)

            return res.status(200).json({
                message: "OK!",
                sale
```

```
            })


        } catch (error) {
            console.log(error)
            return res.status(500).json({
                message: "internal server error -sales.controller"
            })
        }
    }
}
```

- Cambio el SalesController a controladores de funciones de flecha

```
import axios from "axios";
import { Request, Response } from "express";

const eventBroker = axios.create({
  baseURL: "http://localhost:3001",
});

const sales: any[] = [];

export const getAll = (req: Request, res: Response) => {
  return res.status(200).json({ message: "OK", sales });
};

export const createSale = async (req: Request, res: Response) => {
  const { data } = req.body;

  const { quantity } = data;

  const { data: user } = await eventBroker.post("/events", {
    event: "GET_USERS",
  });

  const { data: product } = await eventBroker.post("/events", {
    event: "GET_PRODUCTS",
  });

  const sale = {
    user: user.data.users[0],
    product: product.data.products[0], //el objeto que retorna es data.products
    quantity,
    price: {
      unit: product.data.products[0]?.price,
      total: product.data.products[0]?.price * quantity,
    },
  };

  sales.push(sale);
```

```
  return res.status(200).json({ message: "OK", sales: sale });
};
```

- **NOTA**: pasamos directamente a Node con GraphQL usando el código del curso

---

02 NODE MICROSERVICIOS GRAPHQL - 02 Microservicios en Node con GraphQL

- **NOTA**: reiniciar los servidores en caso de error es en muchos casos necesario durante las pruebas
- Repasemos el API GATEWAY
- No es más que una aplicación REST que llama al eventBroker y devuelve la data
- EL server del api-gateway es asi

```
import express, { Request, Response } from 'express'
import axios from 'axios';
import cors from 'cors'


const app = express()

app.use( express.json() ); // raw
app.use( express.urlencoded({ extended: true }) ); // x-www-form-urlencoded
app.use(cors())
app.use(express.json)

app.post("/api/v1", async (req: Request, res: Response)=>{
  const {event, data: requestData} = req.body

  if(!event){
    return res.status(400).json({
      message: "Event is required"
    })
  }


    try {
      const {data} = await axios.post("http://localhost:3001/events",{
        requestData,
        event: event.toUpperCase()
      })

      return res.status(200).json({
        message: "Success!!",
        data
      })

    } catch (error) {
      return res.status(500).json({
```

```
        message: "Error gateway",

      })
    }
})

app.listen(process.env.PORT, ()=>{
  console.log("API GATEWAY IS RUNNING ON PORT", process.env.PORT)
})
```

- Este endpoint events de tipo POST se encuentra en el server de event-broker, que llama al eventBrokerController
- Es POST porque necesito mandarle el evento ("GET_PRODUCTS",. etc)
- Según el event se llamará al controlador que tiene la lógica, en este caso un método GET

```
import express from "express";
import morgan from "morgan";
import cors from "cors";
import dotenv from "dotenv";

import { eventBrokerController } from "./controllers/events.controller";

dotenv.config();

const app = express();
const port = process.env.PORT;

app.use(cors());
app.use(express.json());

app.use(morgan("dev"));

app.post("/events", eventBrokerController);

app.listen(port, () => {
  console.log("Event Broker is running on port:", port);
});
```

- En el controller del microservicio event-broker hago uso de los enums para filtrar por event

```
import { Request, Response } from "express";
import { ProductsEvent } from "../enums/products.enum";
import { getAllProducts } from "./products.controller";
import { UsersEvent } from "../enums/users.enum";
import { getAllUsers } from "./users.controller";
import { SalesEvent } from "../enums/sales.enum";
import { createSale } from "./sales.controller";

export const eventBrokerController = async (req: Request, res: Response) => {
```

```
  const { event, data } = req.body;

  if (event === ProductsEvent.GET_PRODUCTS) {
    const products = await getAllProducts();

    return res.status(200).json({
      data: products,
    });
  }

  if (event === UsersEvent.GET_USERS) {
    const users = await getAllUsers();

    return res.status(200).json({
      data: users,
    });
  }

  if (event === SalesEvent.CREATE_SALE) {
    const sale = await createSale(data);

    return res.status(200).json({
      data: sale,
    });
  }

  return res.status(404).json({
    message: "Event not found",
  });
};
```

- Desde aquí llamo a cada controlador creado en el eventbroker/src/controllers para cada microservicio.
- Por ejemplo el de products
- event-broker/src/controllers/products.controller

```
import axios from "axios";

const productsApi = axios.create({
  baseURL: "http://localhost:3002/products",
});

export const getAllProducts = async () => {
  const { data } = await productsApi.get("/all"); //apunta al get de products

  return data;
};
```

- En products.routes tengo esto

```
import { Router } from "express";
import { getAll } from "../controllers/products.controller";

const router = Router();

// GET: /products/all
router.get("/all", getAll);

export default router;
```

- Desde aquí hago la petición al endpoint del microservicio, por lo que solo expongo el gateway al exterior
- En el products.controller del ms products tan solo tengo un endpoint con data almacenada en memoria

```
import { Request, Response } from "express";
import { IProduct } from "../interfases/products.interface";

const products: IProduct[] = [
  {
    id: "1",
    name: "Teclado Mecanico",
    price: 150,
    description: "Teclado Mecanico",
  },
];

//aqui tengo el getAll
export const getAll = (req: Request, res: Response) => {
  return res.status(200).json({ message: "OK", products });
};
```

- El server de products es así

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import { ProductsRoutes } from "./routes";

dotenv.config();

const app = express();
const port = process.env.PORT;

app.use(cors());

app.get("/", (req, res) => {
  res.send(`Products Microservice is running: ${port}`);
});
```

```
app.use("/products", ProductsRoutes);

app.listen(port, () => {
  console.log("Products Microservice is running on port:", port);
});
```

- **Resumiendo**:
  - Expongo el puerto localhost:3000 del **api-gateway** en el endpoint /api/v1
  - Desde aqui llamo al endpoint localhost:3001/events del **event-broker**
  - En el event-broker evalúo que tipo de event me están pasando como cadena de texto
  - Tengo los diferentes strings guardados en enums para cada caso de uso
  - Desde el event-broker llamo al controlador correspondiente que apunta al endpoint del microservicio en concreto
  - Desde ese endpoint tengo otro controlador que es el que contiene la lógica del microservicio

---

GraphQL

- Borramos el api-gateway server para instalar graphQL
- Instalo con npm @apollo/server graphql
- Necesito importar **ApolloServer** y **startStandaloneServer**
- necesito que hayan typeDefs y Resolvers (al menos uno!)
- api-gateway/src/index.ts

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone";

import dotenv from "dotenv";

import { typeDefs } from "./typeDefs";
import { resolvers } from "./resolvers";

import { EventBrokerAPI } from "./datasources/eventBroker.datasource";

dotenv.config();

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

//debo desestructurar la url
const { url } = await startStandaloneServer(server, {
  listen: {
    port: parseInt(process.env.PORT),
  },
  context: async () => {
    const { cache } = server;
```

```
    return {
      dataSources: {
        eventBrokerAPI: new EventBrokerAPI({ cache }),
      },
    };
  },
});

console.log(`GraphQL API Gateway started: ${url}`);
```

- api-gateway/src/typeDefs/index.ts
- Usaremos el ejemplo de books
- api-gateway/src/typeDefs/typeDefs.ts

```
export const typeDefs = `#graphql

    type Book{
    title: String
    author: string}

    type Query{
    books: [Books]}
`
```

- En api-gateway/src/reolvers/books.resolver.ts

```
//en los resolvers no suelo tener la data así

const books = [
  {
    title: "The Awakening",
    author: "Kate Chopin",
  },
  {
    title: "City of Glass",
    author: "Paul Auster",
  },
];


export const booksQuery= {
  Query: {
    books: ()=> books
  }
}
```

- En api-gateway/src/reolvers/index.ts uso el spread para que integre todas las funciones que tengo ahi

```
import {booksQuery} from ....

export const resolvers ={
  Query:{
    ...booksQuery
  }
}
```

- Mas adelante usaremos otra estratgeia con el eventBroker
- Lo iremos optimizando

Comunicar GraphQL con el event-broker

- Instalamos con npm i **@apollo/datasource-rest**
- Creo el directorio api-gateawy/src/datasources/eventbroker.datasource.ts
- Uso override para sobreescribir la baseURL que apunta al event-broker
- Uso el método emitEvent que me pide el event (string) y la data (any, no te compliques)
- **Retorno** la petición **POST a /events** usando **this** (tengo disponible .post en el this por implementar **RESTDataSource**)
- Deben ir **dentro** de la propiedad **body**
- Esto lo que hace es enviar la petición (emitEvent, no tiene porqué llamarse así)

```
import { RESTDataSource } from "@apollo/datasource-rest";

export class EventBrokerAPI extends RESTDataSource {
  override baseURL: string = "http://localhost:3001/";

  async emitEvent(event: string, data: any) {
    return this.post("/events", { body: { event, data } });
  }
}
```

- Hay que agregar el dataSource **dentro del contexto** de GraphQL
- context es una función **async**
  - **Desestructuro el caché** del server de Apollo
  - Retorno un objeto con otro objeto llamado **datasources** que contiene el **eventBrokerAPI** con una instancia de este
  - Le paso el caché
- En api-gateway/index.ts

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone";
```

```
import dotenv from "dotenv";

import { typeDefs } from "./typeDefs";
import { resolvers } from "./resolvers";

import { EventBrokerAPI } from "./datasources/eventBroker.datasource";

dotenv.config();

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

const { url } = await startStandaloneServer(server, {
  listen: {
    port: parseInt(process.env.PORT),
  },
  context: async () => {
    const { cache } = server;

    return {
      dataSources: {
        eventBrokerAPI: new EventBrokerAPI({ cache }),
      },
    };
  },
});

console.log(`GraphQL API Gateway started: ${url}`);
```

- Ahora ya puedo usar el eventBroker!

ProductsQuery

- Creo en api-gateway/src/resolvers/products.resolvers.ts
- Creo productsQeury, que será un objeto que reúna todos mis querys
- getAllProducts es una función de flecha con lo que a mi me interesa
    - Desestructuro el event del input y el context. el guión bajo es porque el primer parámetro no me interesa

```
export const productsQuerys = {
  getAllProducts: (_, { input: { event } }, context) => {
    //console.log(event)

    return {
      name: "Teclado Mecanico",
      price: 150,
    };
```

```
  },
};
```

- Debo definir el tipo product
- Si lo tuviera todo dentro de una variable llamada typeDefs sería así
- En Query debo decirle a graphQL que getAllProducts me devolverá un objeto de tipo Product (con name y price)
- api-gateway/src/typeDefs/typeDefs.ts

```
export const typeDefs = `#graphql

  type Product {
    name: String
    price: Int
  }

  type Query {
    books: [Book]
    getAllProducts: Product
  }

`;
```

- Debo registrar el resolver!
- qpi-gateway/src/resolvers/index

```
imports (...)

export const resolvers = {
  Query: {
    ...booksQuery,
    ...productsQuery
  },

};
```

- Para hacer el query desde Apollo (en localhost:3000)

```
query {
  getAllProducts{
    name
    price
  }
}
```

- Esta no es la manera óptima de trabajar, es solo con fines didácticos
- Crearemos un input que nos permita captar la info del input y el context del getAllProducts

---

EventBrokerInput

- Debo definir un input para todos los eventos
- Cualquier evento ('GET_PRODUCTS', etc) siempre vendrá en forma de input
- Más adelante este input será más avanzado y haremos tipados más complejos
- fetAllProducts recibirá un input de tipo EventBrokerInput y retornará algo de tipo Product
- En typeDefs.ts

```
export const typeDefs = `#graphql

  input EventBrokerInput {
    event: String
    data: String

  }

  type Product {
    name: String
    price: Int
  }

  type Query {
    books: [Book]
    getAllProducts(input: EventBrokerInput): Product
  }

`;
```

- Ahora en la query debo pasarle el input
- Guardo el tipo EventBrokerInput en $input y se lo paso a getAllProducts
- En el apartado VARIABLES de ApolloServer coloco el event dentro de input

```
query getAllProducts($input: EventBorkerInput){
  getAllProducts(input: $input){
    name
    price
  }
}

//VARIABLES

{
  "input":{
    "event": "GET_PRODUCTS"
```

```
  }
}
```

- Puedo hacer un console.log al event para ver que realmente lo estoy recibiendo
- En realidad lo que queremos es solo un input, una llave de entrada y una de salida
- La emisión de un evento y ya está
- Lo hemos hecho de esta manera para entender cómo funciona

Cambiar Querys por Mutation

- Usar **un query para cada petición no es óptimo**
- Porqué usar **Mutation**?
- Si no voy a tener que estar estructurando querys, definiendo tipos...
- En el index.ts de los resolvers borro los querys que habían
- Indico que el service me retorne el string de API-GATEWAY (no hará nada más)
- resolvers/index.ts

```
import { eventBrokerMutation } from "./eventBroker.resolvers";

export const resolvers = {
  Query: {
    service: () => "API Gateway",
  }
};
```

- Para comprobar que funciona DEBO BORRAR los type Query de typeDefs.ts y colocar el service
- Le añado un **!** porque siempre lo va a retornar
- typeDefs/typeDefs.ts

```
export const typeDefs = `#graphql

  input EventBrokerInput {
    event: String
    data: String

  }

  type Product {
    name: String
    price: Int
  }

  type Query {
   service: String! ## lo coloco aquí!!
  }

`;
```

- La consulta ahora sería

```
query{
  service
}
```

- Me retorna "API Gateway"
- Defino un nuevo type Mutation
- Mutation porque puedo enviar datos, objetos vacíos, puedo mandar data y **mutarla**
- Diré que recibe un input de tipo EventBrokerInput y **siempre retornará** algo de tipo Response
- En el EventBroker hago el event será **obligatorio**, lo marco con **!**
- Si no coloco un ! **es opcional**
- El queryData será de tipo string **de momento**
- qpi-gateway/src/typeDefs/typeDefs.ts

```
export const baseTypes = `#graphql
  input QueryData {
    sales: CreateSalesInput
  }

  input EventBrokerInput {
    service: String!
    event: String!
    queryData: String ##recibe la data de la query
  }

  union response = Product

  type Query {
    service: String!
  }

  type Mutation {
    sendEvent(input: EventBrokerInput!): Response!
  }
`;
```

- Vamos a definir la Response como una **unión**
- Unión significa **une varios tipos**. La respuesta puede ser de tipo Products, Users, etc
- Lo coloco en el mismo archivo dentro del objeto graphql de typeDefs
- Creo el archivo api-gateway/src/resolvers/EventBrokerMutation.resolver.ts
- El context es con lo que nos vamos a conectar con los datasources

```
export const eventBrokerMutation = {
  sendEvent: (_, input, context)=>{
```

```
    return   {
    __typename: "Product",
    name: "Teclado Mecanico",
    price: 150,
  },

  }
}
```

- En resolvers/index.ts le paso el Mutation

```
import { eventBrokerMutation } from "./eventBroker.resolvers";

export const resolvers = {
  Query: {
    service: () => "API Gateway",
  },
  Mutation: {
    ...eventBrokerMutation,
  },
};
```

- Ahora el query es una mutation
- Al ser la respuesta de tipo Response como definimos en typeDefs, puede ser de tipo Product (y otros que no hemos definido en la union)
- Este código me devuelve el error de **Abstract type**, gql no sabe de que tipo es

```
mutation($input: EvenetBrokerInput){
  sendEvent(input: $input){
    __typename
  }
}

##VARIABLES

{
  "input":{
    "event": "GET_PRODUCTS"
  }
}
```

- Debo definir el __typename en el **EventBrokerMutation**, si no gql no sabe de que tipo es

```
export const eventBrokerMutation = {
  sendEvent: (_, input, context)=>{
    return   {
    __typename: "Product",
```

```
    name: "Teclado Mecanico",
    price: 150,
  },

  }
}
```

- Si trato de obtener el name directamente en la query me da error de Cannot query field "name", puede que sea un fragment de Product
- Para obtener el name debo usar una sintaxis concreta, como el spread operator, donde le digo toda la respuesta (...) conviertela a tipo Product

```
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Product  #Vuelve la respuesta de tipo Product
  }
}

##VARIABLES

{
  "input":{
    "event": "GET_PRODUCTS",
    "queryData": "",
    "type": "Product"
  }
}
```

- Con esto vamos a diferenciar el tipo de respuesta
- GraphQL ya crea un diccionario de estos desde ApolloServer
- Root/Mutation/sendEvent (click on)
- Me dice **Possible Types**
  - Product (name, price)
- Si en las variables desde ApolloServer coloco queryData como un objeto vacío me marca error
- Me pide que sea String como tipé anteriormente
- Necesito tipar el input de otra manera, le añado la propiedad type
  - (borro el __typename anterior del BrokerMutation)

```
export const baseTypes = `#graphql
  input QueryData {
    sales: CreateSalesInput
  }

  input EventBrokerInput {
    type: String! # añado el type forzoso
    event: String!
    queryData: QueryData
  }
```

```
  type Query {
    service: String!
  }

  type Mutation {
    sendEvent(input: EventBrokerInput!): Response!
  }
`;
```

- Pongo en duro el product a devolver

```
export const eventBrokerMutation = {
  sendEvent: (_, {input}, context)=>{
    return   {
    //__typename: "Product",
    name: "Teclado Mecanico",
    price: 150,
  },

  }
}
```

- En las VARIABLES de la query debo añadir el type
- ApolloServer

```
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Product { #Vuelve la respuesta de tipo Product
      name # Obtengo el name de Product
    }
  }
}

##VARIABLES

{
  "input":{
    "event": "GET_PRODUCTS",
    "queryData": "",
    "type": "Product"
  }
}
```

- Desestructuremos algunas cosas del input desde eventBrokerMutation
- Filtro y convierto todo a minúsculas, para buscar el tipo que inlcuya el parámetro (que también paso a minúsculas)
- Me lo devuelve como un arreglo, paso el arreglo a string con toString y le concateno una s al final

- api-gateway/src/resolvers/eventBroker.resolver.ts

```
import { typeList } from "../typelist";

export const eventBrokerMutation = {
  sendEvent: async (_, { input }, context) => {
    const { type, event, queryData } = input;

  //console.log(context)

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s"; //


    return {
      __typename: typename,
      name: "teclado mecánico",
      price: 100
    };
  },
};
```

- Creo la typeList en api-gateway/src/typeList/typeList.ts
- Puedo declarar todos los tipos que quiera en este typeList

```
export const typeList = ["Product", "Sale"];
```

- el console.log(context) me devuelve esto

```
{
  dataSources{
    eventBrokerAPI: EventBrokerAPI{
      deduplicationPromises: Map(0) {},
      httpCache: [HTTPCache],
      logger:[Object [console]],
      baseURL: "http://localhost:3001"
    }
  }
}
```

- Puedo desestructurar el dataSources en lugar de obtener el context

```
import { typeList } from "../typelist";
```

```
export const eventBrokerMutation = {
  sendEvent: async (_, { input }, { dataSources }) => {
    const { type, event, queryData } = input;

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s";


    const { data } = await dataSources.eventBrokerAPI.emitEvent(
      event,
      queryData[typename.toLowerCase()] //paso a minúsculas el typename0
    );

    //console.log(data) me devuelve la petición POST por consola

    const filteredData = {
      [typename.toLowerCase()]: data[typename.toLowerCase()], //paso el typename
de la data a minúsculas
    };

    return {
      __typename: typename,
      name: "Teclado Mecánico",
      price: 100
    };
  },
};
```

- El dataSources.eventBroker está creado en el server index.ts
- api-gateway/index.ts

```
import { ApolloServer } from "@apollo/server";
import { startStandaloneServer } from "@apollo/server/standalone";

import dotenv from "dotenv";

import { typeDefs } from "./typeDefs";
import { resolvers } from "./resolvers";

import { EventBrokerAPI } from "./datasources/eventBroker.datasource";

dotenv.config();

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

const { url } = await startStandaloneServer(server, {
```

```
  listen: {
    port: parseInt(process.env.PORT),
  },
  context: async () => {
    const { cache } = server;

    return {
      dataSources: { //aquí está!!
        eventBrokerAPI: new EventBrokerAPI({ cache }),
      },
    };
  },
});

console.log(`GraphQL API Gateway started: ${url}`);
```

- Llamo al emitEvent que he creado en api-gateway/src/datasources/eventBroker.datasource.ts

```
import { RESTDataSource } from "@apollo/datasource-rest";

export class EventBrokerAPI extends RESTDataSource {
  override baseURL: string = "http://localhost:3001/";

  async emitEvent(event: string, data: any) {
    return this.post("/events", { body: { event, data } });
  }
}
```

- Estoy enviando la petición POST a /events
- Se está conectando al eventBroker, y este se conecta al controller que llama al endpoint del microservicio y devuelve la data
- Esta puede ser una estrategia válida
- Para ello, en lugar de retornar el "Teclado mecánico" en duro, retorno un spread con la data filtarada en el EventBroker.resolver

```
import { typeList } from "../typelist";

export const eventBrokerMutation = {
  sendEvent: async (_, { input }, { dataSources }) => {
    const { type, event, queryData } = input;

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s";

    const { data } = await dataSources.eventBrokerAPI.emitEvent(
      event,
      queryData[typename.toLowerCase()]
```

```
    );

    const filteredData = {
      [typename.toLowerCase()]: data[typename.toLowerCase()],
    };

    return {
      __typename: typename,
      ...filteredData, //aquí!!
    };
  },
};
```

- Es una sola mutation y a partir de aquí puedes hacer 1000 querys si quieres
- Si hago un console.log de la data en getAllProducts del products.controller del event-broker, al hacer una petición me regresa un objeto como este
- event-broker.products.controller.ts

```
import axios from "axios";

const productsApi = axios.create({
  baseURL: "http://localhost:3002/products",
});

export const getAllProducts = async () => {
  const { data } = await productsApi.get("/all");
  console.log(data) //hago un console.log de la data!
  return data;
};
```

- Me devuelve este objeto en la consola del event-broker

```
{
  message: "OK",
  products:[
    {
      id: '1',
      name: 'Teclado Mecánico',
      price: 150,
      description: 'Teclado Mecánico'
    }
  ]
}
```

- Pero en la consola del api-gateway me devuelve **undefined** porque no está haciendo match con la respuesta (que lleva un id y una descripción)

- Si en lugar de desestructurar la data del event-broker.resolver del api-gateway lo guardo en una variable data tal que así

```
import { typeList } from "../typelist";

export const eventBrokerMutation = {
  sendEvent: async (_, { input }, { dataSources }) => {
    const { type, event, queryData } = input;

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s";

  //data sin desestructurar!
    const data  = await dataSources.eventBrokerAPI.emitEvent(
      event,
      queryData[typename.toLowerCase()]
    );

    console.log(data) //hago un console.log de la data sin desestructurar


    return {
      __typename: typename,
      ...data //esto me devuelve el name de la query en null ERROR!
              //porque no hace match
    };
  },
};
```

- Me devuelve en la consola de api-gateway este objeto

```
{products:{message: 'OK', products:[[Object]]}}
```

- Si lo paso a lowerCase y le agrego una s resuelve bien

```
console.log(data.[type.toLowerCase()+ 's'])
```

- En el events.controller renombro todos los valores de return a data:
- event-broker/src/controllers/event-broker.controller.ts

```
import { Request, Response } from "express";
import { ProductsEvent } from "../enums/products.enum";
import { getAllProducts } from "./products.controller";
```

```
import { UsersEvent } from "../enums/users.enum";
import { getAllUsers } from "./users.controller";
import { SalesEvent } from "../enums/sales.enum";
import { createSale } from "./sales.controller";

export const eventBrokerController = async (req: Request, res: Response) => {
  const { event, data } = req.body;

  if (event === ProductsEvent.GET_PRODUCTS) {
    const products = await getAllProducts();

    return res.status(200).json({
      data: products,
    });
  }

  if (event === UsersEvent.GET_USERS) {
    const users = await getAllUsers();

    return res.status(200).json({
      data: users,
    });
  }

  if (event === SalesEvent.CREATE_SALE) {
    const sale = await createSale(data);

    return res.status(200).json({
      data: sale,
    });
  }

  return res.status(404).json({
    message: "Event not found",
  });
};
```

- Ahora resuelve bien. me devuelve el objeto con name, price, id... en la consola de api-gateway
- Me devuelve un array de products
- No puedo hacer algo así como (en typeDefs.ts)

```
  union response = [Product] //revienta la aplicación
```

- Creo un nuevo type Products y tipo correctamente Product

```
export const productsTypes = `#graphql

  type Product {
    id: ID
```

```
    description: String
    name: String
    price: Int
  }


  type Products {
    products: [Product]
  }

`;
```

- Renombro a Products la union en otro archivo
- api-gateway/src/typeDefs/response.union,ts

```
export const responseUnion = `#graphql

  union Response = Products | Sales

`;
```

- Los lanzo desde el typeDefs/index.ts

```
import { productsTypes } from "./products.type";


export const typeDefs = `
  ${productsTypes}
  ${responseUnion}

`;
```

- Ahora la query debe de ser a Products

```
~~~gql
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Products  #de tipo Products!!
    __typename
  }
}

##VARIABLES

{
  "input":{
    "event": "GET_PRODUCTS",
```

```
    "queryData": "",
    "type": "products"
  }
}
```

- Ahora si pido el __typename en la query me salta el error que el type Product no es aplicable a Products
- Es cuando debo agregar la s al final del typename para que coincida con el Products, que es con lo que quiero hacer match

```
import { typeList } from "../typelist";

export const eventBrokerMutation = {
  sendEvent: async (_, { input }, { dataSources }) => {
    const { type, event, queryData } = input;

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s"; //agrégale a todos los tipos una s al final

    const { data } = await dataSources.eventBrokerAPI.emitEvent(
      event,
      queryData[typename.toLowerCase()]
    );

    const filteredData = { //lo paso a minusculas para que siempre el retrono sea
en minúsculas
      [typename.toLowerCase()]

    };

    console.log(filteredData) //En la consola api-gateway retorna
                              //{id: 1, name: 'Teclado mecánico', price: 150,
description: ...}

    return {
      __typename: typename,
      filteredData, //si pido el name en la query devuelve null desde Apollo
    };
  },
};
```

- Sin la filtered data, si regreso la data si obtengo mis productos
- Para usar la filtered data, uso propiedades computadas, como si fuera un arreglo
- Si el valor de la data de la izquierda es distinto, lo renombro para que haga match (capa extra de seguridad)

```
import { typeList } from "../typelist";
```

```
export const eventBrokerMutation = {
  sendEvent: async (_, { input }, { dataSources }) => {
    const { type, event, queryData } = input;

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s";

    const { data } = await dataSources.eventBrokerAPI.emitEvent(
      event,
      queryData[typename.toLowerCase()]
    );

    const filteredData = {  //si la data no tiene el mismo nombre que el tipo
      [typename.toLowerCase()]: data[typename.toLowerCase()],
    };

    return {
      __typename: typename,
      ...filteredData,
    };
  },
};
```

- Tengo un único resolver para hacer cualquier petición desde un solo endpoint

---

Conectando con Sales

- Cómo hago llegar este query al api-gateway
- sales/src/controllers/sales.controller

```
export const createSale = async (req: Request, res: Response) => {
  const { data } = req.body;

  const { quantity } = data;

  const { data: user } = await eventBroker.post("/events", {
    event: "GET_USERS",
  });

  const { data: product } = await eventBroker.post("/events", {
    event: "GET_PRODUCTS",
  });

  const sale = {
    user: user.data.users[0], //no es la mejor sintaxis del mundo!
    product: product.data.products[0], //el objeto de retorno es data.products
    quantity,
```

```
    price: {
      unit: product.data.products[0]?.price,
      total: product.data.products[0]?.price * quantity,
    },
  };

  sales.push(sale);

  return res.status(200).json({ message: "OK", sales: sale });
};
```

- El evento sería CREATE_SALE y la quantity 3, por ejemplo
- Desde POSTMAN a http_localhost:3001/events (si me conecto directamente al broker)

```
{
  "event":"CREATE_SALE",
  "data":{
    "quantity": 10
  }
}
```

- Esto me devuelve un objeto con message: OK, el objeto sale que contiene
  - user, con id, name, email
  - product, que contiene id, name, price, description
  - quantity
  - price, con unit y total
- Esta petición (y respuesta) hay que transformarla a graphQL, una vez comprobado que me conecto correctamente al endpoint

---

Query Global

- Yo debo poder enviarle en VARIABLES del ApolloServer dentro de la queryData la quantity

```
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Products  #de tipo Products!!
      products{
        name
      }
  }
}

##VARIABLES

{
  "input":{
    "event": "CREATE_SALE",
```

```
    "queryData": {
      "quantity": 10 #con esto debería ser suficiente
    },
    "type": "sale"
  }
}
```

- Para poder enviar info y decirle que me devuelva estos tipos, aceptar una información, etc
- Hay que tener en cuenta que el tipo **Query** y el tipo **Mutation** siempre **deben de existir**
- Debo crear un input,
- en api-gateway/src/typeDefs/base.type.ts

```
export const baseTypes = `#graphql
  input QueryData {
    sales: CreateSalesInput
  }

  input EventBrokerInput {
    type: String!
    event: String!
    queryData: QueryData
  }

  type Query {
    service: String!
  }

  type Mutation {
    sendEvent(input: EventBrokerInput!): Response!
  }
`;
```

- NOTA: en las peticiones GET no se debe poner el ! para marcar como obligatorio el retorno
- En el archivo input.type.ts del mismo directorio indico la quantity como un Int y obligatorio

```
export const inputsTypes = `#graphql

  input CreateSalesInput {
    quantity: Int!
  }

`;
```

- En typeList debe estar el tipo Sale
- api-gateway/src/typlist/index.ts

```
export const typeList = ["Product", "Sale"];
```

- La query quedaría asi

```
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Sales
      sales{
          product{
            name
            price
          }
      }
      quantity
  }
}

##VARIABLES

{
  "input":{
    "event": "CREATE_SALE",
    "queryData": {
      "sale":{
        "quantity": 10 #con esto debería ser suficiente
      }
    },
    "type": "sale"
  }
}
```

- Ojo que la queryData renombro queryData al type que le estoy pasando

```
import { typeList } from "../typelist";

export const eventBrokerMutation = {
  sendEvent: async (_, { input }, { dataSources }) => {
    const { type, event, queryData } = input;

    const typename =
      typeList
        .filter((t) => type.toLowerCase().includes(t.toLowerCase()))
        .toString() + "s";

    const { data } = await dataSources.eventBrokerAPI.emitEvent(
      event,
      queryData[typename.toLowerCase()] //aqui! le asigno el type como nombre de
lo que vamos a mandar
    );
```

```
    const filteredData = {
      [typename.toLowerCase()]: data[typename.toLowerCase()],
    };

    return {
      __typename: typename,
      ...filteredData,
    };
  },
};
```

- Con esto la API ya se puede conectar a lo que se quiera
- Creo el tipo Price en api-gateway/src/typeDefs/price.type.ts que viene en el tipo Sale
- Primero el sales.types.ts

```
export const salesTypes = `#graphql

  type Sale {
    product: Product
    quantity: Int
    price: Price
  }

  type Sales {
    sales: Sale
  }

`;
```

- price.type.ts

```
export const pricesTypes = `#graphql

  type Price {
    unit: Int
    total: Int
  }

`;
```

- Lo lanzo a todos desde el index.ts de typeDefs

```
import { baseTypes } from "./base.type";
import { inputsTypes } from "./inputs.type";
import { pricesTypes } from "./prices.type";
import { productsTypes } from "./products.type";
```

```
import { responseUnion } from "./response.union";
import { salesTypes } from "./sales.type";

export const typeDefs = `
  ${productsTypes}

  ${pricesTypes}

  ${salesTypes}

  ${inputsTypes}

  ${responseUnion}

  ${baseTypes}
`;
```

- Creo una query más compleja

```
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Sales
      sales{
          product{
            name
            price
            description
          }
          quantity
          price{
            total
            unit
          }
      }

  }
}

##VARIABLES

{
  "input":{
    "event": "CREATE_SALE",
    "queryData": {
      "sales":{
        "quantity": 10 #con esto debería ser suficiente
      }
    },
    "type": "sale"
  }
}
```

- En lugar de devolver sales, devuelvo sale en el controller del ms sales
- En el controlador de sales solo extraigo la quantity de la data, no el uid, ni el product_id que me retornan undefined en la consola
- sales/src/controllers/sales.controller.ts

```
import axios from "axios";
import { Request, Response } from "express";

const eventBroker = axios.create({
  baseURL: "http://localhost:3001",
});

const sales: any[] = [];

export const getAll = (req: Request, res: Response) => {
  return res.status(200).json({ message: "OK", sales });
};

export const createSale = async (req: Request, res: Response) => {
  const { data } = req.body;

  const { quantity } = data;

  const { data: user } = await eventBroker.post("/events", {
    event: "GET_USERS",
  });

  const { data: product } = await eventBroker.post("/events", {
    event: "GET_PRODUCTS",
  });

  const sale = {
    user: user.data.users[0],
    product: product.data.products[0],
    quantity,
    price: {
      unit: product.data.products[0]?.price,
      total: product.data.products[0]?.price * quantity,
    },
  };

  sales.push(sale);

  return res.status(200).json({ message: "OK", sales: sale }); //En lugar de un
arreglo devuelvo la venta en singular
};
```

- Lo de añadir una s podemos hacerlo un standard para que sea versátil con sales y sale

- Si solo quiero regresar uno no le voy a decir que regrese sales
- Lo tipo en el union response
- api-gateway/src/typeDefs/response.union.ts

```
export const responseUnion = `#graphql

  union Response = Products | Sales

`;
```

- En base.type.ts tengo esto
- api-gateway/src/types/base.type.ts

```
export const baseTypes = `#graphql
  input QueryData {
    sales: CreateSalesInput
  }

  input EventBrokerInput {
    type: String!
    event: String!
    queryData: QueryData
  }

  type Query {
    service: String!
  }

  type Mutation {
    sendEvent(input: EventBrokerInput!): Response!
  }
`;
```

NODE MICROSERVICIOS GRAPHQL - 03 Auth

- Creo el módulo de auth
- Copio el package.json de users y uso npm i
- Copio también el .env (cambio el PORT a 3005) y el tsconfig
- Creo un server normalito
- Vamos a manejar la autenticación con middlewares
- auth/src/index.ts

```
import path from "node:path";
import express from "express";
import cors from "cors";
```

```
import dotenv from "dotenv";

import { AuthRoutes } from "./routes";

import { connectionDB } from "./config/db.config";

dotenv.config();

const app = express();
const port = process.env.PORT;

app.use(cors());
app.use(express.json());
app.use(express.static("public"));

//connectionDB();

app.use("/auth", AuthRoutes);

app.listen(port, () => {
  console.log("Auth Microservice is running on port:", port);
});
```

- En AuthRoutes

```
import { Router } from "express";

import {
  validateEmailMiddleware,
  validateEmailRegistryMiddleware,
} from "../middlewares/validateEmail.middleware";

import {
  hashPasswordMiddleware,
  validatePasswordMiddleware,
} from "../middlewares/hashPassword.middleware";

import {
  verifyGoogleIdTokenMiddleware,
  verifyJWTMiddleware,
} from "../middlewares/verifyJWT.middleware";

import {
  googleSSO,
  login,
  register,
  renewToken,
} from "../controllers/auth.controller";

const router = Router();

router.post(
```

```
  "/register",
  [validateEmailMiddleware, hashPasswordMiddleware],
  register
);

router.post(
  "/login",
  [validateEmailRegistryMiddleware, validatePasswordMiddleware],
  login
);

router.post("/google", [verifyGoogleIdTokenMiddleware], googleSSO);

router.post("/renew-jwt", [verifyJWTMiddleware], renewToken);

export default router;
```

- Haremos la autenticación con google en el siguiente módulo de lecciones

---

Docker y MongoDB

- Creo el docker-compose.yml a nivel de raíz (no en src)

```
version: "3.9"
services:
  mongo:
    image: mongo:latest
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: root
      MONGO_INITDB_DATABASE: auth
    ports:
      - 27017:27017
    volumes:
      - ./database:/data/db  # persistencia para guardar credenciales

  mongo-express:  # servicio para gestionar visualmente mongo en el navegador
    image: mongo-express:latest
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: root
      ME_CONFIG_MONGODB_ADMINPASSWORD: root
      ME_CONFIG_MONGODB_URL: mongodb://root:root@mongo:27017/
```

- Creo la database auth desde el navegador desde **http://localhost:8081/db**

Conectando microservicio con MongoDB

- Usaremos mongoose, npm i mongoose
- Creo la carpeta config/db.config.ts para la configuración de la DB

```
import mongoose from "mongoose";

export const connectionDB = async () => {
  try {
    await mongoose.connect(process.env.DB_CONNECTION);

    console.log("DB Connected");
  } catch (error) {
    console.log("Connection error");
  }
};
```

- En .env tengo la url de la DB
- Aádo authSource=admin para añadir autenticación a la conexión

```
DB_CONNECTION="mongodb://root:root@localhost:27017/auth?authSource=admin"
```

- Llamo a la función connectionDB en el server index.ts

User Model

- Creo la carpeta auth/src/models/user.model.ts
- Lo que vamos a exponer es el modelo User
- El trim en true lo que hace es limpiar cadenas de texto vacias
- Sin el required no es obligatorio

```
import { Schema, model } from "mongoose";

const userSchema = new Schema({
  username: {
    required: true,
    type: String,
    trim: true,
  },
  email: {
    unique: true,
    required: true,
    type: String,
    trim: true,
  },
  password: {
    required: true,
    type: String,
    trim: true,
```

```
  },
  active: {
    type: Boolean,
    default: true,
  },
  google: {
    type: Boolean,
    default: false,
  },
  createdAt: {
    type: Date,
    default: Date.now(),
  },
  updatedAt: {
    type: Date,
    default: Date.now(),
  },
});

export default model("User", userSchema);
```

Registrando usuario

- Creo auth/src/controllers/auth.controller.ts
- Extraigo la data del body
- Creo el usuario, lo salvo
- Le paso el id al token
- Creo la respuesta con el ok en true y usando la data del usuario

```
import { Request, Response } from "express";

import User from "../models/user.model";
import { jwtSign } from "../helpers/jwt.helper";

export const register = async (req: Request, res: Response) => {
  const { email, username, password } = req.body;

  try {
    const user = new User({ username, password, email });

    await user.save();

  //jwtSign.helper.ts
    const token = jwtSign({
      id: user.id,
    });

    return res.status(200).json({
      ok: true,
      message: "User registration successful",
```

```
      user: {
        id: user.id,
        email: user.email,
        username: user.username,
      },
      jwt: token,
    });
  } catch (error) {
    console.log(`Error find: ${error}`);

    return res.status(500).json({
      error,
      ok: false,
    });
  }
};
```

Hasheando el password

- Lo haremos con un middleware
- Creo auth/src/middlewares/hashPassword.middleware.ts
- Le paso next como parámetro para que el middleware, una vez haga el trabajo, continue

```
import { NextFunction, Request, Response } from "express";

import { comparePassword, hashPassword } from "../helpers/hashPassword.helper";
import User from "../models/user.model";

export const hashPasswordMiddleware = (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const { password } = req.body;

  if (!password) {
    return res.status(403).json({
      ok: false,
      message: "password is required",
    });
  }

  const hash = hashPassword(password); //hasheo del password

  req.body.password = hash; //se lo paso a la propiedad password del body

  next();
};
```

- En src/helpers/hashPassword.helper.ts

```
import bcrypt from "bcryptjs";

export const hashPassword = (plainText: string) => {
  const salt = bcrypt.genSaltSync(12);
  const hash = bcrypt.hashSync(plainText, salt);

  return hash;
};
```

- El helper jwtSign es asi

```
import jwt from "jsonwebtoken";

//guardo esta interfaz en src/interfaces
export interface IUserPayload {
  id: string;
}

export const jwtSign = (payload: IUserPayload) => {
  const token = jwt.sign(payload, process.env.JWT_SECRET, {
    expiresIn: "1h",
  });

  return token;
};
```

- En .env especifico la variable de entornp

```
JWT_SECRET="mys3crE7K3yW0rD"
```

- Hago uso de los middlewares en AuthRoutes
- auth/src/routes/authRoutes.routes.ts

```
import { Router } from "express";

import {
  hashPasswordMiddleware,
  validatePasswordMiddleware,
} from "../middlewares/hashPassword.middleware";



import {
  register,
} from "../controllers/auth.controller";
```

```
const router = Router();

router.post(
  "/register",
  [validateEmailMiddleware, hashPasswordMiddleware],
  register
);

export default router;
```

---

validar JWT

- Creamos un nuevo middleware para validar el token
- Si hago un console.log del token que paso desde POSTMAN Auth, Bearer Token hay un espacio en blanco entre Bearer y el token

```
import { NextFunction, Request, Response } from "express";

import { jwtVerify} from "../helpers/jwt.helper";

export const verifyJWTMiddleware = (
  req: any | Request, //coloco any para salvar el error
  res: Response,
  next: NextFunction
) => {
  const authorization = req.headers.authorization; //extraigo la authorization de
los headers

  if (!authorization) {
    return res.status(403).json({
      ok: false,
      message: "Token is required",
    });
  }

  try {
    const token = authorization.split(" ")[1];//separo por espacios, me quedo con
la segunda posición

    const { id }: any = jwtVerify(token); //creo este helper para verificar el
token

    req.uid = id; //le paso el uid a la request

    next(); //llamo a next
  } catch (error) {
    return res.status(500).json({ ok: false, error });
  }
};
```

- El helper en auth/src/helpers/jwt.helper.ts

```
import jwt from "jsonwebtoken";


export const jwtVerify = (token: string) => {
  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET);

    return payload;
  } catch (error) {
    throw error;
  }
};
```

- Una vez validado el token voy a querer renovar el token
- Quiero que cada petición que se me haga traiga un nuevo token, antes verifico el usuario
- Hago que pase por el middleware de verificarJWT y le paso el controller al endpoint

```
export const renewToken = (req: Request | any, res: Response) => {
  const { uid } = req; //se lo añadí a la req en verifyJWTMiddleware
 //genero el nuevo token
    id: uid,
  });

  return res.status(200).json({ ok: true, jwt: token });
};
```

- En el auth.controller

```
router.post("/renew-jwt", [verifyJWTMiddleware], renewToken);
```

- el login no va a necesitar verificar el token, pero si deberemos obtener un token, retornarlo y en función de ese token hacer otras peticiones

---

Login

- El login hace uso de **dos middlewares**
- Primero valido que el usuario exista con validateEmailregistryMiddleware
- auth/src/validateEmail.middleware.ts

```
export const validateEmailRegistryMiddleware = async (
  req: Request | any,
  res: Response,
```

```
  next: NextFunction
) => {
  const { email } = req.body;

  const user = await User.findOne({ email });

  if (!user || !user.active) {
    return res.status(403).json({
      ok: false,
      message: "Email or password invalid",
    });
  }

  req.uid = user.id;

  next();
};
```

- En validatePassword extraigo el uid de la Request y el password del body
- Valido si no viene el password y si no encuentra el user o esta inactivo
- Verifico el password
- Valido si el password es correcto
- auth/src/middlewares/hashPassword.middleware.ts

```
export const validatePasswordMiddleware = async (
  req: Request | any,
  res: Response,
  next: NextFunction
) => {
  const { password } = req.body;
  const uid = req.uid;

  if (!password) {
    return res.status(403).json({
      ok: false,
      message: "password is required",
    });
  }

  const user = await User.findById(uid);

  if (!user || !user.active) {
    return res.status(404).json({
      ok: false,
      message: "Email or password invalid",
    });
  }

  if (user.google) {
    return res.status(403).json({
      ok: false,
```

```
      message: "User must log in with google",
    });
  }

  const hash = user.password;
                                //helper
  const result = comparePassword(password, hash);

  if (!result) {
    return res.status(403).json({
      ok: false,
      message: "Email or password invalid",
    });
  }

  next();
};

//el helper comparePassword de /helpers/hashPassword.helper.ts
export const comparePassword = (plaintText: string, hash: string) => {
  return bcrypt.compareSync(plaintText, hash);
};
```

- En AuthRoutes coloco los middlewares

```
router.post(
  "/login",
  [validateEmailRegistryMiddleware, validatePasswordMiddleware],
  login
);
```

- En el login extraigo el uid
- Genero un nuevo token, lo devuelvo en la response
- Ya he verificado email y el password con los middlewares
- auth/src/controllers/auth.controller

```
export const login = async (req: Request | any, res: Response) => {
  const uid = req.uid;

  try {
    const user = await User.findById(uid);

    const token = jwtSign({
      id: uid,
    });

    return res.status(200).json({
      ok: true,
      message: "User Login successful",
```

```
      user: {
        id: user.id,
        email: user.email,
        username: user.username,
      },
      jwt: token,
    });
  } catch (error) {
    console.log(`Error find: ${error}`);

    return res.status(500).json({
      error,
      ok: false,
    });
  }
};
```

Google Sign in

- En AuthRoutes

```
router.post("/google", [verifyGoogleIdTokenMiddleware], googleSSO);
```

- El middleware

```
export const verifyGoogleIdTokenMiddleware = async (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const { token } = req.body; //extraigo el token del body (ver el html)

  if (!token) {
    return res
      .status(403)
      .json({ ok: false, message: "User not authenticated" });
  }

  const googleUser = await googleVerify(token);//uso el helper

  if (!googleUser) {
    return res
      .status(403)
      .json({ ok: false, message: "User not authenticated" });
  }

  req.body.email = googleUser.email; //le paso el email y username en el body para
usar en el controller
```

```
  req.body.username = googleUser.username;

  next();
};
```

- El helper

```
import { OAuth2Client } from "google-auth-library";

import { IUserPayload } from "../interfaces/IUserPayload.interface";

export const googleVerify = async (token: string) => {
  const client = new OAuth2Client();

  const ticket = await client.verifyIdToken({
    idToken: token,
    audience: process.env.GOOGLE_CLIENT_ID,
  });

  const payload = ticket.getPayload();

  return { email: payload.email, username: payload.name };
};
```

- El controller

```
export const googleSSO = async (req: Request, res: Response) => {
  const { email, username } = req.body; //extraigo el email y username que pasé
desde el middleware

  try {
    let user = await User.findOne({ email });

    if (!user) {
      user = new User({ username, password: ":v", email, google: true });
      await user.save();
    }

    const token = jwtSign({
      id: user.id,
    });

    return res.status(200).json({
      ok: true,
      message: "User signin google",
      user: {
        id: user.id,
        email: user.email,
        username: user.username,
```

```
      },
      jwt: token,
    });
  } catch (error) {
    console.log(`Error find: ${error}`);

    return res.status(500).json({
      error,
      ok: false,
    });
  }
};
```

- Proveedor de botón google sign in: https://developers.google.com/identity/gs/web/guides/overview/
- Abrir página credenciales de la consola API de google
- No hay que pagar
- En el botón top left selecciono el proyecto o genero uno nuevo con el botón PROYECTO NUEVO
- Seguir el proceso en https://developers.google.com/identity/gs/web/guides/get-google-api-clientid
- clicar Crear credenciales / clicar ID de cliente OAuth, seleccioinar aplicación Web en tipo de aplicación
    - Está en la página de consola API Google, con el poyecto de Node seleccionado
    - Confirmo la pantalla de consentimiento. Selecciono usuario interno (intranet), para que cualquiera se pueda conectar **elijo externo**
    - Relleno la información (nombre del proyecto, mi correo, si quiero un lgotip, dominios autorizados, etc)
- Una vez hecho este proceso (ahora si) le doy a CREAR CREDENCIALES en la pantalla de consola API de Google con el proyecto seleccionado
- Me da las variables ID client y Secret client. Descargo el JSON (trae todas mis credenciales)
- Coloco en .env GOOGLE_CLIENT_ID y GOOGLE_SECRET_KEY
- El botón de login y logout en html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Google SignIn</title>

    <script src="https://accounts.google.com/gsi/client" async defer></script>
  </head>
  <body>
    <h1>Google SSO</h1>

    <div
      id="g_id_onload"
      data-client_id="303570499103-
sjs3kb3k0313ldgc0ih7khhdi87iq9k7.apps.googleusercontent.com"
      data-auto_prompt="false"
      data-callback="handleCredentialResponse"
    ></div>
```

```
    <div
      class="g_id_signin"
      data-type="standard"
      data-size="large"
      data-theme="outline"
      data-text="sign_in_with"
      data-shape="rectangular"
      data-logo_alignment="left"
    ></div>

    <button onclick="logout()">Log out</button>

    <script>

      async function handleCredentialResponse(response) {
        const token = response.credential; <!--//extraigo el token-->

        const res = await fetch("/auth/google", { <!--//apunto a mi endpoint-->
          method: "POST",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify({ token }), <!--//paso el token al body como un
string-->
        });

        const data = await res.json(); <!--//extraigo la data de la response-->

        localStorage.setItem("email", data.user.email); <!--// paso el email al
localstorage-->
      }

      async function logout() { <!--//creo el logout-->
        const account = google.accounts.id;

        account.disableAutoSelect();

        const email = localStorage.getItem("email");

        account.revoke(email, (res) => {
          if (res.successful) {
            localStorage.clear();
            location.reload();
          }
        });
      }
    </script>
  </body>
</html>
```

NODE MICROSERVICIOS GQL - 04 RabbitMQ

- Para usar RabbitMQ haré uso de Docker
- RabbitMQ no es la única opción, podriamos usar NATS entre otros
- RabbitMQ es un servidor de mensajería asíncrona que puede trabajar como un balanceador de carga
- Se usa para tener una comunicación fácil entre microservicios
- Tiene muchas herramientas y plugins para extender su funcionalidad, para monitorear

```
version: '3.9'
services:
  rabbitmq:
    container_name: rabbitmq
    image: rabbitmq:3.12.4-management-alpine
    ports:
      - 5672:5672
      - 15672:15672 # para el management desde el navegador
    environment:
      - RABBITMQ_DEFAULT_PASS=admin
      - RABBITMQ_DEFAULT_USER=admin
```

- Corriendo la imagen de Docker, coloco en el navegador htt://localhost:15672
- Pongo user admin, password: admin
- Puedo ver que Prometheus está en 15692, lo veremos más adelante (no expongo el cuerpo en el archivo .yml)
- Los exchanges son los diferentes puentes por los que podemos pasar
- Usaremos **AMQP** que srive por defecto
- Puedo crear mi propio exchange
- Desde el management puedo crear usuarios, configuraciones varias, etc
- RabbitMQ Permite desacoplamiento entre microservicios y una alta escabilidad
- Si tengo una data JSON, esta va a venir en formato Buffer
    - Esto quiere decir que para mandar data voy a tener que serializar la data a string y luego convertirla a Buffer
- Permite un enrutamiento flexible

Comenzando el proyecto

- Normalmente se usa una infraestructura ya creada
- Vamos a configurarlo de manera manual, da mucho control pero se complica a medida que se avanza
- La forma recomendable para hacer esto es con **Nest**
- **AMQP** es la librería que usaremos para usar RabbitMQ

```
npm i amqplib
```

- La versión 1.0 no tiene nada que ver con la 0.9.1 que es la que usaremos (la recomendable)
- En la documentación veremos que hay una API que funciona con promesas (callbacks) y otra API que lo hace con async await
- En el index del eventbroker tengo expuesto el endpoint post "/events" con el controlador
- En el controlador están los enums y los llamados a los microservicios

- Puedo trabajar de otra manera con amqp
- En lugar de tener el código acoplado a endpoints (que pueden cambiar) me comunico directamente con los eventos usando RabbitMQ

Capturando el evento

- Hago una copia del proyecto de Node para poder modificarlo para usar RabbitMQ
- Voy al api-gateway y lo pongo en marcha
- En el api-gateway tengo el service que me devuelve un string
- Hago la query en Apollo Server localhost:3000
- Lo tengo en roots/Query

```
query ExampleQuery{
  service # devuelve "API GATEWAY"
}
```

- Puedo quitar la capa de controladores en el eventBroker y hacer la lógica directamente desde allí
- event-broker.controller

```
import { Request, Response } from "express";
import { ProductsEvent } from "../enums/products.enum";
import { getAllProducts } from "./products.controller";
import { UsersEvent } from "../enums/users.enum";
import { getAllUsers } from "./users.controller";
import { SalesEvent } from "../enums/sales.enum";
import { createSale } from "./sales.controller";

export const eventBrokerController = async (req: Request, res: Response) => {
  const { event, data } = req.body;

  if (event === ProductsEvent.GET_PRODUCTS) {
    const products = await getAllProducts(); //quitar este controlador y añadir
aquí la lógica

    return res.status(200).json({
      data: products,
    });
  }

  if (event === UsersEvent.GET_USERS) {
    const users = await getAllUsers();//quitar este controlador y añadir aquí la
lógica

    return res.status(200).json({
      data: users,
    });
  }
```

```
  if (event === SalesEvent.CREATE_SALE) {
    const sale = await createSale(data);//quitar este controlador y añadir aquí la
lógica

    return res.status(200).json({
      data: sale,
    });
  }

  return res.status(404).json({
    message: "Event not found",
  });
};
```

- Borro todo y dejo solo esto

```
import { Request, Response } from "express";


export const eventBrokerController = async (req: Request, res: Response) => {
  const { event, data } = req.body;

  //console.log(event)

  return res.status(404).json({
    message: "Event not found",
  });
};
```

- Compruebo que recibo el event con una query de CREATE_SALE, por ejemplo
- Debo tener los microservicios correspondientes corriendo

```
mutation($input: EventBrokerInput!){
  sendEvent(input: $input){
    ... on Sales
      sales{
          product{
            name
            price
            description
          }
          quantity
          price{
            total
            unit
          }
      }

  }
```

```
}

##VARIABLES

{
  "input":{
    "event": "CREATE_SALE",
    "queryData": {
      "sales":{
        "quantity": 10 #con esto debería ser suficiente
      }
    },
    "type": "sale"
  }
}
```

- Usaremos este CREATE_SALE para generar una cola con RabbitMQ que todavía no vamos a conectar, pero lo visualizaremos en el panel de RabbitMQ
- RabbitMQ en el navegador

> localhost:15672

Comunicar microservicios con queues

- Usaremos amqplib, lo instalamos en el event-broker
- Instalo los tipos también

> npm i - D @types/amqplib @types/cors @types/dotenv

- Debo crear la conexión con amqplib
- Para usar amqplib recuerda que debe de ser async
- Mi queue es el evento (string)
- Voy a tener que crear un canal de comunicación, una vez se termina la comunicación se destruye
- event-broker/src/controllers/eventBroker.controller

```
import { Request, Response } from "express";
import amqplib from 'amqplib'

const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

export const eventBrokerController = async (req: Request, res: Response) => {
  const { event, data } = req.body;

  const channel= await connection.createChannel()
  await channel.assertQueue(event)
```

```
  return res.status(404).json({
    message: "Event not found",
    event
  });
};
```

- Podría hacer la conexión por separado y trabajarlo con clases
- En el navegador con RabbitMQ veo que tengo la queue CREATE_SALE
- La variable de entrn AMQPLIB_URL es

> amqp://localhost

- La petición la estoy haciendo desde POSTMAN a POST localhost:3001/events pasándole en el body

```
{
  "event": "CREATE_SALE",
  "data": {}

}
```

- De esta manera el eventBroker se conecta a rabbitMQ y no a diferentes puertos para extraer la data
- El eventBroker no necesita controladores, solo la conexión a rabbitMQ
- Sigo exponiendo el método POST a 3001/events pero internamente ya no llamo a otros endpoints

---

Consumir Queue

- Estos eventos pueden disparar excepciones
- Por ejemplo si no recibimos ningún evento

```
import { Request, Response } from "express";
import amqplib from 'amqplib'

const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

export const eventBrokerController = async (req: Request, res: Response) => {
  const { event, data } = req.body;

  if(!event){
    return res.status(500).json({
      message: "Event is required"
    })
  }
  const channel= await connection.createChannel()
  await channel.assertQueue(event)
```

```
  return res.status(404).json({
    message: "Event not found",
    event
  });
};
```

- Ahora los puertos expuestos de los microservicios ya dan igual
- Podemos hacer un endpoint para comprobar que el microservicio de Sales está vivo
- Las rutas me dan igual, me voy a manejar de otra manera con los eventos
- sales/src/index.ts

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
//import { SalesRoutes } from "./routes";

dotenv.config();

const app = express();
const port = process.env.PORT;

app.use(cors());

app.use(express.json());

app.get("/healt", (req, res) => {
  res.send(`Sales Microservice is live on port: ${port}`);
});

//app.use("/sales", SalesRoutes); no necesito las rutas

app.listen(port, () => {
  console.log("Sales Microservice is running on port:", port);
});
```

- Instalo amqplib en sales
- En el event-broker/src/enums tengo todos los enums, incluido el de ventas
- Copio el enum de sales del event-broker y lo copio en src/enum/sales.enum.ts

```
export enum SalesEvent {
  CREATE_SALE = "CREATE_SALE",
  UPDATE_SALE = "UPDATE_SALE",
  DELETE_SALE = "DELETE_SALE",
  GET_SALE = "GET_SALE",
  GET_SALES = "GET_SALES",
}
```

- Copio la conexión a rabbitMQ del eventBroker y la coloco también en el index.ts de Sales

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import { SalesRoutes } from "./routes";
import amqplib from 'amqplib'

dotenv.config();


const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

const app = express();
const port = process.env.PORT;

app.use(cors());

app.use(express.json());

app.get("/", (req, res) => {
  res.send(`Sales Microservice is alive on port: ${port}`);
});

//RabbitMQ
//creo un canal
const channel = await connection.createChannel()


//voy a necesitar un listener
//una vez obtengo el queue voy a consumirlo
await channel.consume(SalesEvent.CREATE_SALE, ()=>{
  console.log("consume CREATE_SALE")
}, noAck: true) //en true para que no almacene las peticiones

app.use("/sales", SalesRoutes);

app.listen(port, () => {
  console.log("Sales Microservice is running on port:", port);
});
```

- Esto me devuelve un status 200 pero no está llegando la info
- Esto es porque en el eventBroker, tras el assertQueue(event) (tras afirmar la cola), no estamos retornando nada
- Uso **sendToQueue** para mandar el evento y en el segundo parametro (content, tiene que ser un **buffer**) lo pongo en Buffer vacío
- event-broker/src/controllers/events.controller.ts

```
import { Request, Response } from "express";
import amqplib from 'amqplib'

const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

export const eventBrokerController = async (req: Request, res: Response) => {
  const { event, data } = req.body;

  if(!event){
    return res.status(500).json({
      message: "Event is required"
    })
  }

  try{
    const channel= await connection.createChannel()
  await channel.assertQueue(event)
                                          //new Buffer is deprecated
  await channel.sendToQueue(event, new Buffer(""))

  return res.status(404).json({
    message: "Event not found",
    event
  });

  }catch(error){
    return res.status(500).json({
      message: error.message
    })
  }

};
```

- En el controlador de sales necesito conectarme con el microservicio de usuarios y de productos
- Concretamente GET_USER y GET_PRODUCT
- sales/src/controllers/sales.controller

```
import axios from "axios";
import { Request, Response } from "express";

//const eventBroker = axios.create({
  //baseURL: "http://localhost:3001",
//});

const sales: any[] = [];
```

```
export const getAll = (req: Request, res: Response) => {
  return res.status(200).json({ message: "OK", sales });
};

export const createSale = async (req: Request, res: Response) => {
  const { data } = req.body;

  const { quantity } = data;

  //const { data: user } = await eventBroker.post("/events", {
    //event: "GET_USERS",
  //});

  //const { data: product } = await eventBroker.post("/events", {
    //event: "GET_PRODUCTS",
  //});

  const sale = {
    user: user.data.users[0],
    product: product.data.products[0],
    quantity,
    price: {
      unit: product.data.products[0]?.price,
      total: product.data.products[0]?.price * quantity,
    },
  };

  sales.push(sale);

  return res.status(200).json({ message: "OK", sales: sale });
};
```

- Hago ciertas modificaciones y coloco data en duro por ahora
- No hay request ni response, solo la función

```
import axios from "axios";
import { Request, Response } from "express";

//const eventBroker = axios.create({
 // baseURL: "http://localhost:3001",
//});


export const createSale = async () => {


const quantity = 10

const sale = {
  quantity,
  price:{
```

```
    unit: 100,
    total: 100
  }
}


  sales.push(sale);

  return sales;
};
```

- LLamo a este createSale desde sales/src/index.ts en lugar del console.log

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import { SalesRoutes } from "./routes";
import amqplib from 'amqplib'

dotenv.config();


const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

const app = express();
const port = process.env.PORT;

app.use(cors());

app.use(express.json());

app.get("/", (req, res) => {
  res.send(`Sales Microservice is alive on port: ${port}`);
});

//RabbitMQ
//creo un canal
const channel = await connection.createChannel()


                                          //async, importante!!!
await channel.consume(SalesEvent.CREATE_SALE, async ()=>{

  const sale = await createSale()

console.log(`Sale: ${JSON.stringify(sale)}`) //convierto a string el objeto que he
creado de sales
```

```
}, noAck: true)

//app.use("/sales", SalesRoutes);

app.listen(port, () => {
  console.log("Sales Microservice is running on port:", port);
});
```

- Esto así solo me sirve para comprobar que se realizó la petición, pero no hay data
- Puedo crear otros consumidores con **SalesEvent.GET_SALES y en const sale = await getAll()**

Enviar y recibir data por queue

- Quiero pasarle la quantity y el price a createSale
- sales/src/controllers/sales.controller

```
export const createSale = async (quantity: number = 10, price:number = 120) => {


const quantity = 10

const sale = {
  quantity,
  price:{
    unit: price,
    total: price * quantity
  }
}


  sales.push(sale);

  return sales;
};
```

- Cómo hago para captarlos en el event-broker para pasárselos a createSale
- event-broker/src/controllers/eventBroker.controller.ts

```
import { Request, Response } from "express";
import amqplib from 'amqplib'

const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

export const eventBrokerController = async (req: Request, res: Response) => {
```

```
  const { event, data } = req.body;

  if(!event){
    return res.status(500).json({
      message: "Event is required"
    })
  }

  try{
    const channel= await connection.createChannel()
  await channel.assertQueue(event)
                                        //en caso de venir la data vacía devolverá
un objeto vacío
  await channel.sendToQueue(event, Buffer.from( JSON.stringify(data || {})))// así
no devuelve undefined

  return res.status(404).json({
    message: "Event not found",
    event
  });

  }catch(error){
    return res.status(500).json({
      message: error.message
    })
  }

};
```

- No lo estoy recibiendo porque en el queue tenemos un onMessage: msg
- Lo puedo capturar mientras lo consumo. El content es un buffer
- sales/src/controllers/sales.controller.ts

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import { SalesRoutes } from "./routes";
import amqplib from 'amqplib'

dotenv.config();


const connection = await amqp.connect({
  hostname: 'localhost',
  username: 'admin',
  password: 'admin'
})

const app = express();
const port = process.env.PORT;
```

```
app.use(cors());

app.use(express.json());

app.get("/", (req, res) => {
  res.send(`Sales Microservice is alive on port: ${port}`);
});

//RabbitMQ
//creo un canal
const channel = await connection.createChannel()


                                          //aquí capturo el msg!!
await channel.consume(SalesEvent.CREATE_SALE, async (msg)=>{

  console.log(msg.content) //recibo un buffer


  //const sale = await createSale()

//console.log(`Sale: ${JSON.stringify(sale)}`)

}, noAck: true)

//app.use("/sales", SalesRoutes);

app.listen(port, () => {
  console.log("Sales Microservice is running on port:", port);
});
```

- Puedo usar toString() para consumir este Buffer

```
console.log(JSON.parse(msg.content.toString()))
//esto devuelve un json con el price y la quantity
```

- Para usar la sale que venga con createSale

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import { SalesRoutes } from "./routes";
import amqplib from 'amqplib'

dotenv.config();


const connection = await amqp.connect({
  hostname: 'localhost',
```

```
  username: 'admin',
  password: 'admin'
})

const app = express();
const port = process.env.PORT;

app.use(cors());

app.use(express.json());

app.get("/", (req, res) => {
  res.send(`Sales Microservice is alive on port: ${port}`);
});

//RabbitMQ
//creo un canal
const channel = await connection.createChannel()


                                          //async, importante!!!
await channel.consume(SalesEvent.CREATE_SALE, async (msg)=>{


  const data = JSON.parse(msg.content.toString())

  const sale = await createSale(data.quantity, data.price)

console.log(`Sale: ${JSON.stringify(sale)}`) //Ahora sale la data que le mando

}, noAck: true)

//app.use("/sales", SalesRoutes);

app.listen(port, () => {
  console.log("Sales Microservice is running on port:", port);
});
```

- Esta no es la mejor forma de trabajar, es con fines didácticos
- Vamos con Nest!

NODE MICROSERVICIOS - 05 REST API con Nest

- Uso el CLI para crear el primer microservicio

> nest new courses-ms

- Primero haremos una API, luego la pasaremos a microservicio
- Tengo un CRUD COMPLETO
- Conexion a **Prisma**

- Hay que instalar Prisma como dependencia de desarrollo

> npm i -D prisma npx prisma init

- También pueso usar npx prisma generate
- Esto crea la carpeta prisma y el archivo .env con una URL de deb (la borro, es de postgres)
- Puedo usar la db que he puesto a mano en mongoCompass

> npx prisma db push

- Para que no de problema de conexión por autorización añado a la variable de entorno donde he colocado la URL de la db

> DATABASE_URL=mongodb://user:password/authdb?retryWrites=true&=majority
> **&authSource=admin**

- Aquí tengo el schema (para autocompletado instalar extension Prisma)

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with your serverless
or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}


model Course {
  id String @default(uuid()) @id //debo remarcar con @id que es un id para que no
de error
  title String
  description String
  author_id String
  active Boolean @default(true)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  @@index([author_id, active])
}
```

- Si no quiero usar uuid puedo usar *id String @default(autoincrement()) @id*
- Con @@index indico los campos que quiero indexar
- Estos indices van a hacer unas copias en el disco para acceder más rápido

- La URL de conexión en el artchivo creado por prisma será (SQLite)

> file:./course.db

- Hago la migración

> npx prisma migrate dev --name init

Insertar datos

- Para sqlite puedo usar TablePlus para poder visualizar (en el curso al principio usa sqlite)
- Puedo aplicar cambios desde la linea de comandos (por ejemplo usar un integer en lugar de uuid)
- Debo usar *id String @default(autoincrement()) @id* entonces

> npx prisma migrate dev --name change uuid to int in id field

- Puedo insertar datos desde TablePlus

Microservicios

- El courses-ms/src/main.ts

```
import { ValidationPipe } from "@nestjs/common";
import { NestFactory } from "@nestjs/core";

import { MicroserviceOptions, Transport } from "@nestjs/microservices";

import { AppModule } from "./app.module";
import { QueuesEnum } from "./enums/queue.enum";

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.RMQ,
      options: {
        urls: [process.env.RMQ_URL],
        queue: QueuesEnum.CoursesQueue,
        queueOptions: {
          durable: true,
        },
        noAck: true,
      },
    },
  );

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
```

```
    }),
  );

  await app.listen();
}
bootstrap();
```

- courses/src/app.module

```
import { Module } from "@nestjs/common";
import { CourseModule } from "./course/course.module";

@Module({
  imports: [CourseModule],
})
export class AppModule {}
```

- courses-ms/src/course/course.module.ts

```
import { Module } from "@nestjs/common";
import { CourseController } from "./course.controller";
import { CourseService } from "./course.service";

@Module({
  controllers: [CourseController],
  providers: [CourseService],
})
export class CourseModule {}
```

- En el controller usamos MessagePattern y Payload (ya no hay decoradores como @Body() o @Params())
- courses-ms/src/course.controller.ts

```
import { Controller } from "@nestjs/common";
import { MessagePattern, Payload } from "@nestjs/microservices";

import { CreateCourseDto } from "./dto/create-course.dto";
import { UpdateCourseDto } from "./dto/update-course.dto";

import { CourseService } from "./course.service";
import { PaginationDto } from "./dto/pagination.dto";

@Controller()
export class CourseController {
  constructor(private readonly courseService: CourseService) {}

  @MessagePattern({ cmd: "healt_course" })
  healt() {
```

```
    return { healt: true };
  }

  @MessagePattern({ cmd: "get_all_courses" })
  findAll(@Payload() paginationDto: PaginationDto) {
    return this.courseService.findAll(paginationDto);
  }

  @MessagePattern({ cmd: "get_course" })
  findOne(@Payload("id") id: string) {
    return this.courseService.findOne(id);
  }

  @MessagePattern({ cmd: "create_course" })
  create(@Payload() createCourseDto: CreateCourseDto) {
    return this.courseService.create(createCourseDto);
  }

  @MessagePattern({ cmd: "update_course" })
  update(@Payload() updateCourseDto: UpdateCourseDto) {
    return this.courseService.update(updateCourseDto);
  }

  @MessagePattern({ cmd: "delete_course" })
  remove(@Payload("id") id: string) {
    return this.courseService.remove(id);
  }

  @MessagePattern({ cmd: "hard_delete_course" })
  removeHard(@Payload("id") id: string) {
    return this.courseService.removeHard(id);
  }
}
```

- En el servicio usamos el cliente de Prisma para comunicarnos con la db
- extiende de PrimsaClient e implementa onModuleInit
- hago la conexión con onModuleInit y await this.$connect()
- No hace falta inyectar nada
- Es this.course porque lo llamamos Course en el Schema

```
import { Injectable, Logger, OnModuleInit } from "@nestjs/common";
import { PrismaClient } from "@prisma/client";

import { RpcException } from "@nestjs/microservices";
import { CreateCourseDto } from "./dto/create-course.dto";
import { PaginationDto } from "./dto/pagination.dto";
import { UpdateCourseDto } from "./dto/update-course.dto";

@Injectable()
export class CourseService extends PrismaClient implements OnModuleInit {
  private readonly logger = new Logger("Course Service");
```

```
  async onModuleInit() {
    await this.$connect();
  }

  async findAll(paginationDto: PaginationDto) {
    try {
      const currentPage = paginationDto.page - 1;
      const limit = paginationDto.limit;

      const courses = await this.course.findMany({
        where: {
          active: true,
        },
        take: limit,
        skip: currentPage * limit,
      });

      return courses;
    } catch (error) {
      throw new RpcException(error.message);
    }
  }

  findOne(id: string) {
    return this.course.findUnique({
      where: {
        id,
        active: true,
      },
    });
  }

  async create(createCourseDto: CreateCourseDto) {
    try {
      const course = await this.course.create({
        data: createCourseDto,
      });

      return course;
    } catch (error) {
      this.logger.error(error.message);
    }
  }

  update(updateCourseDto: UpdateCourseDto) {
    return this.course.update({
      data: updateCourseDto,
      where: {
        id: updateCourseDto.id,
      },
    });
  }
```

```
  remove(id: string) {
    // soft delete => Eliminación suave o eliminación lógica
    return this.course.update({
      data: {
        active: false,
      },
      where: {
        id,
      },
    });
  }

  removeHard(id: string) {
    // hard delete => Eliminación fisica
    return this.course.delete({
      where: {
        id,
      },
    });
  }
}
```

- Cargo con Docker la imagen de mysql en la raíz (fuera de src)

```
version: "3.1"
services:
  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    ports:
      - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: courses
```

- .env course-ms

```
DATABASE_URL="file:./course.db
PORT=3001
```

APi-Gateway

- Crearemos el cliente
- Nos permitirá comunicarnos desde fuera con el microservicio

> nest new api-gateway

- Correrá en el puerto 3000 (variable de entorno PORT)
- Me quedo solo con el **main.ts** y el **app.module**
- Creo el resource de course dentro de api-gateway/src

> nest g res course

-
- Dejo solo el .module, .controller y el main.ts
- Necesitamos registrar el módulo para crear el cliente
- Debo tener instalado @nestjs/microservices (usar npm)
- En api-gateway/src/course/course.module,ts

```
import { Module } from "@nestjs/common";

import { ClientsModule, Transport } from "@nestjs/microservices";

import { CourseController } from "./course.controller";

import { QueuesEnum, ServicesTokens } from "../enums";

import { envs } from "../config/envs";

@Module({
  controllers: [CourseController],
  providers: [],
  imports: [
          {
        name: ServicesTokens.COURSE_SERVICE,//le paso el token de inyección
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl], //conecto a RabbitMQ
          queue: QueuesEnum.CoursesQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },

  ],
})
export class CourseModule {}
```

. Creo un api-gateway/src/enum/services.enum.ts para manejar los tokens de inyección

```
export enum ServicesTokens {
  COURSE_SERVICE = "COURSE_SERVICE",
  AUTH_SERVICE = "AUTH_SERVICE",
}
```

- Puedo crear un archivo index dentro de enums para mejorar las importaciones

```
export * from "./queues.enum";
export * from "./services.enum";
```

- En el main de api-gateway debo configurarlo como una API REST, no es un microservicio
- main de api-gateway

```
import { ValidationPipe } from "@nestjs/common";
import { NestFactory } from "@nestjs/core";

import { AppModule } from "./app.module";

import { envs } from "./config/envs";

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
    }),
  );

  await app.listen(envs.port);
}
bootstrap();
```

- En el api-gateway/src/course/course.controller.ts uso @inject para inyectar el token que tengo en enums
- Uso CLientProxy que se comunicará con el módulo que he registrado gracias al token de inyección
- Uso .send porque **me importa la respuesta**, si no me importara usaría emit
- Para comunicarme uso el patrón que le pasé a courses-ms/src/courses.controller.ts
- Debe ser idéntico, si le pasé un objeto, coloco un objeto.
- El tema de** UseGuards(AuthGuard) se verá más adelante**
- Para trabajar con los Payloads usaremos **Pipes** para transformar la data
- api-gateway/src/course/course.controller.ts

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Inject,
```

```
  Logger,
  Param,
  ParseIntPipe,
  Patch,
  Post,
  Query,
  Request,
  UseGuards,
} from "@nestjs/common";

import { ClientProxy } from "@nestjs/microservices";

import { AuthGuard } from "../auth/guards/auth.guard";
import { ServicesTokens } from "../enums";
import { CreateCourseDto } from "./dto/create-course.dto";
import { UpdateCourseDto } from "./dto/update-course.dto";

@Controller("course")
export class CourseController {
  private readonly logger = new Logger("Course Controller");

  constructor(
    @Inject(ServicesTokens.COURSE_SERVICE)
    private readonly courseService: ClientProxy,
  ) {}

  @Get("healt")
  healt() {
    return this.courseService.send({ cmd: "healt_course" }, {}); //si no envio
data envío un objeto vacío
  }

  @UseGuards(AuthGuard)
  @Get()
  findAll(          //Uso un pipe para pasar de string a número los query
    @Query("limit", ParseIntPipe) limit: number = 5,
    @Query("page", ParseIntPipe) page = 1,
  ) {
    return this.courseService.send({ cmd: "get_all_courses" }, { limit, page });
  }

  @UseGuards(AuthGuard)
  @Get(":id")
  findOne(@Param("id") id: string) {
    return this.courseService.send({ cmd: "get_course" }, { id });
  }

  @UseGuards(AuthGuard)
  @Post()                                          //obtengo la request, veremos
porqué
  create(@Body() createCourseDto: CreateCourseDto, @Request() request) {
    createCourseDto.author_id = request.user.id; //me pide el author_id

    return this.courseService.send(
```

```
      { cmd: "create_course" },
      { ...createCourseDto }, //uso el spread para crear una copia
    );
  }

  @UseGuards(AuthGuard)
  @Patch(":id")
  update(@Param("id") id: string, @Body() updateCourseDto: UpdateCourseDto) {
    return this.courseService.send(
      { cmd: "update_course" },
      { ...updateCourseDto, id },
    );
  }

  @UseGuards(AuthGuard)
  @Delete(":id")
  remove(@Param("id") id: string) {
    return this.courseService.send({ cmd: "delete_course" }, { id });
  }

  @UseGuards(AuthGuard)
  @Delete("hard/:id")
  removeHard(@Param("id") id: string) {
    return this.courseService.send({ cmd: "hard_delete_course" }, { id });
  }
}
```

- En **courses-ms** es donde tengo los patrones cmd con los que me comunico y el servicio (de courses-ms) inyectado
- courses/src/course/course.controller.ts

```
import { Controller } from "@nestjs/common";
import { MessagePattern, Payload } from "@nestjs/microservices";

import { CreateCourseDto } from "./dto/create-course.dto";
import { UpdateCourseDto } from "./dto/update-course.dto";

import { CourseService } from "./course.service";
import { PaginationDto } from "./dto/pagination.dto";

@Controller()
export class CourseController {
  constructor(private readonly courseService: CourseService) {}

  @MessagePattern({ cmd: "healt_course" })
  healt() {
    return { healt: true };
  }

  @MessagePattern({ cmd: "get_all_courses" })
  findAll(@Payload() paginationDto: PaginationDto) {
```

```
    return this.courseService.findAll(paginationDto);
  }

  @MessagePattern({ cmd: "get_course" })
  findOne(@Payload("id") id: string) {
    return this.courseService.findOne(id);
  }

  @MessagePattern({ cmd: "create_course" })
  create(@Payload() createCourseDto: CreateCourseDto) {
    return this.courseService.create(createCourseDto);
  }

  @MessagePattern({ cmd: "update_course" })
  update(@Payload() updateCourseDto: UpdateCourseDto) {
    return this.courseService.update(updateCourseDto);
  }

  @MessagePattern({ cmd: "delete_course" })
  remove(@Payload("id") id: string) {
    return this.courseService.remove(id);
  }

  @MessagePattern({ cmd: "hard_delete_course" })
  removeHard(@Payload("id") id: string) {
    return this.courseService.removeHard(id);
  }
}
```

- En el **service**de course-ms/src/course/course.service.ts es donde conecto con Prisma para interactuar con la db

```
import { Injectable, Logger, OnModuleInit } from "@nestjs/common";
import { PrismaClient } from "@prisma/client";

import { RpcException } from "@nestjs/microservices";
import { CreateCourseDto } from "./dto/create-course.dto";
import { PaginationDto } from "./dto/pagination.dto";
import { UpdateCourseDto } from "./dto/update-course.dto";

@Injectable()
export class CourseService extends PrismaClient implements OnModuleInit {
  private readonly logger = new Logger("Course Service");

  async onModuleInit() {
    await this.$connect();
  }

  async findAll(paginationDto: PaginationDto) {
    try {
      const currentPage = paginationDto.page - 1;
```

```
      const limit = paginationDto.limit;

      const courses = await this.course.findMany({
        where: {
          active: true,
        },
        take: limit, //paginación con Prisma
        skip: currentPage * limit,
      });

      return courses;
    } catch (error) {
      throw new RpcException(error.message); //se verá mas adelante
    }
  }

  findOne(id: string) {
    return this.course.findUnique({
      where: {
        id,
        active: true,
      },
    });
  }

  async create(createCourseDto: CreateCourseDto) {
    try {
      const course = await this.course.create({
        data: createCourseDto,
      });

      return course;
    } catch (error) {
      this.logger.error(error.message);
    }
  }

  update(updateCourseDto: UpdateCourseDto) {
    return this.course.update({
      data: updateCourseDto,
      where: {
        id: updateCourseDto.id,
      },
    });
  }

  remove(id: string) {
    // soft delete => Eliminación suave o eliminación lógica
    return this.course.update({
      data: {
        active: false,
      },
      where: {
        id,
```

```
      },
    });
  }

  removeHard(id: string) {
    // hard delete => Eliminación fisica
    return this.course.delete({
      where: {
        id,
      },
    });
  }
}
```

- Al usar microservicios, lo que me devuelve el findAll es un Observable de tipo any
- Puedo colocar el observable como valor de retorno

```
@UseGuards(AuthGuard)
@Get(":id")
findOne(@Param("id") id: string) : Observable<any> {
  return this.courseService.send({ cmd: "get_course" }, { id });
}
```

- Debo copiar los dto de courses-ms a api-gateway/src/courses/dtos

```
import { IsOptional, IsString } from "class-validator";

// DTO => Data Transfer Object

export class CreateCourseDto {
  @IsString()
  title: string;

  @IsString()
  description: string;

  @IsString()
  @IsOptional()
  author_id: string;
}
```

- ESTOY APUNTANDO AL PUERTO 3000 DE LOCALHOST + ENDPOINT del controller

---

AUTH: Microservicios híbridos con REST

- El microservicio puede comunicarse por http y por comunicación interna de microservicios con la que se añade una capa de seguridad

- Son híbridos porque usan ambos protocolos
- Este microservicio usará su propia db (con mongo)
- Generaremos un AuthGuard, que usaremos para autorizar conexiones
- El gateway es el que se conecta y decide si puede seguir al siguiente ms o no
- Usaremos TCP de entrada pero al final cambiaremos a RabbitMQ y asignaremos RMQ a este auth-ms

Iniciando ms

- Creo un nuevo ms agnóstico fuera de api-gateway

> nest new auth-ms

-
- Usaremos RabbitMQ para comunicar la api-gateway con el ms
- El main de auth-ms es así
- auth-ms/src/main.ts

```
import { ValidationPipe } from "@nestjs/common";
import { NestFactory } from "@nestjs/core";
import { MicroserviceOptions, Transport } from "@nestjs/microservices";

import { AppModule } from "./app.module";
import { QueuesEnum } from "./enums/queue.enum";

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.RMQ,
      options: {
        urls: [process.env.RMQ_URL],
        queue: QueuesEnum.AuthQueue,
        queueOptions: {
          durable: true,
        },
        noAck: true,
      },
    },
  );

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
    }),
  );

  await app.listen();
}
bootstrap();
```

La variable de entorno de RMQ es algo asi

- auth-ms/.env

```

```

- El queues.enum es así
- auth-ms/src/enums/queue.enum.ts

```
export enum QueuesEnum {
  AuthQueue = "AuthQueue",
}
```

- El app.module queda tal cual
- auth-ms/src/app.module

```
import { Module } from "@nestjs/common";
import { AuthModule } from "./auth/auth.module";

@Module({
  imports: [AuthModule],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Dentro de auth-ms creamos el módulo de auth, el controller y el service el auth.module queda así

> nest g mo auth nest g co auth nest g s auth

- Importo el módulo de JWT de @nestjs/jwt y uso el método register al que le paso la variable de la SECRET_KEY
- Le digo que el token expire en 1 hora
- En auth-ms/src/auth/auth.module.ts

```
import { Module } from "@nestjs/common";
import { JwtModule } from "@nestjs/jwt";

import { AuthController } from "./auth.controller";
import { AuthService } from "./auth.service";

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    JwtModule.register({
```

```
      secret: process.env.JWTSecret,
      signOptions: { expiresIn: "1h" },
    }),
  ],
})
export class AuthModule {}
```

- El Schema de Prisma queda así
- auth/prisma/schema.prisma

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with your serverless
or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}

model User {
  id String @map("_id") @id @default(auto()) @db.ObjectId

  email String @unique
  password String
  username String

  active Boolean @default(true)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}
```

- Dentro de auth-ms/src/auth/controller

```
import { Controller } from "@nestjs/common";
import { MessagePattern, Payload } from "@nestjs/microservices";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";

import { AuthService } from "./auth.service";
```

```
@Controller()
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @MessagePattern("auth.healt")
  healt() {
    return { status: 200, message: "ok" };
  }

  @MessagePattern("auth.verify.token")
  verifyToken(@Payload("token") token: string) {
    return this.authService.verifyToken(token);
  }

  @MessagePattern("auth.login")
  login(@Payload() loginUserDto: LoginUserDto) {
    return this.authService.login(loginUserDto);
  }

  @MessagePattern("auth.register")
  register(@Payload() createUserDto: CreateUserDto) {
    return this.authService.register(createUserDto);
  }
}
```

- Veamos el servicio!
- auth-ms/src/auth/auth.service.ts

```
import {
  Injectable,
  OnModuleInit,
  UnauthorizedException,
} from "@nestjs/common";
import { JwtService } from "@nestjs/jwt";
import { RpcException } from "@nestjs/microservices";

import { PrismaClient } from "@prisma/client";

import * as bcrypt from "bcrypt";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";

@Injectable()
export class AuthService extends PrismaClient implements OnModuleInit {
  constructor(private readonly jwtService: JwtService) {
    super();
  }

  async onModuleInit() {
    await this.$connect();
```

```
  }

  private hashPassword(password: string) {
    const salt = bcrypt.genSaltSync(12);

    return bcrypt.hashSync(password, salt);
  }

  private verifyPassword(password: string, hash: string) {
    return bcrypt.compareSync(password, hash);
  }

  private validateUserEmail(email: string) {
    return this.user.findUnique({
      where: {
        email,
      },
    });
  }

  private signToken(payload: { id: string; email: string }) {
    return this.jwtService.sign({
      ...payload,
    });
  }

  async register(createUserDto: CreateUserDto) {
    const { password, email } = createUserDto;

    if (await this.validateUserEmail(email))
      throw new RpcException("Email already exists");

    createUserDto.password = this.hashPassword(password);

    const {
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      password: _,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      active: __,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      updatedAt: ___,
      ...user
    } = await this.user.create({
      data: createUserDto,
    });

    return {
      ...user,
      token: this.signToken({
        id: user.id,
        email,
      }),
    };
  }
```

```
  async login(loginUserDto: LoginUserDto) {
    const { password, email } = loginUserDto;

    const storedUser = await this.validateUserEmail(email);

    if (!storedUser || !this.verifyPassword(password, storedUser.password))
      throw new RpcException("Email or password is invalid");

    const {
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      password: _,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      active: __,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      updatedAt: ___,
      ...user
    } = storedUser;

    return {
      ...user,
      token: this.signToken({
        id: user.id,
        email,
      }),
    };
  }

  verifyToken(token: string) {
    const payload = this.jwtService.verify(token, {
      secret: process.env.JWTSecret,
    });

    if (!payload) throw new UnauthorizedException();

    return {
      token: this.signToken({ id: payload.id, email: payload.email }),
      user: {
        id: payload.id,
        email: payload.email,
      },
    };
  }
}
```

- El dto de createUser (en auth-ms)

```
import { IsEmail, IsString, IsStrongPassword } from "class-validator";

export class CreateUserDto {
  @IsString()
  @IsEmail()
```

```
  email: string;

  @IsString()
  @IsStrongPassword({
    minLength: 8,
    minLowercase: 1,
    minNumbers: 1,
    minSymbols: 1,
    minUppercase: 1,
  })
  password: string;

  @IsString()
  username: string;
}
```

- El dto del login

```
import { IsEmail, IsString, IsStrongPassword } from "class-validator";

export class LoginUserDto {
  @IsString()
  @IsEmail()
  email: string;

  @IsString()
  @IsStrongPassword({
    minLength: 8,
    minLowercase: 1,
    minNumbers: 1,
    minSymbols: 1,
    minUppercase: 1,
  })
  password: string;
}
```

- Para usar mongo uso docker
- docker.compose.yml

```
version: "3.1"
services:
  mongo:
    image: mongo
    container_name: mongo_auth
    restart: always
    ports:
      - 27017:27017
    environment:
```

```
      MONGO_INITDB_ROOT_USERNAME: yirsis
      MONGO_INITDB_ROOT_PASSWORD: BbYa0C2fOtR6
```

- Conectando el microservicio al gateway
- En el api-gateway/arc/auth/auth.module.ts

```
import { Module } from "@nestjs/common";
import { ClientsModule, Transport } from "@nestjs/microservices";

import { AuthController } from "./auth.controller";

import { QueuesEnum, ServicesTokens } from "../enums";

import { envs } from "../config/envs";

@Module({
  controllers: [AuthController],
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
})
export class AuthModule {}
```

- En el controller le inyecto el token usando CLientProxy
- api-gateway/src/auth/auth.controller

```
import {
  Body,
  Controller,
  Get,
  Inject,
  Post,
  Request,
  UseGuards,
} from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";
```

```
import { ServicesTokens } from "../enums";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";
import { AuthGuard } from "./guards/auth.guard";

@Controller("auth")
export class AuthController {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}

  @Get()
  healt() {
    return this.authService.send("auth.healt", {});
  }

  @UseGuards(AuthGuard)
  @Get("verify-token")
  verifyToken(@Request() request: any) {
    return { token: request.token, user: request.user };
  }

  @Post("login")
  login(@Body() loginUserDto: LoginUserDto) {
    return this.authService.send("auth.login", { ...loginUserDto });
  }

  @Post("register")
  register(@Body() createUserDto: CreateUserDto) {
    return this.authService.send("auth.register", { ...createUserDto });
  }
}
```

- En api-gateway/src/auth/guards/auth.guard.ts

```
import {
  CanActivate,
  ExecutionContext,
  Inject,
  Injectable,
  UnauthorizedException,
} from "@nestjs/common";

import { ClientProxy } from "@nestjs/microservices";
import { Request } from "express";
import { firstValueFrom } from "rxjs";
import { ServicesTokens } from "../../enums";

@Injectable()
```

```
export class AuthGuard implements CanActivate {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE) //inyecto AUTH_SERVICE
    private readonly authService: ClientProxy,
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest(); //obtengoi la request
    const token = this.extractTokenFromHeader(request); //desestructuro el token

    if (!token) throw new UnauthorizedException();

    const { token: renewToken, user } = await firstValueFrom( //renombro token a
renewToken
      this.authService.send("auth.verify.token", { token }), //mando el token al
ms
    );

    request["token"] = renewToken; //guardo el nuevo token en la request
    request["user"] = user; //guardo el user en la request

    return renewToken;
  }
                          //helper para extraer el token
  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(" ") ?? [];
    return type === "Bearer" ? token : undefined;
  }
}
```

- En api-gateway/src/auth/config/envs hago la validación de las variables de entorno con **joi**

```
import "dotenv/config";

import * as joi from "joi";

const envSchema = joi
  .object({
    PORT: joi.number().required(),
    RMQ_URL: joi.string().required(),
  })
  .unknown(true); //dejo el unknown en true para el resto de variables de Node

const { error, value } = envSchema.validate(process.env);

if (error) {
  throw new Error(`Config validation error: ${error.message}`);
}

interface Env {
  PORT: number;
  RMQ_URL: string;
```

```
}

const envVars: Env = value;

export const envs = {
  port: envVars.PORT as number,
  rmqUrl: envVars.RMQ_URL as string,
};
```

- **RESUMEN**
- En el main del ms creo el microservicio

```
import { ValidationPipe } from "@nestjs/common";
import { NestFactory } from "@nestjs/core";
import { MicroserviceOptions, Transport } from "@nestjs/microservices";

import { AppModule } from "./app.module";
import { QueuesEnum } from "./enums/queue.enum";

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.RMQ,
      options: {
        urls: [process.env.RMQ_URL],
        queue: QueuesEnum.AuthQueue,
        queueOptions: {
          durable: true,
        },
        noAck: true,
      },
    },
  );

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
    }),
  );

  await app.listen(); //le quito el puerto para que no marque errores
}
bootstrap();
```

- Si trabajara con la capa de transporte TCP pondría Transport.TCP y en un objeto oprions el port: 3001, por ejemplo

- Lo comunico con el api-gateway resgistrándolo en el api-gateway/src/auth/auth.module

```
import { Module } from "@nestjs/common";
import { ClientsModule, Transport } from "@nestjs/microservices";

import { AuthController } from "./auth.controller";

import { QueuesEnum, ServicesTokens } from "../enums";

import { envs } from "../config/envs";

@Module({
  controllers: [AuthController],
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
})
export class AuthModule {}
```

- Desde el controller del api-gateway me comunico con el controller del ms
- El api-gateway es lo que expongo al exterior
- Luego me comunico internamente a través del ClientProxy con el controller del auth-ms usando el token de inyección y el decorador @Inject de @nestjs/common (!)
- El spread operator es importante!
- api-gateway/src/auth/auth.controller

```
import {
  Body,
  Controller,
  Get,
  Inject,
  Post,
  Request,
  UseGuards,
} from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";
```

```
import { ServicesTokens } from "../enums";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";
import { AuthGuard } from "./guards/auth.guard";

@Controller("auth")
export class AuthController {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}

  @Get()
  healt() {
    return this.authService.send("auth.healt", {});
  }

  @UseGuards(AuthGuard)
  @Get("verify-token")
  verifyToken(@Request() request: any) {
    return { token: request.token, user: request.user };
  }

  @Post("login")
  login(@Body() loginUserDto: LoginUserDto) {
    return this.authService.send("auth.login", { ...loginUserDto }); //usar el
spread operattor!!
  }

  @Post("register")
  register(@Body() createUserDto: CreateUserDto) {
    return this.authService.send("auth.register", { ...createUserDto });
  }
}
```

- El .send está llamado al controlador del auth-ms, desde donde me comunico con el authService

```
import { Controller } from "@nestjs/common";
import { MessagePattern, Payload } from "@nestjs/microservices";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";

import { AuthService } from "./auth.service";

@Controller()
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @MessagePattern("auth.healt")
  healt() {
```

```
    return { status: 200, message: "ok" };
  }

  @MessagePattern("auth.verify.token")
  verifyToken(@Payload("token") token: string) {
    return this.authService.verifyToken(token);
  }

  @MessagePattern("auth.login")
  login(@Payload() loginUserDto: LoginUserDto) {
    return this.authService.login(loginUserDto);
  }

  @MessagePattern("auth.register")
  register(@Payload() createUserDto: CreateUserDto) {
    return this.authService.register(createUserDto);
  }
}
```

- El authService (solo hay uno y está en auth-ms)

```
import {
  Injectable,
  OnModuleInit,
  UnauthorizedException,
} from "@nestjs/common";
import { JwtService } from "@nestjs/jwt";
import { RpcException } from "@nestjs/microservices";

import { PrismaClient } from "@prisma/client";

import * as bcrypt from "bcrypt";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";

@Injectable()
export class AuthService extends PrismaClient implements OnModuleInit {
  constructor(private readonly jwtService: JwtService) {
    super();
  }

  async onModuleInit() {
    await this.$connect();
  }

  private hashPassword(password: string) {
    const salt = bcrypt.genSaltSync(12);

    return bcrypt.hashSync(password, salt);
  }
```

```
  private verifyPassword(password: string, hash: string) {
    return bcrypt.compareSync(password, hash);
  }

  private validateUserEmail(email: string) {
    return this.user.findUnique({
      where: {
        email,
      },
    });
  }

  private signToken(payload: { id: string; email: string }) {
    return this.jwtService.sign({
      ...payload,
    });
  }

  async register(createUserDto: CreateUserDto) {
    const { password, email } = createUserDto;

    if (await this.validateUserEmail(email))
      throw new RpcException("Email already exists");

    createUserDto.password = this.hashPassword(password);

    const {
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      password: _,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      active: __,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      updatedAt: ___,
      ...user
    } = await this.user.create({
      data: createUserDto,
    });

    return {
      ...user,
      token: this.signToken({
        id: user.id,
        email,
      }),
    };
  }

  async login(loginUserDto: LoginUserDto) {
    const { password, email } = loginUserDto;

    const storedUser = await this.validateUserEmail(email);

    if (!storedUser || !this.verifyPassword(password, storedUser.password))
      throw new RpcException("Email or password is invalid");
```

```
    const {
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      password: _,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      active: __,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      updatedAt: ___,
      ...user
    } = storedUser;

    return {
      ...user,
      token: this.signToken({
        id: user.id,
        email,
      }),
    };
  }

  verifyToken(token: string) {
    const payload = this.jwtService.verify(token, {
      secret: process.env.JWTSecret,
    });

    if (!payload) throw new UnauthorizedException();

    return {
      token: this.signToken({ id: payload.id, email: payload.email }),
      user: {
        id: payload.id,
        email: payload.email,
      },
    };
  }
}
```

- En el course.module del api-gateway también debo registrar el microservicio de auth y el de course, claro!

```
import { Module } from "@nestjs/common";

import { ClientsModule, Transport } from "@nestjs/microservices";

import { CourseController } from "./course.controller";

import { QueuesEnum, ServicesTokens } from "../enums";

import { envs } from "../config/envs";

@Module({
```

```
  controllers: [CourseController],
  providers: [],
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
      {
        name: ServicesTokens.COURSE_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.CoursesQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
})
export class CourseModule {}
```

- Tengo dos enums en la carpeta api-gateway/src/enums
- El services.enum

```
export enum ServicesTokens {
  COURSE_SERVICE = "COURSE_SERVICE",
  AUTH_SERVICE = "AUTH_SERVICE",
}
```

- Y el queues enum

```
export enum QueuesEnum {
  AuthQueue = "AuthQueue",
  CoursesQueue = "CoursesQueue",
}
```

- Por ahora tengo dos microservicios (auth y courses) y un API REST que es el cliente api-gateway

Cuando no envío nada en el .send, solo el objeto con cmd:lo.que.sea, debo mandar un objeto vacío a continuación, representando la data del payload

```
  @Get()
  healt() {
    return this.authService.send("auth.healt", {});
  }
```

- Necesito los dtos tanto en el ms como en el api-gateway
- En el api-gateway tengo el auth.module, que es donde registro el microservicio

```
import { Module } from "@nestjs/common";
import { ClientsModule, Transport } from "@nestjs/microservices";

import { AuthController } from "./auth.controller";

import { QueuesEnum, ServicesTokens } from "../enums";

import { envs } from "../config/envs";

@Module({
  controllers: [AuthController],
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
})
export class AuthModule {}
```

- En el controller tengo los métodos Get, Post, y todos los decoradores
- Direcciono al microservicio usando el mismo cmd

- Es importante usar el spread operator en el envío de la data para que no de error
- api-gateway/src/auth/auth.controller

```
import {
  Body,
  Controller,
  Get,
  Inject,
  Post,
  Request,
  UseGuards,
} from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";

import { ServicesTokens } from "../enums";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";
import { AuthGuard } from "./guards/auth.guard";

@Controller("auth")
export class AuthController {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}

  @Get()
  healt() {
    return this.authService.send("auth.healt", {});
  }

  @UseGuards(AuthGuard)
  @Get("verify-token")
  verifyToken(@Request() request: any) {
    return { token: request.token, user: request.user };
  }

  @Post("login")
  login(@Body() loginUserDto: LoginUserDto) {
    return this.authService.send("auth.login", { ...loginUserDto }); //usar spread
operator!!
  }

  @Post("register")
  register(@Body() createUserDto: CreateUserDto) {
    return this.authService.send("auth.register", { ...createUserDto });
  }
}
```

- En api-gateway/src/auth/guards tengo mi versión del AuthGuard

```
import {
  CanActivate,
  ExecutionContext,
  Inject,
  Injectable,
  UnauthorizedException,
} from "@nestjs/common";

import { ClientProxy } from "@nestjs/microservices";
import { Request } from "express";
import { firstValueFrom } from "rxjs";
import { ServicesTokens } from "../../enums";

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,   //Inyecto el AUTH_SERVICE como
ClientProxy
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest(); //extraigo la request
    const token = this.extractTokenFromHeader(request); //extraigo el token con
este método

    if (!token) throw new UnauthorizedException(); //lanzo un error si no hay
token

    const { token: renewToken, user } = await firstValueFrom(
      this.authService.send("auth.verify.token", { token }),
    );

    request["token"] = renewToken; //le paso el token a la request
    request["user"] = user; //y el user también

    return renewToken;
  }
                          //método para extraer el token
  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(" ") ?? [];
    return type === "Bearer" ? token : undefined;
  }
}
```

- Es en el servicio de auth-ms donde está toda la mandanga, como verificar el token, etc

---

NODE MICROSERVICIOS - 06 GraphQL como Microservice Client

- Elimino el api-gateway
- Creo un nuevo proyecto

```
nest new client-gql
```

Agregar graphQL

- Instalo

```
npm i @nestjs/graphql @nestjs/apollo @apollo/server graphql
```

- En el app.module uso ApolloServer (Apollo Studio) en lugar del playground (con el navegador)
- Uso autoSchemaFile para que genere automáticamente los Schemas
- join lo importo de path, coloco node: delante porque es la nueva convención con los paquetes de node

```
import { join } from "node:path";

import { Module } from "@nestjs/common";

import { ApolloServerPluginLandingPageLocalDefault } from
"@apollo/server/plugin/landingPage/default";
import { ApolloDriver, ApolloDriverConfig } from "@nestjs/apollo";
import { GraphQLModule } from "@nestjs/graphql";

import { RandomModule } from "./random/random.module";
import { AuthModule } from './auth/auth.module';
import { CoursesModule } from './courses/courses.module';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: join(process.cwd(), "src/schema.gql"), //para que genere
automaticamente los schemas
      playground: false,
      plugins: [ApolloServerPluginLandingPageLocalDefault()],
    }),
    RandomModule,
    //AuthModule,
    //CoursesModule,
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- El main se queda igual

```
import { NestFactory } from "@nestjs/core";

import { AppModule } from "./app.module";
import { envs } from "./config/envs";

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  await app.listen(envs.port);
}
bootstrap();
```

- Recuerda que para que graphql funcione debe tener al menos un query y un resolver

> nest g res random GraphQL (code first) (uso decoradores y Nest se encarga por detrás de hacer los schemas) Generar CRUD (si!)

- El client-gql/src/random.module queda así

```
import { Module } from '@nestjs/common';
import { RandomService } from './random.service';
import { RandomResolver } from './random.resolver';

@Module({
  providers: [RandomResolver, RandomService],
})
export class RandomModule {}
```

- El random.resolver lo dejo solo con el findAll

```
import { Int, Query, Resolver } from "@nestjs/graphql";

import { Random } from "./entities/random.entity";
import { RandomService } from "./random.service";

            //random.entity
@Resolver(() => Random)
export class RandomResolver {
  constructor(private readonly randomService: RandomService) {}

  @Query(() => Int, {
    name: "getRandomNumber", //haré la query con este nombre
    description: "Get a random number",
  })
  findAll() {
    return this.randomService.findAll();
  }
}
```

- Debo definir el tipo en client-gql/src/entities/random.entity.ts

```
import { ObjectType, Field, Int } from '@nestjs/graphql';

@ObjectType()
export class Random {
  @Field(() => Int, { description: 'Example field (placeholder)' })
  exampleField: number;
}
```

- El random service queda así

```
import { Injectable } from "@nestjs/common";

@Injectable()
export class RandomService {
  findAll() {
    return Math.floor(Math.random() * 1000); //número random entre 0 y 1000
  }
}
```

- En el client-gql/schema.gql tengo esto

```
type Query {
  """Get a random number"""
  getRandomNumber: Int!
}
```

- La query la hago así

```
query Query{
    getRandomNumber
}
```

Mutations

- Genero con nest g res auth (code first)
- client-gql/src/auth
- auth.module

```
import { Module } from "@nestjs/common";
import { ClientsModule, Transport } from "@nestjs/microservices";

import { AuthResolver } from "./auth.resolver";
import { AuthService } from "./auth.service";

import { envs } from "../config/envs";
import { QueuesEnum, ServicesTokens } from "../enums";

@Module({
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
  providers: [AuthResolver, AuthService],
})
export class AuthModule {}
```

- El resolver en client-gql/src/auth/auth.resolver.ts
- Uso @Args para capturar el inputType (login.input)

```
import { UseGuards } from "@nestjs/common";
import { Args, Mutation, Query, Resolver } from "@nestjs/graphql";

import { AuthService } from "./auth.service";

import { AuthGuard } from "./guards/auth.guard";

import { LoginInput } from "./dto/login.input";
import { RegisterInput } from "./dto/register.input";

import { GetAuthenticatedUserByToken } from
"../decorators/getUserByToken.decorator";
import { Auth } from "./entities/auth.entity";
import { Token } from "./entities/token.entity";

@Resolver(() => Auth)
export class AuthResolver {
  constructor(private readonly authService: AuthService) {}
```

```
  @Mutation(() => Auth, { name: "login" })
  login(@Args("loginInput") loginInput: LoginInput) {
    return this.authService.login(loginInput);
  }

  @Mutation(() => Auth, { name: "register" })
  register(@Args("registerInput") registerInput: RegisterInput) {
    return this.authService.register(registerInput);
  }

  @Query(() => Token, { name: "verifyJWT" })
  @UseGuards(AuthGuard)
  verifyToken(@GetAuthenticatedUserByToken() user: Token) {
    return this.authService.verifyToken(user);
  }
}
```

- En el service uso firstValueFrom para trabajar con promesas en lugar de Observables
- Inyecto el token de inyección y lo uso con ClientProxy

```
import { Inject, Injectable } from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";

import { firstValueFrom } from "rxjs";
import { ServicesTokens } from "../enums";
import { LoginInput } from "./dto/login.input";
import { RegisterInput } from "./dto/register.input";
import { Auth } from "./entities/auth.entity";
import { Token } from "./entities/token.entity";

@Injectable()
export class AuthService {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}
                                      //al ser async trabajamos con promesas
                                      //Regresa una promesa de tipo Auth
  async login(loginInput: LoginInput): Promise<Auth> {
    const {
      id,
      token,
      email,
      username: name,//renombro el username a name
    } = await firstValueFrom(
      this.authService.send("auth.login", { ...loginInput }),
    );

    return {
      id,
```

```
      email,
      name,
      token,
    };
  }

  async register(registerInput: RegisterInput): Promise<Auth> {
    const {
      id,
      token,
      email,
      username: name,
    } = await firstValueFrom(
      this.authService.send("auth.register", { ...registerInput }),
    );

    return {
      id,
      email,
      name,
      token,
    };
  }

  verifyToken({ token, user }: Token): Token {
    return { token, user };
  }
}
```

- El authGuard es el mismo, solo que extraigo el context de Gql

```
import {
  CanActivate,
  ExecutionContext,
  Inject,
  Injectable,
  UnauthorizedException,
} from "@nestjs/common";

import { GqlExecutionContext } from "@nestjs/graphql";
import { ClientProxy } from "@nestjs/microservices";
import { Request } from "express";
import { firstValueFrom } from "rxjs";
import { ServicesTokens } from "../../enums";

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}
```

```
  async canActivate(context: ExecutionContext): Promise<boolean> {
    let request = context.switchToHttp().getRequest();

    if (!request) {
      const ctx = GqlExecutionContext.create(context); //Gql
      request = ctx.getContext().req;
    }

    const token = this.extractTokenFromHeader(request);

    if (!token) throw new UnauthorizedException();

    const { token: renewToken, user } = await firstValueFrom(
      this.authService.send("auth.verify.token", { token }),
    );

    request["token"] = renewToken;
    request["user"] = user;

    return renewToken;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(" ") ?? [];

    return type === "Bearer" ? token : undefined;
  }
}
```

- En entities tengo auth.entity

```
import { Field, ObjectType } from "@nestjs/graphql";

@ObjectType()
export class Auth {
  @Field(() => String, {
    name: "id",
    description: "id of the user",
  })
  id: string;

  @Field(() => String, {
    name: "name",
    description: "name of the user",
  })
  name: string;

  @Field(() => String, {
    name: "email",
    description: "email of the user",
  })
  email: string;
```

```
  @Field(() => String, {
    name: "token",
    description: "jwt of the user",
    nullable: true,
  })
  token: string;
}
```

- También token.entity

```
import { Field, ObjectType } from "@nestjs/graphql";

@ObjectType()
class UserJWT {
  @Field(() => String, {
    name: "id",
    description: "id of the user",
  })
  id: string;

  @Field(() => String, {
    name: "email",
    description: "email of the user",
  })
  email: string;
}

@ObjectType()
export class Token {
  @Field(() => String, {
    name: "token",
    description: "jwt of the user",
    nullable: true,
  })
  token: string;

  @Field(() => UserJWT, {
    name: "user",
    description: "user data",
  })
  user: UserJWT;
}
```

- En los dtos tengo login.input

```
import { Field, InputType } from "@nestjs/graphql";

@InputType()
```

```
export class LoginInput {
  @Field(() => String, { name: "email", description: "User email" })
  email: string;

  @Field(() => String, { name: "password", description: "User password" })
  password: string;
}
```

- register.input

```
import { Field, InputType } from "@nestjs/graphql";

@InputType()
export class RegisterInput {
  @Field(() => String, { name: "email", description: "User email" })
  email: string;

  @Field(() => String, { name: "username", description: "Username" })
  username: string;

  @Field(() => String, { name: "password", description: "User password" })
  password: string;
}
```

Proceso de autenticación

- Una vez tengo el cliente-gql/src/auth/auth.controller hago el resolver
- Inyecto el AuthService
- El @Resolver devuelve algo de tipo Auth
- Las mutation, por tanto, también
- La Query verifyToken devuelve algo de tipo token
- cliente-gql/src/auth/auth.controller

```
import { UseGuards } from "@nestjs/common";
import { Args, Mutation, Query, Resolver } from "@nestjs/graphql";

import { AuthService } from "./auth.service";

import { AuthGuard } from "./guards/auth.guard";

import { LoginInput } from "./dto/login.input";
import { RegisterInput } from "./dto/register.input";

import { GetAuthenticatedUserByToken } from
"../decorators/getUserByToken.decorator";
import { Auth } from "./entities/auth.entity";
import { Token } from "./entities/token.entity";
```

```
@Resolver(() => Auth)
export class AuthResolver {
  constructor(private readonly authService: AuthService) {}

  @Mutation(() => Auth, { name: "login" })
  login(@Args("loginInput") loginInput: LoginInput) {
    return this.authService.login(loginInput);
  }

  @Mutation(() => Auth, { name: "register" })
  register(@Args("registerInput") registerInput: RegisterInput) {
    return this.authService.register(registerInput);
  }

  @Query(() => Token, { name: "verifyJWT" })
  @UseGuards(AuthGuard)
  verifyToken(@GetAuthenticatedUserByToken() user: Token) {
    return this.authService.verifyToken(user);
  }
}
```

- La cliente-gql/src/entities/auth.entity es esta
- Debo declarar en el **@ObjectType** cada **@Field** con una función en la que debo declarar el tipo de retorno

```
import { Field, ObjectType } from "@nestjs/graphql";

@ObjectType()
export class Auth {
  @Field(() => String, {
    name: "id",
    description: "id of the user",
  })
  id: string;

  @Field(() => String, {
    name: "name",
    description: "name of the user",
  })
  name: string;

  @Field(() => String, {
    name: "email",
    description: "email of the user",
  })
  email: string;

  @Field(() => String, {
    name: "token",
    description: "jwt of the user",
    nullable: true,
```

```
  })
  token: string;
}
```

- El decorador para verificar el token es este
- Uso createParamDecorator, devuelve un token
- Extraigo elk context de Gql, y de este la request
- Retorno el token y el user

```
import { ExecutionContext, createParamDecorator } from "@nestjs/common";
import { GqlExecutionContext } from "@nestjs/graphql";

import { Token } from "../auth/entities/token.entity";

export const GetAuthenticatedUserByToken = createParamDecorator(
  (data, context: ExecutionContext): Token => {
    let request = context.switchToHttp().getRequest();

    if (!request) {
      const ctx = GqlExecutionContext.create(context);
      request = ctx.getContext().req;
    }

    return { token: request.token, user: request.user };
  },
);
```

- Para hacer una mutation desde Apollo Server

```
mutation Login($loginInput: LoginInput!){
  login(loginInput: $loginInput){
    id
    name
    email
  }
}

##VARIABLES

{
  "loginInput":{
    "email": "miemail@correo.com",
    "password": "mi_password"
  }
}
```

- El cliente-gql/controller se conecta al cliente-gql/service que se conecta con el auth-ms/auth.controller que se conecta con el aut-ms/auth.service
- El service de cliente.gql/login es este

```
import { Inject, Injectable } from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";

import { firstValueFrom } from "rxjs";
import { ServicesTokens } from "../enums";
import { LoginInput } from "./dto/login.input";
import { RegisterInput } from "./dto/register.input";
import { Auth } from "./entities/auth.entity";
import { Token } from "./entities/token.entity";

@Injectable()
export class AuthService {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}

  async login(loginInput: LoginInput): Promise<Auth> {
    const {
      id,
      token,
      email,
      username: name,
    } = await firstValueFrom( //me comunico con al auth-ms/controller
      this.authService.send("auth.login", { ...loginInput }),
    );

    return {
      id,
      email,
      name,
      token,
    };
  }
}
```

- Este controlador llama al microservicio auth usando el .send (espero una respuesta) del ClientProxy
- En el auth-ms/src/auth/controllers/auth.controller llamo al auth.login

```
import { Controller } from "@nestjs/common";
import { MessagePattern, Payload } from "@nestjs/microservices";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";

import { AuthService } from "./auth.service";
```

```
@Controller()
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @MessagePattern("auth.healt")
  healt() {
    return { status: 200, message: "ok" };
  }

  @MessagePattern("auth.verify.token")
  verifyToken(@Payload("token") token: string) {
    return this.authService.verifyToken(token);
  }

  //AQUI!!
  @MessagePattern("auth.login")
  login(@Payload() loginUserDto: LoginUserDto) {
    return this.authService.login(loginUserDto);
  }

  @MessagePattern("auth.register")
  register(@Payload() createUserDto: CreateUserDto) {
    return this.authService.register(createUserDto);
  }
}
```

- En el auth-ms/service tengo la lógica

```
import {
  Injectable,
  OnModuleInit,
  UnauthorizedException,
} from "@nestjs/common";
import { JwtService } from "@nestjs/jwt";
import { RpcException } from "@nestjs/microservices";

import { PrismaClient } from "@prisma/client";

import * as bcrypt from "bcrypt";

import { CreateUserDto } from "./dto/createUser.dto";
import { LoginUserDto } from "./dto/loginUser.dto";

@Injectable()
export class AuthService extends PrismaClient implements OnModuleInit {
  constructor(private readonly jwtService: JwtService) {
    super();
  }

  async onModuleInit() {
    await this.$connect();
```

```
  }

  private hashPassword(password: string) {
    const salt = bcrypt.genSaltSync(12);

    return bcrypt.hashSync(password, salt);
  }

  private verifyPassword(password: string, hash: string) {
    return bcrypt.compareSync(password, hash);
  }

  private validateUserEmail(email: string) {
    return this.user.findUnique({
      where: {
        email,
      },
    });
  }

  private signToken(payload: { id: string; email: string }) {
    return this.jwtService.sign({
      ...payload,
    });
  }

async login(loginUserDto: LoginUserDto) {
    const { password, email } = loginUserDto;

    const storedUser = await this.validateUserEmail(email);

    if (!storedUser || !this.verifyPassword(password, storedUser.password))
      throw new RpcException("Email or password is invalid");

    const {
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      password: _,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      active: __,
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      updatedAt: ___,
      ...user
    } = storedUser;

    return {
      ...user,     //método privado
      token: this.signToken({
        id: user.id,
        email,
      }),
    };
  }
}
```

- En auth-ms/src/auth/auth.module tengo registrado el módulo de Jwt en imports

```
import { Module } from "@nestjs/common";
import { JwtModule } from "@nestjs/jwt";

import { AuthController } from "./auth.controller";
import { AuthService } from "./auth.service";

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    JwtModule.register({
      secret: process.env.JWTSecret,
      signOptions: { expiresIn: "1h" },
    }),
  ],
})
export class AuthModule {}
```

- En auth-ms/src/auth/main.ts tengo registrado el microservicio

```
import { ValidationPipe } from "@nestjs/common";
import { NestFactory } from "@nestjs/core";
import { MicroserviceOptions, Transport } from "@nestjs/microservices";

import { AppModule } from "./app.module";
import { QueuesEnum } from "./enums/queue.enum";

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.RMQ,
      options: {
        urls: [process.env.RMQ_URL],
        queue: QueuesEnum.AuthQueue,
        queueOptions: {
          durable: true,
        },
        noAck: true,
      },
    },
  );

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
```

```
    }),
  );

  await app.listen();
}
bootstrap();
```

- Es en el gql-client/src/auth/auth.module donde registro el Ciente

```
import { Module } from "@nestjs/common";
import { ClientsModule, Transport } from "@nestjs/microservices";

import { AuthResolver } from "./auth.resolver";
import { AuthService } from "./auth.service";

import { envs } from "../config/envs";
import { QueuesEnum, ServicesTokens } from "../enums";

@Module({
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
  providers: [AuthResolver, AuthService],
})
export class AuthModule {}
```

- Cliente que usaré desde el client-gql/src/auth/auth.service.ts para comunicarme con el controlador de auth-ms con el auth-ms/auth.service inyectado
- Uso firstValueFrom para trabajar con promesas y async await y no observables

```
import { Inject, Injectable } from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";

import { firstValueFrom } from "rxjs";
import { ServicesTokens } from "../enums";
import { LoginInput } from "./dto/login.input";
```

```
import { RegisterInput } from "./dto/register.input";
import { Auth } from "./entities/auth.entity";
import { Token } from "./entities/token.entity";

@Injectable()
export class AuthService {
  constructor(
    @Inject(ServicesTokens.AUTH_SERVICE)
    private readonly authService: ClientProxy,
  ) {}

  async login(loginInput: LoginInput): Promise<Auth> {
    const {
      id,
      token,
      email,
      username: name,
    } = await firstValueFrom(
      this.authService.send("auth.login", { ...loginInput }),
    );

    return {
      id,
      email,
      name,
      token,
    };
  }

  async register(registerInput: RegisterInput): Promise<Auth> {
    const {
      id,
      token,
      email,
      username: name,
    } = await firstValueFrom(
      this.authService.send("auth.register", { ...registerInput }),
    );

    return {
      id,
      email,
      name,
      token,
    };
  }

  //la verificacion se hace a través del decorador
  verifyToken({ token, user }: Token): Token {
    return { token, user };
  }
}
```

- En el cliente-gateway/auth.resolver

```
@Query(() => Token, { name: "verifyJWT" })
@UseGuards(AuthGuard)
verifyToken(@GetAuthenticatedUserByToken() user: Token) { //uso el decorador
  return this.authService.verifyToken(user);
}
```

- cliente-gql/decorators Para extraer el token de la Request

```
import { ExecutionContext, createParamDecorator } from "@nestjs/common";
import { GqlExecutionContext } from "@nestjs/graphql";

import { Token } from "../auth/entities/token.entity";

export const GetAuthenticatedUserByToken = createParamDecorator(
  (data, context: ExecutionContext): Token => {
    let request = context.switchToHttp().getRequest();

    if (!request) {  //si no puedo extraer el context lo hago de esta otra manera
      const ctx = GqlExecutionContext.create(context);
      request = ctx.getContext().req;
    }

    return { token: request.token, user: request.user };
  },
);
```

En el cliente-gql/auth.service

```
                //desestructuro el token y el user
  verifyToken({ token, user }: Token): Token {
    return { token, user };
  }
```

- En Apollo Server obtengo el token con el register

```
mutation Register($registerInput: RegisterInput!){
  register(registerInput: $registerInput){
    id
    email
    token
  }
}
```

- Con @UseGuard(AuthGuard) le paso el token

```
query Query(){
  verifyToken
}

## VARIABLES
Headers/Authorization/Bearer jdhsudhskjdhskjdhskjdh
---------

## Otros archivos
```

- Si no le paso el token en las variables me dice Unauthorized
- Por eso uso el decorador, para captar el token
- Fuera de auth, en src tengo config/envs.ts

```
import "dotenv/config";

import * as joi from "joi";

const envSchema = joi
  .object({
    PORT: joi.number().required(),
    RMQ_URL: joi.string().required(),
  })
  .unknown(true); //dejo el unknown en true para el resto de variables de node

const { error, value } = envSchema.validate(process.env);

if (error) {
  throw new Error(`Config validation error: ${error.message}`);
}

interface Env {
  PORT: number;
  RMQ_URL: string;
}

const envVars: Env = value;

export const envs = {
  port: envVars.PORT as number,
  rmqUrl: envVars.RMQ_URL as string,
};
```

- Tengo también enums

- /queues

```
export enum QueuesEnum {
  AuthQueue = "AuthQueue",
  CoursesQueue = "CoursesQueue",
}
```

- /services

```
export enum ServicesTokens {
  COURSE_SERVICE = "COURSE_SERVICE",
  AUTH_SERVICE = "AUTH_SERVICE",
}
```

Courses

- El client-gql/src/courses/courses.module

```
import { Module } from "@nestjs/common";
import { ClientsModule, Transport } from "@nestjs/microservices";
import { envs } from "../config/envs";
import { QueuesEnum, ServicesTokens } from "../enums";
import { CoursesResolver } from "./courses.resolver";
import { CoursesService } from "./courses.service";

@Module({
  imports: [
    ClientsModule.register([
      {
        name: ServicesTokens.AUTH_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.AuthQueue,
          queueOptions: {
            durable: true,
          },
          noAck: true,
        },
      },
      {
        name: ServicesTokens.COURSE_SERVICE,
        transport: Transport.RMQ,
        options: {
          urls: [envs.rmqUrl],
          queue: QueuesEnum.CoursesQueue,
          queueOptions: {
```

```
            durable: true,
          },
          noAck: true,
        },
      },
    ]),
  ],
  providers: [CoursesResolver, CoursesService],
})
export class CoursesModule {}
```

- El resolver

```
import { UseGuards } from "@nestjs/common";
import { Args, Mutation, Query, Resolver } from "@nestjs/graphql";
import { Token } from "../auth/entities/token.entity";
import { AuthGuard } from "../auth/guards/auth.guard";
import { GetAuthenticatedUserByToken } from
"../decorators/getUserByToken.decorator";
import { CoursesService } from "./courses.service";
import { CreateCourseInput } from "./dto/create-course.input";
import { PaginationInput } from "./dto/pagination.input";
import { UpdateCourseInput } from "./dto/update-course.input";
import { Course } from "./entities/course.entity";

@Resolver(() => Course)
export class CoursesResolver {
  constructor(private readonly coursesService: CoursesService) {}

  @Query(() => [Course], { name: "getAllCourses" })
  @UseGuards(AuthGuard)
  findAll(@Args("paginationInput") paginationInput: PaginationInput) {
    return this.coursesService.findAll(paginationInput);
  }

  @Query(() => Course, { name: "getCourseById", nullable: true })
  @UseGuards(AuthGuard)
  findOne(@Args("id", { type: () => String }) id: string) {
    return this.coursesService.findOne(id);
  }

  @Mutation(() => Course)
  @UseGuards(AuthGuard)
  createCourse(
    @Args("createCourseInput") createCourseInput: CreateCourseInput,
    @GetAuthenticatedUserByToken() { user }: Token,
  ) {
    createCourseInput.author_id = user.id;

    return this.coursesService.create(createCourseInput);
  }
```

```
  @Mutation(() => Course)
  @UseGuards(AuthGuard)
  updateCourse(
    @Args("updateCourseInput") updateCourseInput: UpdateCourseInput,
  ) {
    return this.coursesService.update(updateCourseInput);
  }

  @Mutation(() => Course, { name: "deleteCourse" })
  @UseGuards(AuthGuard)
  removeCourse(@Args("id", { type: () => String }) id: string) {
    return this.coursesService.remove(id);
  }
}
```

- La entidad de Course en src/entities/course.entity.ts

```
import { Field, ObjectType } from "@nestjs/graphql";

@ObjectType()
export class Course {
  @Field(() => String, { description: "ID by course" })
  id: string;

  @Field(() => String, { description: "title by course" })
  title: string;

  @Field(() => String, { description: "description by course" })
  description: string;

  @Field(() => String, { description: "Author ID by course" })
  author_id: string;
}
```

- En el service me comunico con el microservicio

```
import { Inject, Injectable } from "@nestjs/common";
import { ClientProxy } from "@nestjs/microservices";
import { ServicesTokens } from "../enums";
import { CreateCourseInput } from "./dto/create-course.input";
import { PaginationInput } from "./dto/pagination.input";
import { UpdateCourseInput } from "./dto/update-course.input";

@Injectable()
export class CoursesService {
  constructor(
    @Inject(ServicesTokens.COURSE_SERVICE)
    private readonly courseService: ClientProxy,
```

```
  ) {}

  findAll(paginationInput: PaginationInput) {
    return this.courseService.send(
      { cmd: "get_all_courses" },
      { ...paginationInput },
    );
  }

  findOne(id: string) {
    return this.courseService.send({ cmd: "get_course" }, { id });
  }

  create(createCourseInput: CreateCourseInput) {
    return this.courseService.send(
      { cmd: "create_course" },
      { ...createCourseInput },
    );
  }

  update(updateCourseInput: UpdateCourseInput) {
    return this.courseService.send(
      { cmd: "update_course" },
      { ...updateCourseInput },
    );
  }

  remove(id: string) {
    return this.courseService.send({ cmd: "delete_course" }, { id });
  }
}
```

- En los dtos de course dentro de client-gql
- create-course.input.ts

```
import { Field, InputType } from "@nestjs/graphql";

@InputType()
export class CreateCourseInput {
  @Field(() => String, { description: "Course title" })
  title: string;

  @Field(() => String, { description: "Course description" })
  description: string;

  @Field(() => String, { description: "Course Author ID", nullable: true })
  author_id?: string;
}
```

- pagination.input.ts

```
import { Field, InputType, Int } from "@nestjs/graphql";

@InputType()
export class PaginationInput {
  @Field(() => Int, { nullable: true, description: "Page number" })
  page?: number;

  @Field(() => Int, { nullable: true, description: "Limit results" })
  limit?: number;
}
```

- update-course.input.ts

```
import { Field, InputType, PartialType } from "@nestjs/graphql";
import { CreateCourseInput } from "./create-course.input";

@InputType()
export class UpdateCourseInput extends PartialType(CreateCourseInput) {
  @Field(() => String)
  id: string;
}
```