BASICS gRPC 01

01 gRPC NODE - Server

- Creo el archivo .node-version (linux y mac)

```
20.10.0
```

> npm init -y

- Dependencias:

> npm i @ypes/google-protobuf grpc_tools_node_protoc_ts ts-node typescript -D

- Estas dependencias no son de desarrollo (estas nospermitirán generar el código)

> npm i @grpc/grpc-js @grpc/proto-loader

Construir archivo proto

- Creo la carpeta proto en la raíz
- Creo el archivo de ejemplo employees.proto
- Defino la sintaxis
- Defino los paquetes para la posterior autogeneración de código
- Defino la entidad
- Defino los servicios
- Defino los tipos de Request y Response

```
//declaro la sintaxis con la que voy a trabajar
syntax = "proto3";

//sección de paquetes (nombrará los paquetes autogenerados)
package employees;
option go_package = "grpc-node/basic/pb/employees";
option csharp_namespace = "Employees";

//Creo mi entidad  (tipando las propiedades)
message Employee{
    int32 id = 1;
    int32 badgeNumber= 2;
    string firstName = 3;
    string lastName = 4;
    float vacationAccrualRate = 5;
    float vacationAccrued = 6;
}
```

```
//Se pueden crear tipos de datosparticulares y ser asociados mediante contenedores

//Defino los servicios (lo que va a ser el puente entre el servidor y el cliente)
//Como va a generar una interfaz le coloco I al principio (por convención)
service IEmployeeService{
    rpc GetByBadgeNumber(GetByBadgeNumberRequest) returns (EmployeeResponse);
    //getByBadgeNumberRequest y EmployeeResponse son dos tipos de datos que
todavía no he construido
    //es un servicio unario porque a una petición me devuelve un solo response

    rpc Save(EmployeeRequest) returns (EmployeeResponse);

    rpc getAll(GetAllRequest) returns (stream EmployeeResponse); //devuelve un
stream, significa que es un servicio de streaming dlldo del servidor
                                                                    //cuando el
servidor considere que ha enviado todos los datos cerrará la conexión
                                                                    //me enviará el
flujo de datos de tods los EmployeeResponse, tantos  como tenga en  mi

    rpc AddPhoto (stream AddPhotoRequest) returns (AddPhotoResponse);
    //streaming del lado del ciente; el cliente va a enviar datos hasta que
considere que ha enviado todos los datos y cierre la conexión
    //Cuando cierre la  conexión el servidor va a entender que ha terminado y va a
emitir la Response

    rpc SaveAll (stream EmployeeRequest) returns (stream EmployeeResponse);
    //tendrmos streaming del lado del servidor  y del cliente
    //Cualquiera de los dos puede cerrar la conexión y una vez se cierra el
servidor procesa y devuelve la Response
    //El servidor también puede devolver una Response a cada petición, cada 10
peticiones, etc
   //Puedo enviar datos y recibirlos de fora arbitraria
 }


message GetByBadgeNumberRequest{
    int32 badgeNumber = 1; //contiene un entero con el nombre de atributo
badgeNumber que nos va a permitir recibir el número de badge de cualquier empleado
}

message EmployeeResponse{
    Employee employee = 1; //agrupa la abstracción de Employee
}

message EmployeeRequest{
    Employee employee = 1;
}

message AddPhotoRequest{
    bytes data = 1; //vamos  a eeenviar bytes de una foto en varios paquetes hasta
enviar toda la foto
}

message AddPhotoResponse {
```

```
    bool isOk = 1;
}

message GetAllRequest{}
```

- Para generar el autocódigo creo una carpeta enla raíz llamada scripts/proto-gen.sh
- Primero le digo ue es un archivo bash
- Le indico el directorio donde estoy almacenando los archivos .proto
- Escribo el comando: uso npm para invocar proto-loader que genera los archivos ts definidos en .proto con lasopciones de la librería grpc-js, y le indico el directorio de salida

```
#!/bin/bash

PROTO_DIR=./../proto

yarn proto-loader-gen-types --grpcLib=@grpc/grpc-js --outDir=proto/ proto/*.proto
```

- Creo el script en el json

```
 "scripts": {
    "proto:gen": "sh scripts/proto-gen.sh"
  }
```

- Lo ejecuto, me genera employees.ts y un archivo .ts por cada servicio
- employees.ts

```
import type * as grpc from '@grpc/grpc-js';
import type { MessageTypeDefinition } from '@grpc/proto-loader';

import type { IEmployeeServiceClient as _employees_IEmployeeServiceClient,
IEmployeeServiceDefinition as _employees_IEmployeeServiceDefinition } from
'./employees/IEmployeeService';

type SubtypeConstructor<Constructor extends new (...args: any) => any, Subtype> =
{
  new(...args: ConstructorParameters<Constructor>): Subtype;
};

export interface ProtoGrpcType {
  employees: {
    AddPhotoRequest: MessageTypeDefinition
    AddPhotoResponse: MessageTypeDefinition
    Employee: MessageTypeDefinition
    EmployeeRequest: MessageTypeDefinition
    EmployeeResponse: MessageTypeDefinition
    GetAllRequest: MessageTypeDefinition
```

```
    GetByBadgeNumberRequest: MessageTypeDefinition
    IEmployeeService: SubtypeConstructor<typeof grpc.Client,
_employees_IEmployeeServiceClient> & { service:
_employees_IEmployeeServiceDefinition }
  }
}
```

-En cada archivo hay una interfaz de la Request o la Resonse y el output -Por ejemplo, en Employee.ts

```
// Original file: proto/employees.proto


export interface Employee {
  'id'?: (number);
  'badgeNumber'?: (number);
  'firstName'?: (string);
  'lastName'?: (string);
  'vacationAccrualRate'?: (number | string);
  'vacationAccrued'?: (number | string);
}

export interface Employee__Output {
  'id'?: (number);
  'badgeNumber'?: (number);
  'firstName'?: (string);
  'lastName'?: (string);
  'vacationAccrualRate'?: (number);
  'vacationAccrued'?: (number);
}
```

- Si analizo el cliente que me ha generado (ctrl+click sobre _employees_IEmployeeServiceClient) puedo oservar una sobrecarga demétodos importante
- Porque podemos enviar metadatos, opciones, solo o con callbacks

```
export interface IEmployeeServiceClient extends grpc.Client {
  AddPhoto(metadata: grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  AddPhoto(metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  AddPhoto(options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  AddPhoto(callback: grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  addPhoto(metadata: grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
```

```
    addPhoto(metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
    addPhoto(options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
    addPhoto(callback: grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;

    GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;

    Save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    Save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    Save(argument: _employees_EmployeeRequest, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    Save(argument: _employees_EmployeeRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    save(argument: _employees_EmployeeRequest, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
    save(argument: _employees_EmployeeRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;

    SaveAll(metadata: grpc.Metadata, options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;
```

```
  SaveAll(options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;
  saveAll(metadata: grpc.Metadata, options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;
  saveAll(options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;

  getAll(argument: _employees_GetAllRequest, metadata: grpc.Metadata, options?:
grpc.CallOptions): grpc.ClientReadableStream<_employees_EmployeeResponse__Output>;
  getAll(argument: _employees_GetAllRequest, options?: grpc.CallOptions):
grpc.ClientReadableStream<_employees_EmployeeResponse__Output>;

}
```

Mensajes - Unary

- Hemos creado .proto con la definición de mi entidad y la interfaz de servicio, toca implementar esos servicios
- Crearemos una DB simulada
- Creo la carpeta src/EmployeesDB.ts

```
import {Employee} from '../proto/employees/Employee'

export class EmployeesDB{

    private employees: Employee [] =[]

    constructor(){
        this.employees = [
            {
                id: 1,
                badgeNumber: 2080,
                firstName: 'Grace',
                lastName: 'Decker',
                vacationAccrualRate: 2,
                vacationAccrued: 30
            },
            {
                id: 2,
                badgeNumber: 2030,
                firstName: 'Peter',
                lastName: 'Doherty',
                vacationAccrualRate: 4,
                vacationAccrued: 25
            },
            {
                id: 3,
```

```
                badgeNumber: 2010,
                firstName: 'John',
                lastName: 'Scofield',
                vacationAccrualRate: 5,
                vacationAccrued: 12
            }
        ]
    }


    public getEmployees(): Employee[]{
        return this.employees
    }

    public saveEmployee(employee: Employee): Employee{
        this.employees.push(employee)
        return employee
    }

    public getEmployeeBybadgeNumber(badgeNumber: number): Employee | undefined{
        const employee = this.employees.find(employee => employee.badgeNumber ===
badgeNumber)
        if(!employee){
            throw new Error('Employee not found')
        }
        return employee
    }

}
```

- En los archivos generados (concretamente en proto/IEmployeeService) encuentro la interfaz del servicio

```
export interface IEmployeeServiceHandlers extends
grpc.UntypedServiceImplementation {
  AddPhoto: grpc.handleClientStreamingCall<_employees_AddPhotoRequest__Output,
_employees_AddPhotoResponse>;

  GetByBadgeNumber:
grpc.handleUnaryCall<_employees_GetByBadgeNumberRequest__Output,
_employees_EmployeeResponse>;

  Save: grpc.handleUnaryCall<_employees_EmployeeRequest__Output,
_employees_EmployeeResponse>;

  SaveAll: grpc.handleBidiStreamingCall<_employees_EmployeeRequest__Output,
_employees_EmployeeResponse>;

  getAll: grpc.handleServerStreamingCall<_employees_GetAllRequest__Output,
_employees_EmployeeResponse>;

}
```

-Al tipar el servicio en EmployeeService.ts (el archivo que he cread para implementar los servicios) me marca error porque faltanlosmétodos

- Con quick fix del IDE lo arreglo al instante (ojo que el servicio no lo estoy construyendo como una clase!)

```
import { ServerReadableStream, sendUnaryData, ServerUnaryCall, ServerDuplexStream,
ServerWritableStream } from "@grpc/grpc-js";
import { AddPhotoRequest__Output } from "../proto/employees/AddPhotoRequest";
import { AddPhotoResponse } from "../proto/employees/AddPhotoResponse";
import { EmployeeRequest__Output } from "../proto/employees/EmployeeRequest";
import { EmployeeResponse } from "../proto/employees/EmployeeResponse";
import { GetAllRequest__Output } from "../proto/employees/GetAllRequest";
import { GetByBadgeNumberRequest, GetByBadgeNumberRequest__Output } from
"../proto/employees/GetByBadgeNumberRequest";
import { IEmployeeServiceHandlers } from "../proto/employees/IEmployeeService";
import { EmployeesDB } from "./EmployeesDB";


const _employeesDB = new EmployeesDB()


const  EmployeesService : IEmployeeServiceHandlers={

    AddPhoto: function (call: ServerReadableStream<AddPhotoRequest__Output,
AddPhotoResponse>, callback: sendUnaryData<AddPhotoResponse>): void {
        throw new Error("Function not implemented.");
    },

    GetByBadgeNumber: function (call:
ServerUnaryCall<GetByBadgeNumberRequest__Output, EmployeeResponse>, callback:
sendUnaryData<EmployeeResponse>): void {
        const req = call.request as GetByBadgeNumberRequest //lo casteo al objeto
que necesito trabajar
        //si reviso os archivos generados, GetByBadgNumberReuest.ts tiene un
badgeNumber

        if(req.badgeNumber){
            const badgeNumber = req.badgeNumber
            const employee = _employeesDB.getEmployeeBybadgeNumber(badgeNumber)
                callback(null, {employee})
        }

        //si no hay employee volvemos a usar elcallback para indicar el error
        callback({
            name: "badgeNumber is undefined",
            message:"invalid input"
        }, {employee: undefined})

    },
```

```
    Save: function (call: ServerUnaryCall<EmployeeRequest__Output,
EmployeeResponse>, callback: sendUnaryData<EmployeeResponse>): void {
        throw new Error("Function not implemented.");
    },
    SaveAll: function (call: ServerDuplexStream<EmployeeRequest__Output,
EmployeeResponse>): void {
        throw new Error("Function not implemented.");
    },
    getAll: function (call: ServerWritableStream<GetAllRequest__Output,
EmployeeResponse>): void {
        throw new Error("Function not implemented.");
    }
}

export {
    EmployeesService
}
```

-Para prbar que el servicio funcione debemos crear el servidor! -En la raíz creo sever.ts

```
import * as protoLoader from "@grpc/proto-loader"
import * as grpc from "@grpc/grpc-js"
import path from 'path'
import {ProtoGrpcType} from './proto/employees'
import {EmployeesService} from './src/EmployeesService'

const PORT = 8082

//ubicacion archivo proto
const PROTO_FILE="./proto/employees.proto"


//necesitamos crear una instancia de objeto de grpc usando employees.proto
//para ello importamos @grpc/proto-loader
const packageDefinition = protoLoader.loadSync(path.resolve(__dirname,
PROTO_FILE)) //importamos el archivo proto

//con packageDefinition podremos obtener la definición de tipos grpc
//para ello importamos @grpc/grpc-js
//lo  tipo como unknown para poderlo castear con el tipo de la interface generada
desde employees.proto volcada en employees.ts
const grpcObj= (grpc.loadPackageDefinition(packageDefinition) as unknown) as
ProtoGrpcType //ProtoGrpcType es una interface que tiene los tipos

//de todos los mensajes

//EN el main hago todos los llamados a las configuraciones particulares
function main(){
    //podemos agregar tantos servicios como queramos divididos por entidades,
ahora solo tenemos una
    const server = getServer()
    const serverCredentials= grpc.ServerCredentials.createInsecure()//server sin
```

```
autenticación

    //usamos bindAsync para conectar el server al localhost:puerto, el segundo
parametro  es el tipo de auth (aqui sin auth)
    //el tercer parámetro es un callback que me devuelve el error o el puerto
donde se pudo conectar
    server.bindAsync(`0.0.0.0:${PORT}`, serverCredentials, (err, port)=>{
            if(err){
                console.log(err)
                return
            }
            console.log(`Conectado en el puerto ${port}`)
            //server.start() --->deprecated!
    })
}

//configurar el server de grpc
function getServer(){
    const server = new grpc.Server()

    //este método recibe la definicion de un servicio como primer parámetro
    //y la implementación de ese servicio como segundo parámetro
(EmployeesService.ts)
    //Si exploro employees.ts observo la definición delservicio en service:
_employees_IEmployeeServiceDefinition
    //En grpcObj tengo todo
    server.addService(grpcObj.employees.IEmployeeService.service,
EmployeesService)


    return server
}

main()
```

- Creo el script para iniciar el server

```
"start:server":"ts-node server.ts"
```

- Usamos POSTMAN para usarlo como cliente
- import a proto file e importo employees com una nueva API
- Ahora puedo seleccionar un método
- Le añado badgeNumber al mensaje
- *NOTA*: algo en el código de este servicio no está bien. Los servicios posteriores funcionan pero este no

---

Mensajes - Server Streaming

- Vamos con obtener todos los empleados

```
 getAll: function (call: ServerWritableStream<GetAllRequest__Output,
EmployeeResponse>): void {
        throw new Error("Function not implemented.");
    }
```

- Tenemos un servicio grpc del tipo streaming del lado del servidor que recibe como objeto de petición el GetAllReuest y devuelve una EmployeeResponse (no un arreglo)
- Cada vez que mandemos un mensaje a través del streaming mandaremos un empleado que el cliente va a capturar en su conjunto como un arreglo
- call tiene unmetodo write que recibe un chunk de respuesta (de tipo EmployeeResponse)
- Hay que terminar la conexión

```
getAll: function (call: ServerWritableStream<GetAllRequest__Output,
EmployeeResponse>): void {
    const employees  = _employeesDB.getEmployees()
    employees.forEach(employee=>{
        call.write({employee})
    } )

    call.end()

}
```

- En la respuesta observo que cada empleado es un envío por parte del servidor, en lugar de venir tdo en un arreglo
- La conexión durará hasta que el servidor considere que ha enviado toda la data

---

Mensajes - Client Streaming

- Subiremos una foto con AddPhoto

```
AddPhoto: function (call: ServerReadableStream<AddPhotoRequest__Output,
AddPhotoResponse>, callback: sendUnaryData<AddPhotoResponse>): void {
    throw new Error("Function not implemented.");
}
```

- El servidor está recibiendo como parámetro de call un stream de lectura ServerReadableStream, por el que se nos va a enviar un chunk de información, en algún momento se va a detener ese envío de tipo AddPhotoRequest y devolver un dato unario de tipo AddPhotoResponse en el callback
- Por eso es streaming del lado del cliente, porque es este quien envía información hasta un momento determinado -EmployeesService.ts

```

```

```
AddPhoto: function (call: ServerReadableStream<AddPhotoRequest__Output,
AddPhotoResponse>, callback: sendUnaryData<AddPhotoResponse>): void {
        //guardamos el stream en un archivo
        const writableStream  = fs.createWriteStream('upload_photo.png')

        //lo que hareos cada vez que lleguen datos
        call.on('data', (request:AddPhotoRequest)=>{ //si busco la interfaz de
AddPhotoRequest contine data de tipo Buffer

            writableStream.write(request.data)
        })

        //lo que haremos cuando terminen de llegar esos datos
        call.on('end', ()=>{
            //guardará todos los bytes dentro de la ruta upload_photo
            writableStream.end()
            console.log("File uploaded successfully!!")
        })

    }
```

- Para el streaming del lado del cliente para una foto no puedo hacerlo con POSTMAN
- Crearé un cliente para ello con un script

Mensajes - Bidireccional Streaming

- Vamos a implmentar un servicio full duplex bidireccional de streaming
- Usaremos SaveAll ara el ejemplo, dónde el cliente va a ir enviando employees y el server irá devolviendo información

```
SaveAll: function (call: ServerDuplexStream<EmployeeRequest__Output,
EmployeeResponse>): void {
    throw new Error("Function not implemented.");
}
```

- Tenemos un ServerDuplexStream, nos indica que tenemos un canalduplex de streaming donde el cliente y el servidor van a poder enviar información de tipo EmployeeRequest con una respuesta de tipo EmployeeResponse
- La estrategia es muy similar a la de AddPhoto

```
 SaveAll: function (call: ServerDuplexStream<EmployeeRequest__Output,
EmployeeResponse>): void {
        let count= 0

        call.on('data', (request: EmployeeRequest)=>{
            if(request.employee){
```

```
                const employee = request.employee
                _employeesDB.saveEmployee(employee)
                count ++
                call.write({employee})
            }
        })

        call.on('end', ()=>{
            console.log(`${count} employees saved`)
            call.end() //siempre cerrar la conexión!!!
        })
    },
```

- Si ahora voy a POSTMAN y edoy a invoke ABRE EL STREAMING pero no aparece nada en consola
- Esto solo abrió la conexión. Debo darle a SEND para enviar la data y acabar con END STREAMING
- El cliente puede enviar tantos datos como quiera y el servidor enviar una respuesta cuando lo considere necesario
- No necesariamente debe enviar una respuesta a cada petición, puedo enviar un video en varios chunks y cuando tenga un 10% enviar una notificación

---

Establecer conexión segura

- Creo la carpeta ssl y dentro un archivo ssl.sh con este código

```
#!/bin/bash

rm *.pem
rm *.srl
rm *.cnf
rm *.crt
rm *.key
rm *.csr

# 1. Generate CA's private key and self-signed certificate
openssl req -x509 -newkey rsa:4096 -days 365 -nodes -keyout ca-key.pem -out ca-
cert.pem -subj "/C=FR/ST=Occitanie/L=Toulouse/O=Test
Org/OU=Test/CN=*.test/emailAddress=test@gmail.com"

echo "CA's self-signed certificate"
openssl x509 -in ca-cert.pem -noout -text

# 2. Generate web server's private key and certificate signing request (CSR)
openssl req -newkey rsa:4096 -nodes -keyout server-key.pem -out server-req.pem -
subj "/C=FR/ST=Ile de France/L=Paris/O=Server
TLS/OU=Server/CN=*.tls/emailAddress=tls@gmail.com"

# Remember that when we develop on localhost, It’s important to add the IP:0.0.0.0
as an Subject Alternative Name (SAN) extension to the certificate.
echo "subjectAltName=DNS:*.tls,DNS:localhost,IP:0.0.0.0" > server-ext.cnf
# Or you can use localhost DNS and grpc.ssl_target_name_override variable
```

```
# echo "subjectAltName=DNS:localhost" > server-ext.cnf

# 3. Use CA's private key to sign web server's CSR and get back the signed
certificate
openssl x509 -req -in server-req.pem -days 60 -CA ca-cert.pem -CAkey ca-key.pem -
CAcreateserial -out server-cert.pem -extfile server-ext.cnf

echo "Server's signed certificate"
openssl x509 -in server-cert.pem -noout -text

# 4. Generate client's private key and certificate signing request (CSR)
openssl req -newkey rsa:4096 -nodes -keyout client-key.pem -out client-req.pem -
subj "/C=FR/ST=Alsace/L=Strasbourg/O=PC
Client/OU=Computer/CN=*.client.com/emailAddress=client@gmail.com"

# Remember that when we develop on localhost, It’s important to add the IP:0.0.0.0
as an Subject Alternative Name (SAN) extension to the certificate.
echo "subjectAltName=DNS:*.client.com,IP:0.0.0.0" > client-ext.cnf

# 5. Use CA's private key to sign client's CSR and get back the signed certificate
openssl x509 -req -in client-req.pem -days 60 -CA ca-cert.pem -CAkey ca-key.pem -
CAcreateserial -out client-cert.pem -extfile client-ext.cnf

echo "Client's signed certificate"
openssl x509 -in client-cert.pem -noout -text
```

- Nos generarálos ceryificados que necesitamos
- Esto solo es útil en localhost
- Para ejecutarlo creo el script

```
"ssl:gen": "cd ssl && chmod +x ssl.sh && sh ssl.sh"
```

- Dentro de la carpeta ssl creo SSLService.ts en el que generaremos las credenciales de seguridad para el servidor y para el cliente

```
import { ServerCredentials } from "@grpc/grpc-js";
import *as fs from 'fs'
import path from 'path'

export class SSLService{
    static getServerCredentials():ServerCredentials{
        const serverCert = fs.readFileSync(path.resolve(__dirname,
'../../ssl/server-cert.pem')) //importamos  el certificado
        const serverKey =  fs.readFileSync(path.resolve(__dirname,
'../../ssl/server-key.pem'))  //importamos la clave

        //el primer parámetro es el root del Buffer, lo mandamos como nulo
        //el segundo es un diccionario de certificados y claves
        //le  envio false como tercero para que no chequee  el certificado del
```

```
cliente, eso lo haremos más adelante
        return ServerCredentials.createSsl(null,[{cert_chain:serverCert,
private_key:serverKey}], false)


    }
}
```

- Para configurarlo en el server.ts

```
function main(){

    const server = getServer()
    const serverCredentials= SSLService.getServerCredentials()
    server.bindAsync(`0.0.0.0:${PORT}`, serverCredentials, (err, port)=>{
            if(err){
                console.log(err)
                return
            }
            console.log(`Conectado en el puerto ${port}`)

    })
}
```

- Proximamente crearemos el cliente gRPC, aprenderemosaimplementar los métodos y crear unaconexión segura dellado del cliente

---

gRPC NODE - Client

- Creo el client.ts en la raíz
- Uso parte del código de server.ts como el puerto, la ubicación del archivo proto, la definición del paquete (packageDefinition) y la definición de los objetos de grpc (grpcObj)

```
import * as protoLoader from "@grpc/proto-loader"
import * as grpc from "@grpc/grpc-js"
import path from 'path'
import {ProtoGrpcType} from './proto/employees'

const PORT = 8082

const PROTO_FILE="./proto/employees.proto"


const packageDefinition = protoLoader.loadSync(path.resolve(__dirname,
PROTO_FILE)) //importamos el archivo proto


const grpcObj= (grpc.loadPackageDefinition(packageDefinition) as unknown) as
```

```
ProtoGrpcType

//haremos que el server funcione de momento con las credenciales inseguras
//lo modifico en server.ts tambien, luego lo cambio
const channelCredentials= grpc.credentials.createInsecure() //notar que aqui son
.credentials no ServerCredentials
const client= new grpcObj.employees.IEmployeeService(`0.0.0.0:${PORT}`,
channelCredentials)

//creamos la conexión del cliente
//ncesitamos  undeadLine, un tiempo  de espera para que el cliente se conecte (5-
10 segundos)
const deadLine= new Date()
deadLine.setSeconds(deadLine.getSeconds()+10) //le añado 10 segundos

//esperamos a que el cliente este listo para conectarse
client.waitForReady(deadLine, (err)=>{
    if(err){
        throw new Error('Error  creating Client')

    }
    onClientReady()
})

function onClientReady(){
    console.log('working!')
}
```

- Modifico las credenciales del server para trabajar de momento de forma insegura

```
function main(){
    const server = getServer()
    //podemos agregar tantos servicios como queramos divididos por entidades,
ahora solo tenemos una
    const serverCredentials= grpc.ServerCredentials.createInsecure()//server sin
autenticación

    server.bindAsync(`0.0.0.0:${PORT }`, serverCredentials, (err,port)=>{
        if(err){
            console.error(err)
            return
        }
        console.log(`Server running at port ${port}`)
    })
}
```

- Genero el script para inicializar el cliente

```
"start:client":"ts-node client.ts",
```

- Por ahora solo nos devuelve el console.log y se cierra la conexión
- Porué dispongo de los métodos desde client?
- Al crearlo apunta a la nterfaz de servicio

```
const client= new grpcObj.employees.IEmployeeService(`0.0.0.0:${PORT}`,
channelCredentials)
```

- Si miro la interfaz

```
export interface ProtoGrpcType {
  employees: {
    AddPhotoRequest: MessageTypeDefinition
    AddPhotoResponse: MessageTypeDefinition
    Employee: MessageTypeDefinition
    EmployeeRequest: MessageTypeDefinition
    EmployeeResponse: MessageTypeDefinition
    GetAllRequest: MessageTypeDefinition
    GetByBadgeNumberRequest: MessageTypeDefinition            //puedo clicar
encima de esta interfaz
    IEmployeeService: SubtypeConstructor<typeof grpc.Client,
_employees_IEmployeeServiceClient> & { service:
_employees_IEmployeeServiceDefinition }
  }
}
```

-Si voy a _employees_IEmployeeServiceClient veo como me define todo estos métodos en IEmployeeService

```
export interface IEmployeeServiceClient extends grpc.Client {
  AddPhoto(metadata: grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  AddPhoto(metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  AddPhoto(options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  AddPhoto(callback: grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  addPhoto(metadata: grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  addPhoto(metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  addPhoto(options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_AddPhotoResponse__Output>):
```

```
grpc.ClientWritableStream<_employees_AddPhotoRequest>;
  addPhoto(callback: grpc.requestCallback<_employees_AddPhotoResponse__Output>):
grpc.ClientWritableStream<_employees_AddPhotoRequest>;

  GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  GetByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, metadata:
grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  getByBadgeNumber(argument: _employees_GetByBadgeNumberRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;

  Save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  Save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  Save(argument: _employees_EmployeeRequest, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  Save(argument: _employees_EmployeeRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, options:
grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  save(argument: _employees_EmployeeRequest, metadata: grpc.Metadata, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  save(argument: _employees_EmployeeRequest, options: grpc.CallOptions, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;
  save(argument: _employees_EmployeeRequest, callback:
grpc.requestCallback<_employees_EmployeeResponse__Output>): grpc.ClientUnaryCall;

  SaveAll(metadata: grpc.Metadata, options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;
  SaveAll(options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;
  saveAll(metadata: grpc.Metadata, options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
```

```
_employees_EmployeeResponse__Output>;
  saveAll(options?: grpc.CallOptions):
grpc.ClientDuplexStream<_employees_EmployeeRequest,
_employees_EmployeeResponse__Output>;

  getAll(argument: _employees_GetAllRequest, metadata: grpc.Metadata, options?:
grpc.CallOptions): grpc.ClientReadableStream<_employees_EmployeeResponse__Output>;
  getAll(argument: _employees_GetAllRequest, options?: grpc.CallOptions):
grpc.ClientReadableStream<_employees_EmployeeResponse__Output>;

}
```

- Entonces, el cliente tiene el método de getByBadgeNumber que es el qe usaremos ahora

```
import * as protoLoader from "@grpc/proto-loader"
import * as grpc from "@grpc/grpc-js"
import path from 'path'
import {ProtoGrpcType} from './proto/employees'

const PORT = 8082

const PROTO_FILE="./proto/employees.proto"


const packageDefinition = protoLoader.loadSync(path.resolve(__dirname,
PROTO_FILE)) //importamos el archivo proto


const grpcObj= (grpc.loadPackageDefinition(packageDefinition) as unknown) as
ProtoGrpcType

//haremos que el server funcione de momento con las credenciales inseguras
//lo modifico en server.ts tambien, luego lo cambio
const channelCredentials= grpc.credentials.createInsecure() //notar que aqui son
.credentials no ServerCredentials
const client= new grpcObj.employees.IEmployeeService(`0.0.0.0:${PORT}`,
channelCredentials) //el cliente apunta a la interfaz del servicio.

//dispongo  de todos los métodos

//creamos la conexión del cliente
//ncesitamos  undeadLine, un tiempo  de espera para que el cliente se conecte (5-
10 segundos)
const deadLine= new Date()
deadLine.setSeconds(deadLine.getSeconds()+10) //le añado 10 segundos

//esperamos a que el cliente este listo para conectarse
client.waitForReady(deadLine, (err)=>{
    if(err){
        throw new Error('Error  creating Client')

    }
```

```
    onClientReady()
})


//Creo el método usando client
const getEmployeeByBadgeNumber = ()=>{
    client.getByBadgeNumber({badgeNumber: 2010}, (err, response)=>{
        if(err){
            console.error(err)
            return
        }
        console.log(`Employee: with badgeNumber ${response?.employee?.badgeNumber}
has name ${response?.employee?.firstName}`)

    })
}

//lo añado a la función que se ejecuta en la conexión
function onClientReady(){
    getEmployeeByBadgeNumber()
}
```

- Ejecuto el server y el cliente con los scripts correspondientes
- Sigo conlo otros métodos, save.
- En el server, save es así

```
Save: function (call: ServerUnaryCall<EmployeeRequest__Output, EmployeeResponse>,
callback: sendUnaryData<EmployeeResponse>): void {
    const req = call.request as EmployeeRequest
    if(req.employee){
        const employee = req.employee
        _employeesDB.saveEmployee(employee)
        callback(null, {employee})
    }

    callback({
        name: "employee is undefined",
        message:"invalid input"
    }, null)
},
```

- En el cliente es así

```
const saveEmployee= ()=>{
    const employee:  Employee ={
        id: 1000,
        badgeNumber: 2080,
        firstName: "John",
```

```
    }

    client.save({employee}, (err, response)=>{
        if(err){
            console.error(err)
            return
        }
        console.log(`Employee saved with badgeNumber:
${response?.employee?.badgeNumber}`)
    })

}

function onClientReady(){
    //getEmployeeByBadgeNumber()
    saveEmployee()
}
```

Mensajes - Server Streaming

- Construiremos getAll
- Si yo poso el cursor encima de getAll puedo ver con la ayuda del IDE que es un grpc.CientReadableStream que nos va a devolver un EmployeeResponse

```
const getAll= ()=>{
    client.getAll
}
```

- Empty es un objeto vacío que viene de protobuf. Con esto envío un objeto vacío en la petición

```
import {Empty} from 'google-protobuf/google/protobuf/empty_pb'
```

- El método

```
const getAll= ()=>{
    const stream = client.getAll(new Empty()) //este stream nos va a permitir
escuchar eventos

    const employees:Employee[]= []

    stream.on('data', (response)=>{
        const employee = response.employee
        employees.push(employee)
        console.log(`Fetch employeewith badgeNumber ${employee.badgeNumber}`)
//cada vez que obtengamos datos se imprimirá
        //cada chunk de datos será un empleado
    })
```

```
    stream.on('error', (err)=>{
        //en un entorno de producción aqui se hacen políticas de reintentos,
reestablecer la conexión,o usar otros frameworks
        //como temporal.io para que las ejecuciones que no se completaron de forma
exitosa se completen aunque haya habido un error
        console.log(err)
    })
    stream.on('end', ()=>{
        console.log(`${employees.length} total employees`)
    })
}
```

Mensajes - Client Streaming

- AddPhoto desde el cliente
- Hago lo mismo, escribo client.AddPhoto y observo los tipos posando el cursor encima

```
(method) IEmployeeServiceClient.AddPhoto(metadata: grpc.Metadata, options:
grpc.CallOptions, callback: grpc.requestCallback<AddPhotoResponse__Output>):
grpc.ClientWritableStream<AddPhotoRequest>
```

- Observo que la response es de tipo grpc.ClientWritableStream
- Para implementar este métodono necesitamos nada, podemos enviar un callback vacío

```
const addPhoto =()=>{
   const stream= client.AddPhoto(()=>{})
    //con fs vamos a crear un readstream que nospermitirá leer un png,
descomponerlo en chunks y enviárselo al server
    const fileStream= fs.createReadStream('./badgePhoto.png')

    fileStream.on('data', (chunk)=>{
        stream.write({data: chunk})
    })

    fileStream.on('end',  ()=>{
        stream.end()
    })

}
```

- Esto me crea un archivo en la raíz llamado upload_photo.png

---

Mensajes - Bidirectional Streaming

- Por cada vez que se guarde un cliente el servidor me va a devolver una respuesta

- Alguien tiene que cerrar la conexión (el cliente o el servidor)
- Por la naturaleza de este método, es elcliente quien está enviando. Cuando terminecierra el streaming - client.ts

```
const saveAll=()=>{
    const stream= client.saveAll() //esto crea el canal

    const employeesToSave =[
        {
            id: 4,
            badgeNumber: 2090,
            firstName: 'Johnee',
            lastName: 'Scofieldaaa',
            vacationAccrualRate: 50,
            vacationAccrued: 120
        },
        {
            id: 5,
            badgeNumber: 2023,
            firstName: 'Johneet',
            lastName: 'Scofieldaaarrr',
            vacationAccrualRate: 50,
            vacationAccrued: 120
        }

    ]

    const employees:Employee []= []
    stream.on('data',(response)=>{
        employees.push(response.employee)
        console.log('employee saved!')
    })
    stream.on('error',(err)=>{
        console.log(err)
    })

    stream.on('end',()=>{
        console.log(employees.length)
    })

    employeesToSave.forEach(employee=>{
        stream.write({employee})
    })

    //cierro la conexión
    stream.end()

}

function onClientReady(){
    //getEmployeeByBadgeNumber()
    //saveEmployee()
```

```
    //getAll()
    //addPhoto()
    saveAll()
}
```

Estalecer conexión segura

- Habilitamos la conexion segura en el server

```
function main(){
    const server = getServer()

    const serverCredentials= SSLService.getServerCredentials()

    server.bindAsync(`0.0.0.0:${PORT }`, serverCredentials, (err,port)=>{
        if(err){
            console.error(err)
            return
        }
        console.log(`Server running at port ${port}`)

    })
}
```

- Creo otro método estático en SSLService

```
import { ChannelCredentials, ServerCredentials } from "@grpc/grpc-js";
import *as fs from 'fs'
import path from 'path'

export class SSLService{
    static getServerCredentials():ServerCredentials{
        const serverCert = fs.readFileSync(path.resolve(__dirname,
'../../ssl/server-cert.pem')) //importamos  el certificado
        const serverKey =  fs.readFileSync(path.resolve(__dirname,
'../../ssl/server-key.pem'))  //importamos la clave

        //el primer parámetro es el root del Buffer, lo mandamos como nulo
        //el segundo es un diccionario de certificados y claves
        //le  envio false como tercero para que no chequee  el certificado del
cliente, eso lo haremos más adelante
        return ServerCredentials.createSsl(null,[{cert_chain:serverCert,
private_key:serverKey}], false)


    }

    static getChannelCredentials(): ChannelCredentials{
```

```
        const rootCert = fs.readFileSync(path.resolve(__dirname, './../ssl/ca-
cert.pem'))

        return ChannelCredentials.createSsl(rootCert)
    }
}
```

- LLamo el método en el cliente

```
const channelCredentials= SSLService.getChannelCredentials()
const client=new grpcObj.employees.IEmployeeService(`0.0.0.0:${PORT}`,
channelCredentials)
```

DRIVE YOUR CITY - MICROSERVICES

Drive Your City

- Aplicación de microservicios para uso de bicicletas en la ciudad
- Tenemos un dock con varias bicis aparcadas, con la aplicación podemos desbloquear la bici y desplazarnos
- Se contará el número de km
- Como podemos estar hablando de miles de bicicletas necesitamos un gran desempeño que sea escalable, y con baja latencia

---

Arquitectura

- Capa más externa:
  - Cliente:
    - BikeIoT: se comunica con el servidor
- Siguiente capa:
  - Microservicios: (responsabilidad única, CRUD)
    - RideService: para controlar mis viajes. Comenzar un viaje requiere una orquestación de varios eventos.
      - Requiere verificar que la bici este disponible, que la persona tenga una cuenta habilitada, etc
    - DockService: crear un dock donde aparcar la bici, etc
    - BikeService: para controlar las bicis. Añadir o retirar una bici a un dock
- Capa más interna: usaremos CockroachDB (altamente escalable)
  - Load Balancer:
    - DB-1
      - DB-2
      - DB-3

- Podemos pensar que el microservicio de dock lo usaremos pocas veces a lo largo del día (crear, eliminar, actualizar un dock)
- Sin embargo ride requiere una escalabilidad alta, ya que en hora punta puede tener una alta demanda
- gRPC es adecuado por los requerimientos de calidad
- Usaremos gRPC para comunicarnos entre microservicios
- Vamos a tener un cliente en ride que se va a comunicar con el dock, dock se comunicará con bike y así

Scaffolding y Docker

- Crearemos las carpetas que usaremos a lo largo de todo el trabajo
- Instanciarermos la DB y el balanceador de carga
- El proyecto se llama DriveYourCity
- Creamos en la raíz docker-compose.yml

```
version: '3.9'

services:

  roach-0:
    container_name: roach-0
    hostname: roach-0
    image:  cockroachdb/cockroach-unstable:v23.2.0-beta.1
    # --insecure porque no nos complicaremos con autenticación
    # al crear un cluster nos uniremos a varias instancias, para ello uso join
    # establezco el puerto 26257 y también el de advertise que necesita
cockroachDB
    # limito la cantidad de memoria que van a usar nuestros componentes y el caché
    command: start --insecure --join=roach-0,roach-1,roach-2 --listen-addr=roach-
0:26257 --advertise-addr=roach-0:26257 --max-sql-memory=.25 --cache=.25
    environment:
      - 'ALLOW_EMPTY_PASSWORD=yes' # le permito conectarse sin password

  roach-1:
    container_name: roach-1
    hostname: roach-1
    image:  cockroachdb/cockroach-unstable:v23.2.0-beta.1
    command: start  --insecure --join=roach-0,roach-1,roach-2 --listen-addr=roach-
1:26257 --advertise-addr=roach-1:26257 --max-sql-memory=.25 --cache=.25
    environment:
      - 'ALLOW_EMPTY_PASSWORD=yes'

  roach-2:
    container_name: roach-2
    hostname: roach-2
    image:  cockroachdb/cockroach-unstable:v23.2.0-beta.1
    command: start  --insecure --join=roach-0,roach-1,roach-2 --listen-addr=roach-
2:26257 --advertise-addr=roach-2:26257 --max-sql-memory=.25 --cache=.25
    environment:
      - 'ALLOW_EMPTY_PASSWORD=yes'
```

```
  init: # con init nos aseguraremos de que todo esté inicializado
    container_name: init
    image:  cockroachdb/cockroach-unstable:v23.2.0-beta.1 # uso la misma imagen de
cockroach para inicializar este container
    command: init --host=roach-0 --insecure # para inicializar el cluster completo
basta con incializar uno de los nodos que ya está estable
    depends_on:
      - roach-0

  lb: # balanceador de carga para acceder a estas instancias
    container_name: lb
    hostname: lb
    build: haproxy # este balanceador se llama haproxy
    ports:  # usa 3 puertos
      - "26000:26000" # conecta a todo el cluster
      - "8080:8080" # cockroachDB ofrece una interfaz en el 8080
      - "8081:8081" # para que los nodos entre diferentes instancias de
cockroachDB estén actualizados usamos 8081
    depends_on: # dependerá de que estas instancias estén inicializadas, por ello
crearé init para asegurarme de ello
      - roach-0
      - roach-1
      - roach-2

  client: # para poder ejecutar comandos al cluster creamos un cliente
    container_name: client
    hostname: client
    image:  cockroachdb/cockroach-unstable:v23.2.0-beta.1
    entrypoint: ["/usr/bin/tail", "-f", "/dev/null"] # ubicación del comando que
nos va a permitir acceder a esta instancia de cockroachDB, ver logs, etc
```

- Para crear el container de haproxy creo una nueva carpeta en la raíz del proyecto llamada haproxy
- Creo el Dockerfile donde defino el contenedor del balanceador de carga

```
FROM haproxy:lts-bullseye // usamos esta imagen

LABEL maintainer="artemervits at gmail dot com" // nos permitirá identificar este
container (asigno valor por defecto)

COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg // copiaremos nuestro archivo
de configuración haproxy.cfg en la ubicación del container /usr/...

// expongo los tres puertos que necesito (que ya he definido antes)
EXPOSE 26257
EXPOSE 8080
EXPOSE 8081
```

- Dentro de la misma carpeta haproxy creo haproxy.cfg copn el código por defecto

```
global
    log stdout format raw local0 info
    maxconn 20000

defaults
    log                 global
    timeout connect     10m
    timeout client      30m
    timeout server      30m
    option              clitcpka
    option              tcplog

listen cockroach-jdbc
    bind :26000    ---> vamos a hacer un bind de todas las instancias que están en
roach-0, roach-1, roach-2 del puerto 26257 al 26000
    mode tcp
    balance leastconn
    option httpchk GET /health?ready=1
    server roach-0 roach-0:26257 check port 8080   ---> corriendo en el 26257
(puerto por defecto de cockroachDB)
    server roach-1 roach-1:26257 check port 8080
    server roach-2 roach-2:26257 check port 8080

listen cockroach-ui
    bind :8080   ---> lo mismo con el 8080
    mode tcp
    balance leastconn
    option httpchk GET /health
    server roach-0 roach-0:8080 check port 8080
    server roach-1 roach-1:8080 check port 8080
    server roach-2 roach-2:8080 check port 8080

listen stats
    bind :8081  ---> podremos acceder al balanceador de carga a través del 8081
    mode http
    stats enable
    stats hide-version
    stats realm Haproxy\ Statistics
    stats uri /
```

- docker compose up no nos sirve porque tenemos que construir primero el contenedor de haproxy
- Uso -f para indicar que es el archivo de la raíz con -d build porquye tenemos dependencias con todos aquellos contenedores que necesiten ser construidos

> docker compose -f docker-compose.yml up -d --build

- En localhost:8081 puedo vere el balanceador de carga
- En localhost:8080 tengo la UI de cockroachDB
    - Tiene las DB defaultdb, postgres y system que trae por defecto

---

02 gRPC - Definir dominio en archivos .proto

- Tenemos el Dock, Bike y Ride
- En un dock pueden haber 0 o muchas bicis
- Una bici puede estar en 1 Dock (ranura de estacionamiento)
- Una bici hace un viaje
- En un viaje hay un dock de origen y un dock de destino
- Creo la carpeta proto en la raíz con Entities.proto
- Primero siempre defino la siuntaxis y el nombre del paquete

```
syntax = "proto3";

package DriveYourCity;

message Bike {
    int32   id = 1;
    int32   totalKm = 2;
    Dock    dock = 3;
}

message Dock {
    int32       id = 1;
    int32       maxBikes = 2;
    repeated    Bike bikes = 3; //arreglo repetido de bicis
}

message Ride {
    int32   id = 1;
    int32   km = 2;
    Bike    bike = 3;
    Dock    originDock = 4;
    Dock    targetDock = 5;
}
```

- Ahora definiremos los proto de los 3 microservicios
- Se puede construir de varias formas. En este caso vamos a dividir las interfaces en archivos diferentes, ya que es más claro
- Creo Dock.proto
- Como siempre, declaro la sintaxis, el paquete y en este caso importo lel .proto de entidades

```
syntax = "proto3";

package DriveYourCity;

import "Entities.proto";

service IDockService {
  rpc CreateDock (CreateDockRequest) returns (DockResponse);
  //es una response de stream pq no queremos enviar un objeto gigante con todos
```

```
los docks
  //enviaremos unbo tras otro
  rpc GetAllDocks (GetAllDocks) returns (stream DockResponse);
  rpc GetDockById (GetDockByIdRequest) returns (DockResponse);
  rpc IsDockAvailable(IsDockAvailableRequest) returns (IsDockAvailableResponse);
}

// Communication Entities - Requests
message CreateDockRequest {
  Dock dock = 1;
}
message GetAllDocks {

}
message GetDockByIdRequest {
  int32 dockId = 1;
}
message IsDockAvailableRequest {
  int32 dockId = 1;
}

// Communication Entities - Responses
message DockResponse {
  Dock dock = 1;
}
message IsDockAvailableResponse {
  bool isAvalable = 1;
}
```

- Seguimos con las bicicletas
- Debemos poder obtener una bici por el id, crear una bici y añadir o quitar una bici de un dock

```
syntax = "proto3";

package DriveYourCity;

import "Entities.proto";

service IBikeService {
    rpc GetBikeById (GetBikeByIdRequest) returns (BikeResponse);
    rpc CreateBike (BikeRequest) returns (BikeResponse);
    rpc AttachBikeToDock (AttachBikeToDockRequest) returns (BikeResponse);
    rpc UnAttachBikeFromDock (UnAttachBikeFromDockRequest) returns (BikeResponse);
}

// Communication Entities - Requests
message GetBikeByIdRequest {
  int32 bikeId = 1;
}
message BikeRequest {
  Bike bike = 1;
```

```
}

message AttachBikeToDockRequest {
  int32 bikeId = 1;
  int32 dockId = 2;
  int32 totalKms = 3; //queremos el número de km antes de ser añadida al nuevoi
dock
}

message UnAttachBikeFromDockRequest {
  int32 bikeId = 1;  //solo necesito el bikeId si ya está en un dock
}

// Communication Entities - Responses
message BikeResponse {
  Bike bike = 1;
}
```

- Vamos con ride
- Voy a tener que comprobar que esa bicileta esté disponible en un dock, que se pueda desacoplar del dock y actualizar la instancia de la bicileta y el dock en la base de datos para generar consistencia
- El objetivo de esta prueba de concepto es ir actualizando la información de kilometraje de la bicicleta

```
syntax = "proto3";

package DriveYourCity;

import "Entities.proto";

service IRideService {
    rpc StartRide (StartRideRequest) returns (RideResponse);
    rpc UpdateRide (stream UpdateRideRequest) returns (stream RideResponse);
//actualizaremos la distancia recorrida cada x por stream bidireccional
    //necesito manteener este canal TCP abierto para enviar y recibir de forma
continua la información
    rpc EndRide (EndRideRequest) returns (stream EndRideResponse); //endRide es un
stream porque enviaremos diferente información
}

// Communication Entities - Requests
message StartRideRequest {
    int32 bikeId = 1;    //con qué bicicleta voy as inicar el viaje
}

message UpdateRideRequest {
    int32 rideId    = 1;
    int32 newKms    = 2;
}

message EndRideRequest {
    int32 rideId = 1; //necesito el id del ride
```

```
    int32 dockId = 2; //necesito el dock de destino
}

// Communication Entities - Responses
message RideResponse {
    Ride ride = 1;
}

message EndRideResponse {
    string info = 1;
}
```

- Con esto tenemos definido el contexto de dominio del proyecto
- Tenemos las entidades y los servicios
- **BikeIoT** se va a conectar **solamente a los servicios de ride**
- Bike y dock se comunicarán entre ellos
- Ride se comunicará con dock también (para verificar si un dock está disponible antes de terminar un viaje)
- Vamos a tener todo en un mismo repositorio
- Como compartir las definiciones de .proto
    - Un repo para los archivos .proto:
        - Están todas las definiciones en un mismo lugar.
        - Muchas dependencias.
        - Si ese repo se rompe los demás van a tenr problemas
    - Separados en repositorios por cada servicio:
        - Un repo por cada subdominio.
        - Separamos mejor las responsabilidades.
        - Muchos pequeños repos
        - Va a tocar construir un pipeline para cada repo para generar el código fuente
    - Copiar y pegar
        - Muy válida y bastante usada
        - Sencilla de usar
        - Todos los repo van a tener una copia del modelo de dominio
        - No creo dependencias a dominios externos ni divido en pequeños subdominios
        - Vamos a tener muchas versiones que pueden desactualizarse
        - Puede volverse poco claro (qué controla, qué maneja, puesto que está todo copiado)
    - gRPC server reflection protocol
- Aquí lo manejaremos todo en la carpeta raíz

---

03 gRPC - Prisma y CockroachDB

- Definiremos la DB con prisma
- Creo la carpeta database en la raíz
- Creamos el proyecto con yarn init
- Instalamos prisma

```
yarn add prisma -dev
```

- Añadimos dos scripts en el packaghe.json
- Para verificar todos los cambios entre la DB y el archivo .prisma para generar la migración y mantener la DB actualizada uso db:migrate
    - uso dev para migrar en modo desarrollo y definimos el schema pasaándole la ubi

```
"scripts": {
    "db:migrate": "npx prisma migrate dev --schema ./prisma/schema.prisma",
    "db:push": "npx prisma db push --schema ./prisma/schema.prisma"
  }
```

- Creo la carpeta prisma y creo el archivo schema.prisma
- Prisma puede generar el schema a raíz de una DB y al revés, generar una DB a través de un schema

```
//defino la conexión
datasource db {
  provider = "cockroachdb" //el tipo de db
  url      = env("DATABASE_URL") //la variable de entorno donde está la URL DE LA
db
}

generator client {
  provider        = "prisma-client-js" //prisma generará una librería para acceder
a nuestro proyecto dado un schema
}                                       //es decir, usaremos de forma nativa
Prisma.loquesea para realizar als operaciones con la db

model Dock {
  id        Int      @id
  maxBikes  Int      @default(5)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  bikes Bike[]  //la referencia está en Bike
  originRides Ride[] @relation("originDock") //relaciones con ride
  targetRides Ride[] @relation("targetDock") //relaciones con ride
}

model Bike {
  id        Int       @id
  dockId    Int?
  totalKm   Int       @default(0)
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt
  dock      Dock?     @relation(fields: [dockId], references: [id]) //referencia
con el dock
  //rides     Ride[]    @relation("ride")
}

model Ride {
  id            Int       @id @default(sequence())
  km            Int
```

```
  bikeId        Int
  originDockId  Int
  targetDockId  Int?
  createdAt     DateTime  @default(now())
  updatedAt     DateTime  @updatedAt
  //bike          Bike      @relation("ride", fields: [bikeId], references: [id])
//referencia con Bike
  originDock    Dock      @relation("originDock", fields: [originDockId],
references: [id])
  targetDock    Dock?     @relation("targetDock", fields: [targetDockId],
references: [id])
}
```

- Creo la variable de entorno en .env (en la raíz)
- CockroachDB usa postgres por debajo
- Deshabilito el password

```
DATABASE_URL="postgresql://root@localhost:26000/driveyourcity?sslmode=disable"
```

- Uso el comando migrate

> yarn run db:migrate

- Para conectarme puedo usar TablePlus
  - host: localhost
  - user: root
  - Port: 26000
  - Database: driveyourcity
  - SSLMode: disabled
- Si voy al UI de cockroach en localhost:8080 puedo ver a driveyouircity y ya voy a poder ver la actividad SQL

---

04 gRPC - Dock Service

- Dock es el dispositivo al que las bicis llegan y de las que son tomadas para realizar los viajes
- Esate servicio se va a conectar al balanceador de carga que le va a dar acceso a la DB que está dividida en 3 instancias
- En Dock.proto creamos los 4 servicios que usaremos
  - Crear un dock
  - Obtener todos los docks
  - Obtener un dock por id
  - Saber si el dock está disponible
- Vamos a hacer el scaffolding del proyecto
- Es algo que vamos a tener que hacer con cada uno de los microservicios
- Creo el directorio de dock-service
- Uso yarn init

- Instalo las dependencias de node

> npm i @grpc/grpc-js @grpc/proto-loader @prisma/client npm i --save-dev @types/google-protobuf grpc_tools_node_protoc_ts prisma ts-node typescript

- Creamos los scriptsç

```
 "scripts": {
    "db:gen": "sh scripts/prisma-gen.sh", //generar la DB
    "proto:gen": "sh scripts/proto-gen.sh", //generar los archivos de ts con las definiciones declaradas en proto
    "start:server": "ts-node server.ts" //ejecutar el servidor
  }
```

- Creo la carpeta scripts
- Para crear el script para prisma creo prisma-gen.sh

```
# copio el schema
cp -f ./../database/prisma/schema.prisma ./../dock-service/schema.prisma
# genero el documento de node
npx prisma generate --schema ./schema.prisma
# remnuevo el documento para no generar basura
rm ./../dock-service/schema.prisma
```

- Creo proto-gen.sh

```
#!/bin/bash

PROTO_DIR=./../../proto

# usamos la librería proto-loader para a travgés de la librería grpc generar todos los archivos que contengan .proto
yarn proto-loader-gen-types --grpcLib=@grpc/grpc-js --outDir=src/proto/ ./../proto/*.proto
```

- *NOTA*: si los scripts no funcionan realizar el código manualmente
  - Para prisma copiar schema.prisma en la raíz del microservicio y ejecutar npx ...
- En node_modules/@prisma/client puedo ver todos los archivos ts y js generados
- Ahora ejecutamos **yarn run proto:gen**
- Esto genera todos los archivos con el código fuente dentro de la carpeta proto/DriveYourCity y los archivos Bike, Dock, Entities
- Creo .env con la URL de la DB (postgres, ya que cockroachDB usa postgres por debajo)

```
DATABASE_URL="postgresql://root@localhost:26000/driveyourcity?sslmode=disable"
```

- Crearemos src/persistence dónde almacenaremos toda la persistencia
- Creo src/services, donde crearemos la implementación de los servicios de grpc y otros servicios de apoyo que vamos a necesitar
- Creo la carpeta src/utils con código de utilidad
- Creo el server.ts en la raíz
- De momento lo hacemos inseguro

```
// @ts-ignore
import path from 'path'
import * as grpc from '@grpc/grpc-js'
import * as protoLoader from '@grpc/proto-loader'
import { ProtoGrpcType } from './src/proto/Dock';
import { DockService } from './src/service/DockService';

const PORT = 9081;
const DOCK_PROTO_FILE = './../proto/Dock.proto';

const dockPackageDef = protoLoader.loadSync(path.resolve(__dirname,
DOCK_PROTO_FILE));
const dockGrpcObj = (grpc.loadPackageDefinition(dockPackageDef) as unknown) as
ProtoGrpcType;

function main() {
    const server = getServer();
    const serverCredentials = grpc.ServerCredentials.createInsecure();

    server.bindAsync(`0.0.0.0:${PORT}`, serverCredentials,
        (err, port) => {
            if (err) {
                console.error(err)
                return
            }
            console.log(`dock server as started on port ${port}`)
            server.start()
        })
}

function getServer() {
    const server = new grpc.Server();

    server.addService(dockGrpcObj.DriveYourCity.IDockService.service, new
DockService())

    return server
}

main()
```

- Dentro de la carpeta utils creo el archivoi prisma.ts para la conexión
- Es un código genérico que se conecta a la instancia definidaq en la variable de entorno

```
import { PrismaClient } from '@prisma/client';

declare global {
  var prisma: PrismaClient | undefined;
}

export const prisma = global.prisma || new PrismaClient();

if (process.env.NODE_ENV !== 'production') {
  global.prisma = prisma;
}

async function connectDB() {
  try {
    await prisma.$connect();
    console.log('? Database connected successfully');
  } catch (error) {
    console.log(error);
    process.exit(1);
  } finally {
    await prisma.$disconnect();
  }
}

export default connectDB;
```

- En utils creo también gRPC.ts para definir errores personalizados
- Para ello importo Status que es lo que nos ofrece la librería de grpc
- Si clico encima con ctrl puedo ver el archivo de definiciones constants.d.ts con todos los status disponibles

```
export declare enum Status {
    OK = 0,
    CANCELLED = 1,
    UNKNOWN = 2,
    INVALID_ARGUMENT = 3,
    DEADLINE_EXCEEDED = 4,
    NOT_FOUND = 5,
    ALREADY_EXISTS = 6,
    PERMISSION_DENIED = 7,
    RESOURCE_EXHAUSTED = 8,
    FAILED_PRECONDITION = 9,
    ABORTED = 10,
    OUT_OF_RANGE = 11,
    UNIMPLEMENTED = 12,
    INTERNAL = 13,
    UNAVAILABLE = 14,
    DATA_LOSS = 15,
    UNAUTHENTICATED = 16
}
```

```
export declare enum LogVerbosity {
    DEBUG = 0,
    INFO = 1,
    ERROR = 2,
    NONE = 3
}
/**
 * NOTE: This enum is not currently used in any implemented API in this
 * library. It is included only for type parity with the other implementation.
 */
export declare enum Propagate {
    DEADLINE = 1,
    CENSUS_STATS_CONTEXT = 2,
    CENSUS_TRACING_CONTEXT = 4,
    CANCELLATION = 8,
    DEFAULTS = 65535
}
export declare const DEFAULT_MAX_SEND_MESSAGE_LENGTH = -1;
export declare const DEFAULT_MAX_RECEIVE_MESSAGE_LENGTH: number;
```

- Uso los que necesito en utils/gRPC.ts

```
import { Status } from "@grpc/grpc-js/build/src/constants";

export const NotFoundError = (entity: string, id: number) => ({ code:
Status.NOT_FOUND, message: `${entity} with id ${id} not found` });
//cuando nos invocan una función pero los argumentos no son insufcientes muestro
este error
export const InvalidArgumentError = (args: string[]) => ({ code:
Status.INVALID_ARGUMENT, message: `${args.join(', ')} missing arguments.` });
//error desconocido o no controlado
export const InternalError = (message: string) => ({ code: Status.INTERNAL,
message });
```

- Este es el scaffolding completo del proyecto
- Creo en src/persistence/DockPersistence.ts
- Creamos la interfaz de acceso a datos a la entidad
- Creamos la clase implementando la interfaz. Le digo al IDE que me cree los métodos rápidamente
- Para obtener el cliente de prisma creo un atributo privado _prisma
- Importo (la conexión) prisma de utils, la paso por el constructor
- DockCreateInput es un componente generado ese componente nativamente al doiminio por Prisma desde el esquema (importo Prisma de @prisma/client)
- Uso async await porque voy a interactuar con la DB, por lo que el método devuelve una promesa
- Para obtener un dock por id, uso where para indicar que el id que le mando por parámetro es el id que busco y tambiénn quiero obtener las bicicletas de ese dock, por lo que uso includes
- Para saber si un dock está disponible o no uso $queryRaw para ejecutar syntaxis de SQL con un valor de is_avaliable: false por defecto

- Selecciono la selección del número de bicis desde Bike y dock dónde el "Dock".id == "Bike"."dockId"
- Este número debe de ser inferior al número máximo de bicis que el dock puede recibir, donde "Dock".id es el id pasado como parámetro
- Si la respuesta devuelve algo (con .length), tomo el primer resultado que está disponible y retorno la variable bboleana is_avaliable
- Si no devolvemos faLso indicando que el dock no puede recibir más bicicletas

```
import { prisma } from "./../utils/prisma";
import { Prisma } from "@prisma/client";
import { Dock } from "../proto/DriveYourCity/Dock";

export interface IDockPersistence {
    createDock(dock: Dock): Promise<Dock>;
    getAllDocks(): Promise<Dock[]>;
    getDockById(id:number): Promise<Dock | undefined>;
    isDockAvailable(id: number): Promise<boolean>;
}

export class CockroachDBDockPersistence implements IDockPersistence {

    private _prisma;

    constructor() {
        this._prisma = prisma;
    }

    async createDock(dock: Dock): Promise<Dock> {
        const input = dock as Prisma.DockCreateInput;
        const newDock = await this._prisma.dock.create({ data: input });
        return newDock;
    }
    async getAllDocks(): Promise<Dock[]> {
        return this._prisma.dock.findMany();
    }
    async getDockById(id: number): Promise<Dock | undefined> {
        const dock = await this._prisma.dock.findFirst({
            where: {
              id,
            },
            include: {
                bikes: true,
            }
          });
        return dock as Dock;
    }
    async isDockAvailable(id: number): Promise<boolean> {
        const result = await prisma.$queryRaw<{is_available: false}[]>`
            SELECT (SELECT COUNT(*) as numBikes
                    FROM "Bike",
                        "Dock"
                    WHERE "Dock".id = "Bike"."dockId"
```

```
                    AND "Dock".id = ${id}) < "maxBikes" as is_available
            FROM "Dock"
            WHERE "Dock".id = ${id};
        `;
        if(result.length) {
            return result[0].is_available;
        }
        return false;
    }
}
```

- Para trabajar el servicio voy a src/service/DockService.ts
- Creo una nueva instancia de CockRoachDBDockPersistence
- La clase DockService implementa la interfaz IDockServiceHandlers
    - Esta interfaz está ubicada en dock-service/src/proto/DriveYourCity/IDockService con todos los métodos declarados en .proto a implementar
- Para **crear el dock** obtengo el body con call.request, extraigo el dock
- Usaré try catch
- Hago la validación: si hay dock usop el método create de dockPersistence (CockRoachDBPersistence)
- Devuelvo en el callback el primer parámetro (error) en null y de segundo el dock
- Si hay un error lo capturo con el catch y uso en el primer argumento del callback uno de mis errores custom y como segundo parámetro le paso el dock como undefined
- Para **obtener todos los docks** uso el método de dockPersistence
- Con un forEach uso call.write para abrir la conexión por streaming y pasarle uno a uno todos los docks
- Cierro la conexión con call.end
- Para **obtener el dock por id** extraigo el dockId del body de la request con call.request.dockId
- Hago la validación y la inserción
- Si la inserción no va bien, Capturo el error con un ternario y lo guardo en una variable y lo devuelvo en el callback junto al dock
- Si no hay error su valor será null
- Si no hay dockId es que los argumentos paqsados son erróneos, por lo que lanzo mi error custom correswpondiente
- En el catch capto un posible error no manejado
- Para saber si **el dock está disponible** extraigo el id de la request con call.request.dockId
- Hago la validación, si hay dockId uso la DB para obtener la respuesta. Devuelvo en el callback el error como null, y la respuesta de la consulta
- Si no hay dockId es que no se pasó cómo argumento, lanzomi error custom
- En el catch capturo un posible error no manejado
- Exporto DockService!

```
import { ServerUnaryCall, ServerWritableStream, sendUnaryData } from "@grpc/grpc-
js";
import { IDockServiceHandlers } from "../proto/DriveYourCity/IDockService";
import { DockResponse } from "../proto/DriveYourCity/DockResponse";
import { GetAllDocks__Output } from "../proto/DriveYourCity/GetAllDocks";
import { GetDockByIdRequest__Output } from
"../proto/DriveYourCity/GetDockByIdRequest";;
```

```
import { CreateDockRequest__Output } from
"../proto/DriveYourCity/CreateDockRequest";
import { IsDockAvailableRequest__Output } from
"../proto/DriveYourCity/IsDockAvailableRequest";
import { IsDockAvailableResponse } from
"../proto/DriveYourCity/IsDockAvailableResponse";
import { CockroachDBDockPersistence } from "../persitence/DockPersistence";
import { InternalError, InvalidArgumentError, NotFoundError } from
"../utils/gRPC";

const dockPersistence = new CockroachDBDockPersistence();

class DockService implements IDockServiceHandlers {
    [name: string]: import("@grpc/grpc-js").UntypedHandleCall;

    async CreateDock (call: ServerUnaryCall<CreateDockRequest__Output,
DockResponse>, callback: sendUnaryData<DockResponse>): Promise<void> {
        try {
            const dock = call.request.dock;
            console.log('CreateDock', { dock });
            if (dock) {
                const newDock = await dockPersistence.createDock(dock);
                callback(null, { dock: newDock });
            }
        } catch (err) {
            callback(InternalError(err as string), { dock: undefined });
        }
    }

    async GetAllDocks (call: ServerWritableStream<GetAllDocks__Output,
DockResponse>): Promise<void> {
        console.log('GetAllDocks');
        const docks = await dockPersistence.getAllDocks();
        docks.forEach(dock => call.write({ dock }));
        call.end();
    }

    async GetDockById (call: ServerUnaryCall<GetDockByIdRequest__Output,
DockResponse>, callback: sendUnaryData<DockResponse>): Promise<void> {
        try {
            const dockId = call.request.dockId;
            console.log('GetDockById', { dockId });
            if (dockId) {
                const dock = await dockPersistence.getDockById(dockId);
                const error = dock ? null : NotFoundError('dock', dockId);
                callback(error, { dock });
            }
            callback(InvalidArgumentError(['dockId']), { dock: undefined });
        } catch (err) {
            callback(InternalError(err as string), { dock: undefined });
        }
    }

    async IsDockAvailable(call: ServerUnaryCall<IsDockAvailableRequest__Output,
```

```
IsDockAvailableResponse>, callback: sendUnaryData<IsDockAvailableResponse>):
Promise<void> {
        try {
            const dockId = call.request.dockId;
            console.log('IsDockAvailable', { dockId });
            if (dockId) {
                callback(null, { isAvalable: await
dockPersistence.isDockAvailable(dockId) });
            }
            callback(InvalidArgumentError(['dockId']), { isAvalable: false });
        } catch (err) {
            callback(InternalError(err as string), { isAvalable: false });
        }
    }
}

export {
    DockService
}
```

- Ahora solo queda hacer las pruebas con POSTMAN (ya creamos el servidor anteriormente)
- Para probar el microservicio incializamos docker-compose en la raíz de DriveYourCity

```
docker compose -f docker-compose.yml up -d --build
```

- Para sincronizar el schema con la instancia de la DB usaré el script db:push

```
 "db:push": "npx prisma db push --schema ./prisma/schema.prisma"
```

---

05 gRPC Bike Service

- Bike se encarga no solo del CRUD de una bici, también de asociar o desasociar unabici a un dock
- Observemos Bike.proto

```
syntax = "proto3";

package DriveYourCity;

import "Entities.proto";

service IBikeService {
    rpc GetBikeById (GetBikeByIdRequest) returns (BikeResponse);
    rpc CreateBike (BikeRequest) returns (BikeResponse);
    rpc AttachBikeToDock (AttachBikeToDockRequest) returns (BikeResponse);
    rpc UnAttachBikeFromDock (UnAttachBikeFromDockRequest) returns (BikeResponse);
}

// Communication Entities - Requests
```

```
message GetBikeByIdRequest {
  int32 bikeId = 1;
}
message BikeRequest {
  Bike bike = 1;
}

message AttachBikeToDockRequest {
  int32 bikeId = 1;
  int32 dockId = 2;
  int32 totalKms = 3;
}

message UnAttachBikeFromDockRequest {
  int32 bikeId = 1;
}

// Communication Entities - Responses
message BikeResponse {
  Bike bike = 1;
}
```

- BikeService no solo expone un servicio de gRPC a RideService, también consume DockService
- Vamos a crear el microservicio, también vamos a crear un cliente que se conecte al servicio anterior (DockService) para realizar algunas tareas
- Creamos la carpeta BikeService, hacemos todo el scaffolding hecho anteriormente en DockService
- Es recomendable tener un repositorio con todo el scaffolding para poder reproducir los microservicios sin tener que repetir una y otra vez el código
- Genero el código fuente y el cliente de prisma con prisma-gen.sh y proto-gen.sh
- Creo la carpeta src/persistence/BikePersistence
- Realizamos la misma operación, importamos prisma de /utils/prisma con la conexión y se la pasamos al constructor guardándola en un atributo privado
- Para **crear una bici** necesitamos crear un input. Le paso el id que es lo que tengo
- Uso this._prisma.bike.create y en un objeto le paso en la data el input
- Para **retornar todas las bicis** uso findMany
- Para **obtener la bici por id** uso findFirst, colocando en el where el id e incluyendo el dock en include
- Retorno bike como Bike
- Para **actualizar la bici** guardo la bike como Prisma.BikeUpdateInput
- Si clico encima de BikeUpdateInput + ctrl obtengo esto
- index.d.ts

```
export type BikeUpdateInput = {
    id?: IntFieldUpdateOperationsInput | number
    totalKm?: IntFieldUpdateOperationsInput | number
    createdAt?: DateTimeFieldUpdateOperationsInput | Date | string
    updatedAt?: DateTimeFieldUpdateOperationsInput | Date | string
    dock?: DockUpdateOneWithoutBikesNestedInput //puede tener un dock
    rides?: RideUpdateManyWithoutBikeNestedInput //puede tener un ride
  }
```

- Si clico encima de DockUpdateOneWithoutBikesNestedInput + ctrl obtengo esto
- Son todas las operaciones que puedo hacer con una relación

```
 export type DockUpdateOneWithoutBikesNestedInput = {
    create?: XOR<DockCreateWithoutBikesInput,
DockUncheckedCreateWithoutBikesInput>
    connectOrCreate?: DockCreateOrConnectWithoutBikesInput
    upsert?: DockUpsertWithoutBikesInput
    disconnect?: DockWhereInput | boolean
    delete?: DockWhereInput | boolean
    connect?: DockWhereUniqueInput
    update?: XOR<XOR<DockUpdateToOneWithWhereWithoutBikesInput,
DockUpdateWithoutBikesInput>, DockUncheckedUpdateWithoutBikesInput>
  }
```

- En BikePersistence.ts, en update, si bike.dock existe quiero conectar mi bici a ese dock
- Si no existe, lo que quiero es desconectar mi bici del dock al que esté conectado
- Para el update uso prisma.bike.update, pasándole el id a where, y pasándole el objeto data

```
import { Prisma } from '@prisma/client';
import { prisma } from '../utils/prisma';
import { Bike } from "../proto/DriveYourCity/Bike";

export interface IBikePersistence {
    createBike(bike: Bike): Promise<Bike>;
    getAllBikes(): Promise<Bike[]>
    getBikeById(id:number): Promise<Bike | undefined>
    updateBike(id: number, bike: Partial<Bike>): Promise<Bike | undefined>
}

export class CockroachDBBikePersistence implements IBikePersistence {

    private _prisma;

    constructor() {
        this._prisma = prisma;
    }

    async createBike(bike: Bike): Promise<Bike> {
        const input: Prisma.BikeCreateInput = {
            id: bike.id!,
        }
        const newBike = await this._prisma.bike.create({ data: input });
        return newBike;
    }

    async getAllBikes(): Promise<Bike[]> {
        return this._prisma.bike.findMany();
```

```
    }

    async getBikeById(id: number): Promise<Bike | undefined> {
        const bike = await this._prisma.bike.findFirst({
            where: {
              id,
            },
            include: {
                dock: true,
            },
          });
        return bike as Bike;
    }

    async updateBike(id: number, bike: Partial<Bike>): Promise<Bike | undefined> {
        const data: Prisma.BikeUpdateInput = bike as Prisma.BikeUpdateInput;
        if(bike.dock) {
            data.dock = { connect: { id: bike.dock?.id }};
        } else {
            data.dock = { disconnect: true};
        }
        const updatedBike = await prisma.bike.update({
            where: { id },
            data
        });
        return updatedBike;
    }
}
```

- Ya tenemos la persistencia, podemos crear nuestro servicio
- Pero antes creemos el cliente que se conecta al dock
- En service creo DockClient.ts

```
import path from 'path'
import * as grpc from '@grpc/grpc-js'
import * as protoLoader from '@grpc/proto-loader'
import { ProtoGrpcType } from '../proto/Dock';

const PORT = 9081; //puerto del server de Dock
const DOCK_PROTO_FILE = './../../../proto/Dock.proto';

const dockPackageDef = protoLoader.loadSync(path.resolve(__dirname,
DOCK_PROTO_FILE)); //creo la definición del paquete pasándole la ruta del proto
const dockGrpcObj = (grpc.loadPackageDefinition(dockPackageDef) as unknown) as
ProtoGrpcType; //creo el objeto para crear la instancia del servicio

const channelCredentials = grpc.credentials.createInsecure(); //de momento
inseguro
const dockServiceClient = new
dockGrpcObj.DriveYourCity.IDockService(`0.0.0.0:${PORT}`, channelCredentials)
//creo el servicio y lo mapeo al puerto
```

```
const dockClient = {  //solo necesito interactuar con el Dock para saber si está
disponible
    isDockAvailable: async (dockId: number): Promise<boolean> => {
        return new Promise((resolve, reject) => {
            dockServiceClient.IsDockAvailable({dockId}, (err, response) => { //el
servicio recibe el id (que recoge con el call.request)
                                                                              //y
el callback con el error de primer parámetro y la response de segundo
                if(response) {
                    resolve(response.isAvalable as boolean); // si hay response
resolvemos la promesa devolviendo el isAvaliable como boolean
                }
                reject(err); // si no hay respuesta usamos reject y devolvemnos el
error
            });
        })
    }
}

export {
    dockClient
}
```

- Encapsula y divide muy bien las responsabilidades
- Usaré este cliente para comunicarme con dock-microservice desde bike-microservice
- Entonces lo que he hecho es **crear un cliente en bike-microservice que se conecte al puerto de dock-microservice con una nueva instancia de DockService, e implementar el código con el cliente para interactuar con el servicio**
- Vayamos con el BikeService. Dónde está?
    - En src/proto/DriveYourCity/IBikeService
    - Contiene todas las definiciones. la que nos interesa es **IBikeServiceHandlers**
- Quiero explicar este código:

```
[name: string]: import("@grpc/grpc-js").UntypedHandleCall;
```

- **[name: string]**:
    - Esto es una sintaxis de TypeScript para definir un índice de propiedad en una interfaz. Significa que cualquier propiedad del objeto puede tener un nombre de tipo string.
- **import("@grpc/grpc-js").UntypedHandleCall**:
    - Esto se refiere a un tipo que está siendo importado desde el módulo @grpc/grpc-js. En este caso, UntypedHandleCall es un tipo exportado por este módulo.
- **import("@grpc/grpc-js")** es la forma de importar tipos o valores desde un módulo externo en TypeScript sin necesidad de hacer una importación explícita en la parte superior del archivo.
- Creo una instancia de new CockroachDBBikePersistence()
- Para el attach de la bici al dock abro un try catch (como en los casos anteriores)
- extraigo las propiedades del body con call.request.propiedad

- Si tengo bikeId y dockId uso el dockClient para ver si el dock esta disponible y uso la instancia de la db de bike (bikePersistence) para obtener la bici por id
- Si no hay dock mando un error con el callback
- Si el dock de la bici es distinto de null es que ya está ligada a un dock
- Y si no usamos updatedBike, le paso el id, actualizo el total de km de la bici y le indico el id del dock
- Si no es ninguno de estos casos lanzo un custom error de invalid arguments y en el catch recojo el error no manejado (de haberlo)

```
import { ServerUnaryCall, handleUnaryCall, sendUnaryData } from "@grpc/grpc-js";
import { Status } from "@grpc/grpc-js/build/src/constants";
import { IBikeServiceHandlers } from "../proto/DriveYourCity/IBikeService";
import { BikeRequest__Output } from "../proto/DriveYourCity/BikeRequest";
import { BikeResponse } from "../proto/DriveYourCity/BikeResponse";
import { GetBikeByIdRequest__Output } from
"../proto/DriveYourCity/GetBikeByIdRequest";
import { AttachBikeToDockRequest__Output } from
"../proto/DriveYourCity/AttachBikeToDockRequest";
import { CockroachDBBikePersistence } from "../persitence/BikePersistence";
import { dockClient } from "./DockClient";
import { InternalError, InvalidArgumentError, NotFoundError } from
"../utils/gRPC";
import { UnAttachBikeFromDockRequest__Output } from
"../proto/DriveYourCity/UnAttachBikeFromDockRequest";

const bikePersistence = new CockroachDBBikePersistence();
class BikeService implements IBikeServiceHandlers {

    [name: string]: import("@grpc/grpc-js").UntypedHandleCall; //explicación de
este código arriba!

    async AttachBikeToDock(call: ServerUnaryCall<AttachBikeToDockRequest__Output,
BikeResponse>, callback: sendUnaryData<BikeResponse>): Promise<void> {
        try {
            const bikeId = call.request.bikeId;
            const dockId = call.request.dockId;
            const totalKm = call.request.totalKms ? call.request.totalKms : 0;

            console.log('AttachBikeToDock', { bikeId, dockId, totalKm });
            if(bikeId && dockId) {
                const isDockAvailable = await dockClient.isDockAvailable(dockId);
                const bike = await bikePersistence.getBikeById(bikeId);
                if(!isDockAvailable) {
                    callback({ code: Status.FAILED_PRECONDITION, message: `dock
with id ${dockId} not available` }, { bike: undefined });
                } else if(bike?.dock !== null) {
                    callback({ code: Status.FAILED_PRECONDITION, message: `bike
with id ${bikeId} is attached to the dock ${bike?.dock?.id}` }, { bike: undefined
});
                }else {
                    const updatedBike = await bikePersistence.updateBike(bikeId, {
totalKm: bike?.totalKm! + totalKm!, dock: { id: dockId } });
                    callback(null, { bike: updatedBike });
```

```
                }
            }
            callback(InvalidArgumentError(['dockId', 'bikeId']), { bike: undefined
});
        } catch (err) {
            callback(InternalError(err as string), { bike: undefined });
        }
    }

    async UnAttachBikeFromDock(call:
ServerUnaryCall<UnAttachBikeFromDockRequest__Output, BikeResponse>, callback:
sendUnaryData<BikeResponse>): Promise<void> {
        try {
            const bikeId = call.request.bikeId;

            console.log('UnAttachBikeFromDock', { bikeId });
            if(bikeId) {
                const bike = await bikePersistence.getBikeById(bikeId);
                if(bike?.dock !== null) {
                    const updatedBike = await bikePersistence.updateBike(bikeId, {
dock: null });
                    callback(null, { bike: updatedBike });
                } else {
                    callback({ code: Status.FAILED_PRECONDITION, message: `bike
with id ${bikeId} is not attached to any dock` }, { bike: undefined });
                }
            }

        } catch (err) {
            callback(InternalError(err as string), { bike: undefined });
        }
    }

    async CreateBike(call: ServerUnaryCall<BikeRequest__Output, BikeResponse>,
callback: sendUnaryData<BikeResponse>): Promise<void> {
        try {
            const bike = call.request.bike;
            console.log('CreateBike', { bike });
            if(bike) {
                const newBike = await bikePersistence.createBike(bike);
                callback(null, { bike: newBike });
            }
        } catch (err) {
            callback(InternalError(err as string), { bike: undefined });
        }
    }

    async GetBikeById(call: ServerUnaryCall<GetBikeByIdRequest__Output,
BikeResponse>, callback: sendUnaryData<BikeResponse>): Promise<void> {
        try {
            const bikeId = call.request.bikeId; //extraigo el id de la request
            console.log('GetBikeById', { bikeId });
            if (bikeId) {
                const bike = await bikePersistence.getBikeById(bikeId); //uso el
```

```
método
                const error = bike ? null : NotFoundError('bike', bikeId); //si
hay bike el error es null, si no mando el custom error
                callback(error, { bike }); //retorno con el callback el error y la
bike
            }
        callback(InvalidArgumentError(['dockId']), { bike: undefined }); //si no
es ninguno de esos casos es que el argumento no es válido
        } catch (err) {
            callback(InternalError(err as string), { bike: undefined });// si hay
un error lo capturo con el catch
        }
    }
}

export {
    BikeService
}
```

- Creo el server.ts

```
// @ts-ignore
import path from 'path'
import * as grpc from '@grpc/grpc-js'
import * as protoLoader from '@grpc/proto-loader'
import { ProtoGrpcType } from './src/proto/Bike';
import { BikeService } from './src/service/BikeService';

const PORT = 9082;
const BIKE_PROTO_FILE = './../proto/Bike.proto';

const bikePackageDef = protoLoader.loadSync(path.resolve(__dirname,
BIKE_PROTO_FILE));
const bikeGrpcObj = (grpc.loadPackageDefinition(bikePackageDef) as unknown) as
ProtoGrpcType;

function main() {
    const server = getServer();
    const serverCredentials = grpc.ServerCredentials.createInsecure();

    server.bindAsync(`0.0.0.0:${PORT}`, serverCredentials,
        (err, port) => {
            if (err) {
                console.error(err)
                return
            }
            console.log(`ride server as started on port ${port}`)
            server.start()
        })
}

function getServer() {
```

```
    const server = new grpc.Server();

    server.addService(bikeGrpcObj.DriveYourCity.IBikeService.service, new
BikeService())

    return server
}

main()
```

- Creo el .env

```
DATABASE_URL="postgresql://root@localhost:26000/driveyourcity?sslmode=disable"
```

- Debemos iniciar los dos servers, el de dock y el de bike
- En POSTMAN uso createBike y le paso un objeto con id:1, dock: {}
- Para añadir la bici al dock mando en el objeto el dockId, el bikeId, y el totalkm
- Si intento asociar la misma bici me salta error de que la bici con id tal ya está asociada al dock con id tal

---

07 gRPC - RideService (última sección)

- Nos quedan tres casos de uso
  - Comenzar el viaje
  - A medida que se desarrolla el viaje enviar información del viaje
  - Terminar el viaje
- RideService se conecta al Dockservice y al BikeService
- A futuro podemos pensar en un componente IoT que esté instalado en las biciletas que transmita y se comunique con estos servicios, ofreciendo o consumiendo los servicios gRPC
- Creado todo el scafolding, los scripts, el cliente de prisma y los archivos de codigo fuente basado en los .proto, creo las carpetas persistence, service, etc
- Creo src/persistence/RidePeristence.ts
- Importo Prisma de @prisma/client y la conexión que cree con la DB de /utils/prisma
- Para **createRide** creo el input seteando el km a 0, conectando el id de la bike y el id del dock de origen
- Retorno el .create pasándole en la data el input
- Para el **getRideById** uso .findFirst.
- En la cláusula where le paso el id y en include quiero bike, originDock y targetDock en true para obtenerlos
- Devuelvo el ride
- Para el **updateRide** creo la data con el ride y lo casteo a Prisma.RideUpdateInput
- Si hay un targetDock conecto el id, si no pongo el disconnect en true
- Utilizo _prisma.ride.update donde el where tiene el id, le paso la data y en include tengo el bike, originDock y targetDock en true
- Retorno el updateRide

```
import { Prisma } from '@prisma/client';
import { prisma } from '../utils/prisma';
import { Ride } from "../proto/DriveYourCity/Ride";

export interface IRidePersistence {
    createRide(ride: Ride): Promise<Ride>;
    getAllRides(): Promise<Ride[]>
    getRideById(id:number): Promise<Ride | undefined>
    updateRide(id: number, ride: Ride): Promise<Ride | undefined>
}

export class CockroachDBRidePersistence implements IRidePersistence {

    private _prisma;

    constructor() {
        this._prisma = prisma; //de utils/prisma
    }

    async createRide(ride: Ride): Promise<Ride> {

        const input: Prisma.RideCreateInput = {
            km: 0,
            bike: {
                connect: { id: ride.bike?.id! }
            },
            originDock: {
                connect: { id: ride.originDock!.id! } //conecto el dock de origen
con el id del nuevo viaje
            }
        };
        return this._prisma.ride.create({ data: input });

    }
    getAllRides(): Promise<Ride[]> {
        return this._prisma.ride.findMany()
    }

    async getRideById(id: number): Promise<Ride | undefined> {
        const ride = await this._prisma.ride.findFirst({
            where: {
                id,
            },
            include: {
                bike: true,
                originDock: true,
                targetDock: true,
            }
        });
        return ride as Ride;
    }

    async updateRide(id: number, ride: Ride): Promise<Ride | undefined> {
```

```
        const data: Prisma.RideUpdateInput = ride as Prisma.RideUpdateInput;
//creo el objeto de datos a actualizar
        //para actualizar el targetDock, en caso de que no sea nulo lo conecto a
la llave foránea que me están enviando como dock de destino
        //en el caso de que no exista anulo cualquier tipo de conexión
        data.targetDock = ride.targetDock ? { connect: { id: ride.targetDock.id }}
: { disconnect: true };

        let updatedRide = await this._prisma.ride.update({
            where: { id },
            data,
            include: {
                bike: true,
                originDock: true,
                targetDock: true,
            }
        });


        return updatedRide;
    }

}
```

- La conexión a la DB de /utiuls/prisma es tal que así

```
import { PrismaClient } from '@prisma/client';

declare global {
  var prisma: PrismaClient | undefined;
}

export const prisma = global.prisma || new PrismaClient();

if (process.env.NODE_ENV !== 'production') {
  global.prisma = prisma;
}

async function connectDB() {
  try {
    await prisma.$connect();
    console.log('? Database connected successfully');
  } catch (error) {
    console.log(error);
    process.exit(1);
  } finally {
    await prisma.$disconnect();
  }
}

export default connectDB;
```

- También en /utils tengo el mismo archivo de errores custom que creé en los otros microservicios importando Status de grpc-js
- /utils/gRPC.ts

```
import { Status } from "@grpc/grpc-js/build/src/constants";

export const NotFoundError = (entity: string, id: number) => ({ code:
Status.NOT_FOUND, message: `${entity} with id ${id} not found` });
export const InvalidArgumentError = (args: string[]) => ({ code:
Status.INVALID_ARGUMENT, message: `${args.join(', ')} missing arguments.` });
export const InternalError = (message: string) => ({ code: Status.INTERNAL,
message });
```

- Necesito construir los clientes para comunciar ride con bikes y docks
- Creo en src/services/DockClient.ts (reutilizo el archivo escrito anteriormente) y BikeClient.ts
- DockClient.ts

```
import path from 'path'
import * as grpc from '@grpc/grpc-js'
import * as protoLoader from '@grpc/proto-loader'
import { ProtoGrpcType } from '../proto/Dock';

const PORT = 9081;
const DOCK_PROTO_FILE = './../../../proto/Dock.proto';

const dockPackageDef = protoLoader.loadSync(path.resolve(__dirname,
DOCK_PROTO_FILE));
const dockGrpcObj = (grpc.loadPackageDefinition(dockPackageDef) as unknown) as
ProtoGrpcType;

const channelCredentials = grpc.credentials.createInsecure();
const dockServiceClient = new
dockGrpcObj.DriveYourCity.IDockService(`0.0.0.0:${PORT}`, channelCredentials)

const dockClient = {
    isDockAvailable: async (dockId: number): Promise<boolean> => {
        return new Promise((resolve, reject) => {
            dockServiceClient.IsDockAvailable({dockId}, (err, response) => {
                if(response) {
                    resolve(response.isAvalable as boolean);
                }
                reject(err);
            });
        })
    }
}

export {
```

```
    dockClient
}
```

- Del BikeClient necesito el getBikeById, el attach Bike y unattachBike
- Creo el objeto de BikeClient y construyo los métodos
- Para resolver esta tarea asíncrona uso una promesa empleando resolve, reject
- Llamo al servicio y le paso el bikeId, en el callback tengo el error y la response
- Si hay response uso el resolve y devuelvo el response.bike como Bike
- Si no uso reject y devuelvo el error
- El BikeClient.ts

```
import path from 'path'
import * as grpc from '@grpc/grpc-js'
import * as protoLoader from '@grpc/proto-loader'
import { ProtoGrpcType } from '../proto/Bike';
import { Bike } from '../proto/DriveYourCity/Bike';

const PORT = 9082;
const BIKE_PROTO_FILE = './../../../proto/Bike.proto';

const bikePackageDef = protoLoader.loadSync(path.resolve(__dirname,
BIKE_PROTO_FILE));
const bikeGrpcObj = (grpc.loadPackageDefinition(bikePackageDef) as unknown) as
ProtoGrpcType;

const channelCredentials = grpc.credentials.createInsecure();
const bikeServiceClient = new
bikeGrpcObj.DriveYourCity.IBikeService(`0.0.0.0:${PORT}`, channelCredentials)

const bikeClient = {
    getBikeById: async (bikeId: number): Promise<Bike> => {
        return new Promise((resolve, reject) => {
            bikeServiceClient.GetBikeById({bikeId}, (err, response) => {
                if(response) {
                    resolve(response.bike as Bike);
                }
                reject(err);
            });
        })
    },

    unAttachBikeFromDock: async (bikeId: number): Promise<Bike> => {
        return new Promise((resolve, reject) => {
            bikeServiceClient.UnAttachBikeFromDock({bikeId}, (err, response) => {
                if(response) {
                    resolve(response.bike as Bike);
                }
                reject(err);
            });
        })
    },
```

```
        //si compruebo en el archivo .proto que necesito para el attach es el
bikeId, el dockId y el totalKms
    attachBikeToDock: async (bikeId: number, dockId: number, totalKms: number):
Promise<Bike> => {
        return new Promise((resolve, reject) => {
            bikeServiceClient.AttachBikeToDock({bikeId, dockId, totalKms}, (err,
response) => {
                if(response) {
                    resolve(response.bike as Bike);
                }
                reject(err);
            });
        })
    }
}

export {
    bikeClient
}
```

- RideService.ts
- RideService imlementa RideServiceHandlers desde IRideService
- Al ser métodos async devuelven una promesa
- Creo la instancia de CockroachDBRidePersistence

```
import { ServerDuplexStream, ServerUnaryCall, ServerWritableStream, sendUnaryData
} from "@grpc/grpc-js";
import { IRideServiceHandlers } from "../proto/DriveYourCity/IRideService";
import { CockroachDBRidePersistence } from "../persitence/RidePersistence";
import { EndRideRequest__Output } from "../proto/DriveYourCity/EndRideRequest";
import { EndRideResponse } from "../proto/DriveYourCity/EndRideResponse";
import { RideResponse } from "../proto/DriveYourCity/RideResponse";
import { StartRideRequest__Output } from
"../proto/DriveYourCity/StartRideRequest";
import { UpdateRideRequest, UpdateRideRequest__Output } from
"../proto/DriveYourCity/UpdateRideRequest";
import { bikeClient } from "./BikeClient";
import { Ride } from "../proto/DriveYourCity/Ride";
import { dockClient } from "./DockClient";
import { InternalError, InvalidArgumentError, NotFoundError } from
"../utils/gRPC";

const ridePersistence = new CockroachDBRidePersistence();

class RideService implements IRideServiceHandlers {
    [name: string]: import("@grpc/grpc-js").UntypedHandleCall;

    async StartRide(call: ServerUnaryCall<StartRideRequest__Output, RideResponse>,
callback: sendUnaryData<RideResponse>): Promise<void> {
        try {
```

```
            const bikeId = call.request.bikeId;
            console.log('StartRide', { bikeId });
            if(bikeId) {
                let bike = await bikeClient.getBikeById(bikeId);  //si tengo el
bikeId obtengo la bike
                const ride: Ride = {          //creo el objeto de ride
                    bike,
                    originDock: bike.dock
                }
                await bikeClient.unAttachBikeFromDock(bikeId); //con el cliente de
bike desviculo la bici del dock
                const newRide = await ridePersistence.createRide(ride); // creo el
ride en la DB
                callback(null, { ride: newRide }); //devuelvo en el callback el
error en null y el newRide
            }
            callback(InvalidArgumentError(['bikeId']), { ride: undefined }); //si
no hay bikeId devuelvo el custom error
        } catch (err) {
            callback(InternalError(err as string), { ride: undefined }); //en el
catch capturo el error no manejado
        }
    }

    async UpdateRide(call: ServerDuplexStream<UpdateRideRequest__Output,
RideResponse>): Promise<void> {
        //al ser un streaming dispongo del método .on donde tengo la data y end
        call.on('data', async (request: UpdateRideRequest) => {//el callback de la
request de tipo UpdateRideRequest es async pq consultaré la DB
            const newKms = request.newKms!; //extraigo la data de la request
            const rideId = request.rideId!;

            console.log('UpdateRide', { rideId, newKms });
            const ride = await ridePersistence.getRideById(rideId); //obtengo el
ride por el id
            if (ride) {                                    //si tengo el ride
actualizo pasándole el id del ride y la suma de los nuevos kms
                const updatedRide = await ridePersistence.updateRide(rideId, { km:
ride.km! + newKms });
                call.write({ride: updatedRide}); //uso .write para escribir por el
streaming
            } else {
                call.end(); //si no hay ride cierro la conexión
            }
        });

        call.on('end', () => {
            call.end(); //me aseguro de cerrar la conexión cuando no hay más data
        });
    }

    async EndRide(call: ServerWritableStream<EndRideRequest__Output,
EndRideResponse>): Promise<void> {
```

```
        try {
            const rideId = call.request.rideId; //extraigo la data de la request
            const targetDockId = call.request.dockId;
            console.log('EndRide', { rideId, targetDockId });
            if(rideId && targetDockId) {
                const ride = await ridePersistence.getRideById(rideId); //obtengo
el ride por el id
                if(ride && ride.originDock && ride.targetDock === null) { // si
tengo el ride y el originDock, pero el targetDock es null es que el viaje puede
terminar busco un dock disponible
                    const isDockAvailable = await
dockClient.isDockAvailable(targetDockId);
                    if(isDockAvailable) {
                        const updatedRide = await
ridePersistence.updateRide(rideId, { targetDock: { id: targetDockId } }); // si
hay un dock disponible actualizo el id del targetDock y le asigno el targetDock al
bike
                        const updatedBike = await
bikeClient.attachBikeToDock(ride.bike?.id!, updatedRide?.targetDock?.id!,
ride.km!);

                        //creo el objeto que quiero transmitir por streaming
                        const informData = [
                            `ride with id: ${updatedRide?.id} finished`,
                            `origin dock = ${updatedRide?.originDock?.id}`,
                            `target dock = ${updatedRide?.targetDock?.id}`,
                            `Total Kms = ${updatedRide?.km}`,
                            `bike with id ${updatedBike.id} and new total Kms
${updatedBike?.totalKm}`,
                        ]

                        informData.forEach(info => call.write({info})); // al ser
streaming escribo la data con .write
                        call.end(); //cierro la conexión
                    }
                    //en streaming no dispongo del callback, dispongo de .emit
para emitir un error
                    call.emit('error', new Error(`dock with id
${ride.originDock.id} is not available for handle more bikes.`));

                } else {
                    call.emit('error', new Error(`ride with id ${rideId} is not
available to end.`));
                }
            }
        } catch (err) {
            call.emit('error', err); //capturo un posible error no manejado
        }
    }
}

export {
    RideService
}
```

- Creo el server.ts en la raiz de ride-service

```
// @ts-ignore
import path from 'path'
import * as grpc from '@grpc/grpc-js'
import * as protoLoader from '@grpc/proto-loader'
import { ProtoGrpcType } from './src/proto/Ride';
import { RideService } from './src/service/RideService';

const PORT = 9083;
const RIDE_PROTO_FILE = './../proto/Ride.proto';

const ridePackageDef = protoLoader.loadSync(path.resolve(__dirname,
RIDE_PROTO_FILE));
const rideGrpcObj = (grpc.loadPackageDefinition(ridePackageDef) as unknown) as
ProtoGrpcType;

function main() {
    const server = getServer();
    //const serverCredentials = SSLService.getServerCredentials();
    const serverCredentials = grpc.ServerCredentials.createInsecure();

    server.bindAsync(`0.0.0.0:${PORT}`, serverCredentials,
        (err, port) => {
            if (err) {
                console.error(err)
                return
            }
            console.log(`ride server as started on port ${port}`)
            server.start()
        })
}

function getServer() {
    const server = new grpc.Server();

    server.addService(rideGrpcObj.DriveYourCity.IRideService.service, new
RideService())

    return server
}

main()
```

- En el archivo .env coloco el string de conexión

```
DATABASE_URL="postgresql://root@localhost:26000/driveyourcity?sslmode=disable"
```

- Debo levantar todos los servers corriendo Docker de fondo para poder probar la funcionalidad de ride (y las otras) con POSTMAN

Conclusiones

- Vale la pena tomar el tiempo para diseñar y construir con proto para luego pasar a la implementación
- gRPC nos permite comunicarnos de una forma rápida y económica entre servicios
- Una vez hecho el scaffolding la implementación es bastante rápida
- Cuando presentes un proyecto así, en lugar de comentarlo por funcionalidad cuenta una historia que haga que cobre sentido todo el trabajo
- gRPC se adapata de una manera muy elegante, escalable, util, sencilla para construir software eficiente