Nest

HELLO!

I am Karl Devooght

You can find me at @karl.devooght

I’m a frontend freelance developer

And a lot of fancy role names !

1.

The motivation

A (very) brief history of JS backend

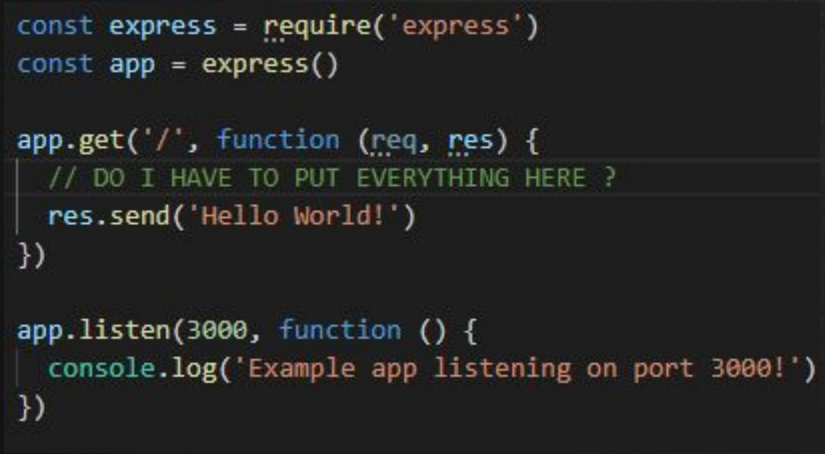
Node.js → Express.js → Hapi.js

27 May 2009

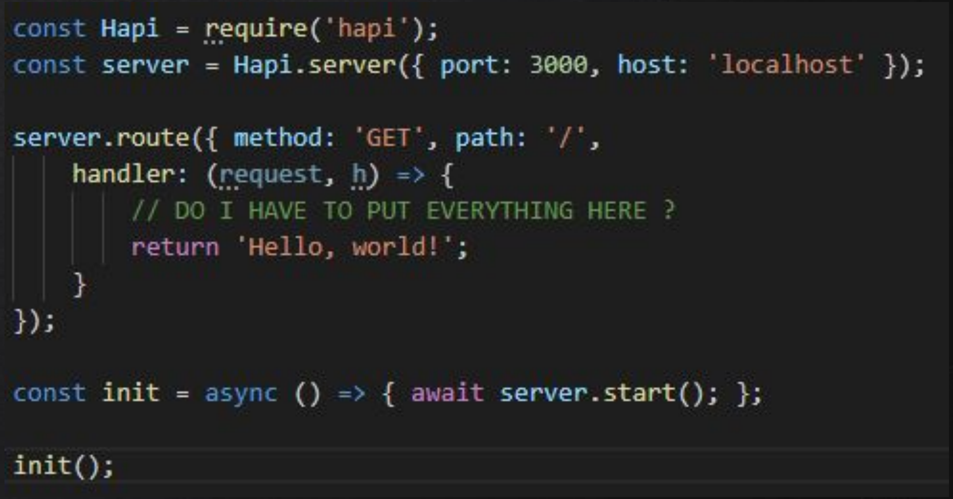
2010

2013

THAT 'S NOT MY ARCHITECTURE

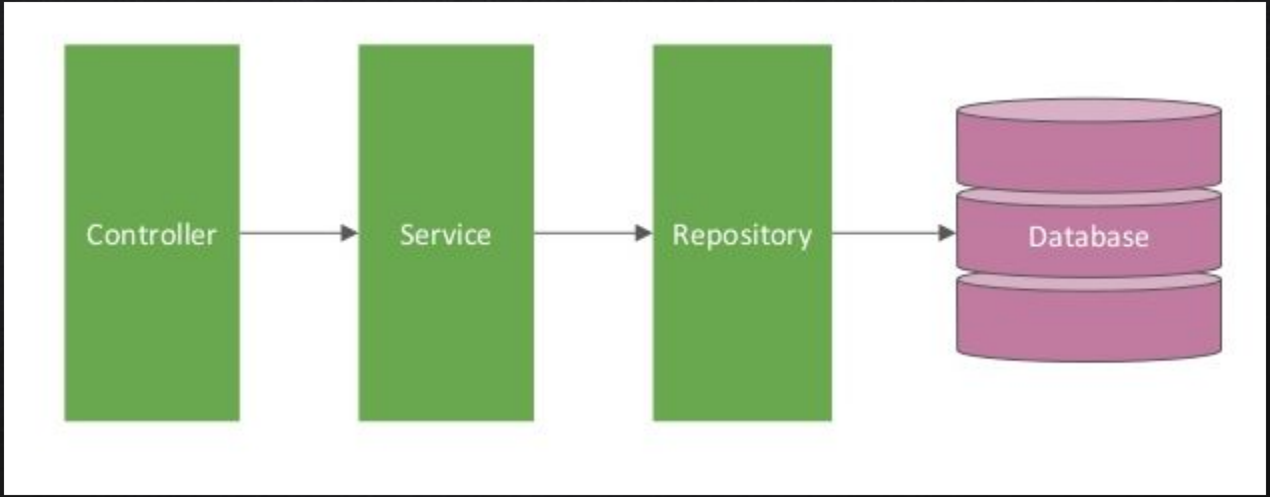


```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
  // DO I HAVE TO PUT EVERYTHING HERE ?
  res.send('Hello World!')
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!')
})
```



```
const Hapi = require('hapi');
const server = Hapi.server({ port: 3000, host: 'localhost' });

server.route({ method: 'GET', path: '/',
    handler: (request, h) => {
        // DO I HAVE TO PUT EVERYTHING HERE ?
        return 'Hello, world!';
    }
});

const init = async () => { await server.start(); };

init();
```

THAT 'S MY ARCHITECTURE



NEST IN A WORD

A backend framework inspired by a frontend framework

NEST IN A WORD

•

An higher level of abstraction based on Express or Fastify

NEST IN A WORD

•

Write in pure javascript or typescript (babel)

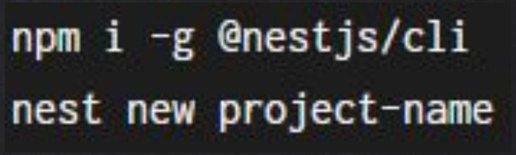
NEST IN A WORD

•

Come with a lot of tools, techniques & recipes

NEST in a second

•



```
npm i -g @nestjs/cli
nest new project-name
```

NEST in a (kind of) powerful demo

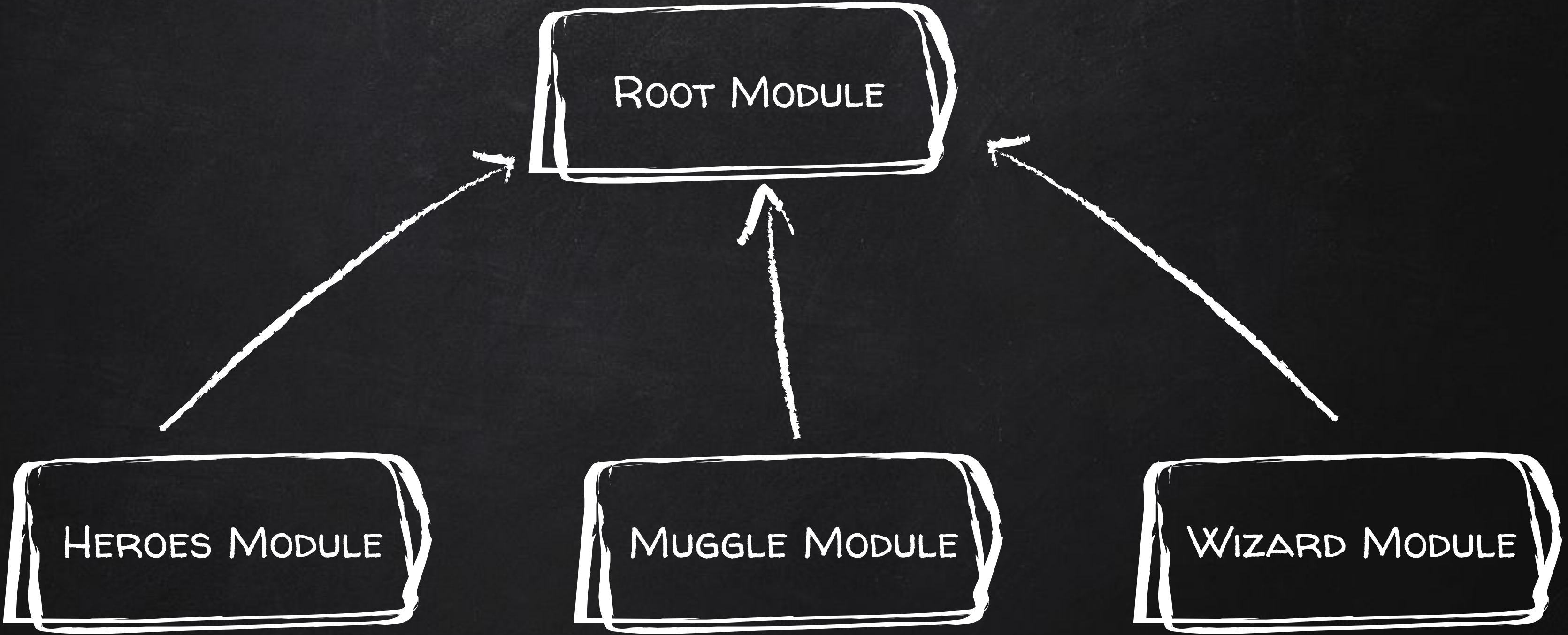
-

2.

The core

Module

- ✘ Container of cohesive code related to a part of the application domain
- ✘ At least one module => Root Module
- ✘ Feature Module & Shared Module
- ✘ Easily turn into micro-service


Root Module
Heroes Module
Muggle Module
Wizard Module

Controller

.

- ✘ Handling incoming request and returning response
- ✘ Massive use of decorators
  Routing, Request Params / Body, Headers, etc.
- ✘ Deal with asynchronicity
- ✘ One controller for all requests (by default)

Provider

.

- ✘ Something that can inject dependencies (IoC)
- ✘ Every Nest entity has a provider
- ✘ A class with @Injectable
- ✘ SOLID compliant

Custom Provider

•

Instance ID / Token	Instance Maker
String	useClass
Symbol	useValue
Class	useFactory
...	

Middleware

.

- ✘ Function called before route handling
- ✘ Access to request and response
- ✘ Same meaning than Express middleware
- ✘ Call next() when intend to move forward

Guard

.

- ✘ Determine if a request should be handled by a route handler
- ✘ Class with a canActivate method
- ✘ Access to the execution context (eg. route handler)
- ✘ Can enrich execution context with metadata

EXCEPTION

.

- ✘ Handle exceptions across the application
- ✘ Built-in Http Exception
- ✘ Exception filter

PIPE

- ✘ Transforms input data to desired output
- ✘ Can act as data validator using joi or class-validator
- ✘ Built-in pipes such as ValidatorPipe

3.

The techniques

```
git push -u origin mast
```

DATABASE

TypeORM

- ✘ Object Relational Mapper
- ✘ Support of a dozen of DB
- ✘ Repository Design Pattern
- ✘ Decorators (@Entity)

git push -u origin mast

DATABASE

•

Repository

- ✘ CRUD-based generated
- ✘ Custom Repository
- ✘ Just need to inject

```
git push -u origin mast
```

DATABASE

Misc

✘ Multi-database support

✘ Access to connection object

✘ Access to entity manager object

Configuration

Store config in the environment

-- Twelve Factor App

git push -u origin mast

Configuration

Dotenv

- ✘ Loads environment variable from env. file
- ✘ Use of parser service
- ✘ Inject anywhere into the app

git push -u origin master

CACHING

cache-manager

- ✘ Cache library for Node JS
- ✘ Unified API for storage providers
- ✘ Inject anywhere into the controllers

git push -u origin mast

SERIALIZATION

•

class-transformer

- ✘ Data manipulation before sending them
- ✘ Exclude, Transform & Compute properties
- ✘ ClassSerializerInterceptor by default

```
git push -u origin mast
```

COMPRESSION

compression

- ✘ JS library for compression
- ✘ Just an express middleware

Security

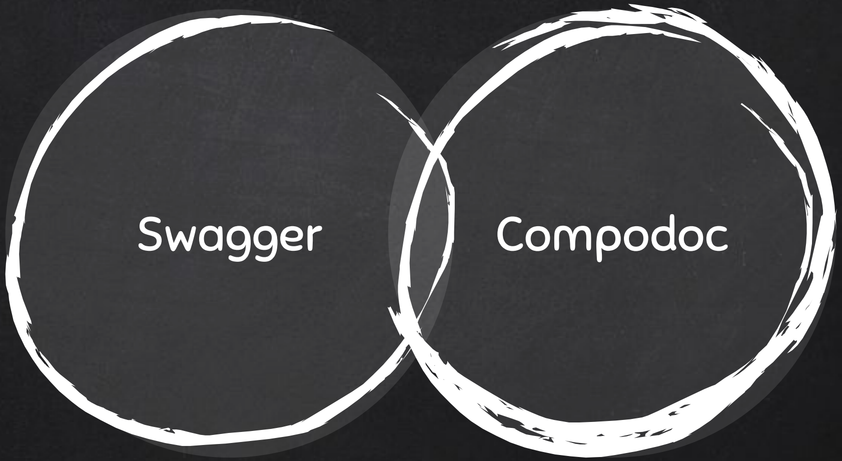
Helmet

Cors

Csurf

Rate-limit

Documentation



4.

GRAPHQL

DEFINITION

A query language for API

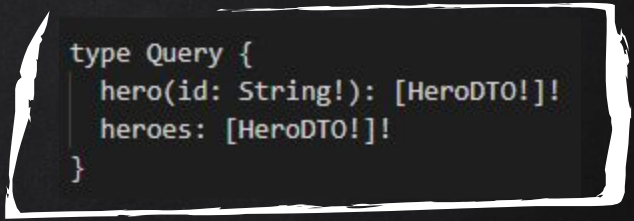
which allows to formulate requests as data structure

Pros

- ✘ Request what you want no more no less
- ✘ Multiple transports (HTTP, MQTT, WebSocket)
- ✘ “True” Single API for multiple clients
- ✘ Self-documenting through GraphQL schema

Graphql schema

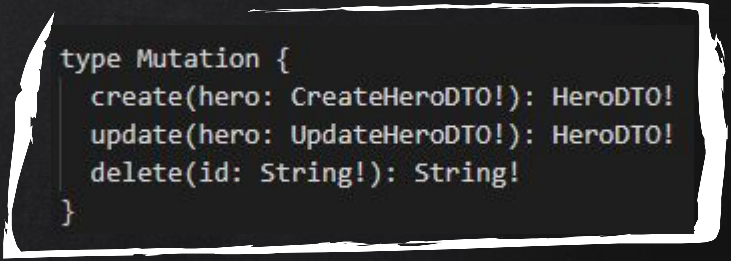
Query



```
type Query {
  hero(id: String!): [HeroDTO!]!
  heroes: [HeroDTO!]!
}
```

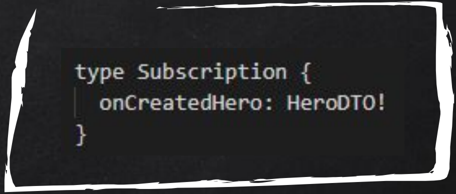
Graphql schema

Mutation


```
type Mutation {
  create(hero: CreateHeroDTO!): HeroDTO!
  update(hero: UpdateHeroDTO!): HeroDTO!
  delete(id: String!): String!
}
```

Graphql schema

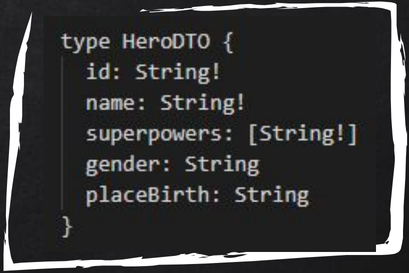
Subscription



```
type Subscription {
  onCreatedHero: HeroDTO!
}
```

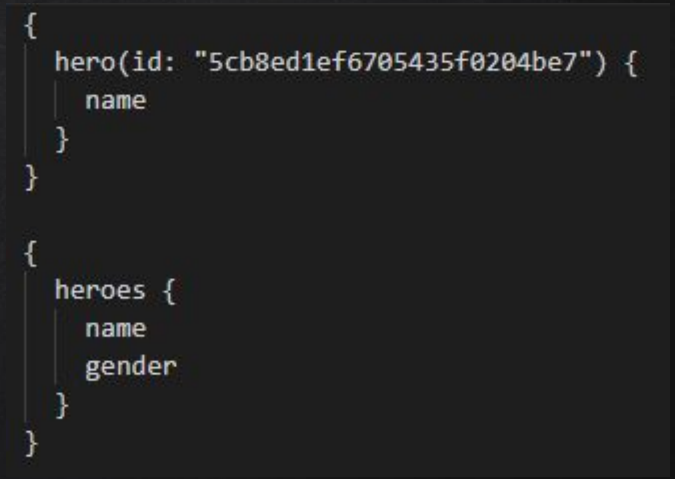
Graphql schema

Data structure


```
type HeroDTO {
  id: String!
  name: String!
  superpowers: [String!]
  gender: String
  placeBirth: String
}
```

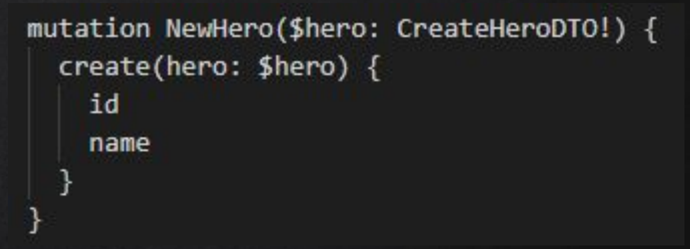
Graphql query

Query


```
{
  hero(id: "5cb8ed1ef6705435f0204be7") {
    name
  }
}

{
  heroes {
    name
    gender
  }
}
```

Graphql query

Mutation



```
mutation NewHero($hero: CreateHeroDTO!) {
  create(hero: $hero) {
    id
    name
  }
}
```

How can we do that in Nest ?

How can i switch from rest to graphql ?

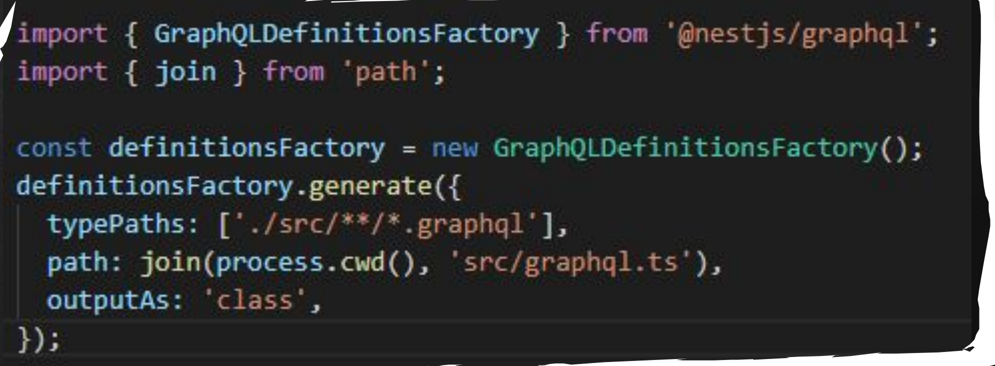
First approach

- ✘ Install some dependencies (Appolo Server & @nestjs/graphql)
- ✘ Define Graphql schema file (.graphql file)
- ✘ (Option) Generate related typings (ts-node generate-typings)
- ✘ Import and configure a Nest Graph module
- ✘ Convert controller to graphql resolver

First approach

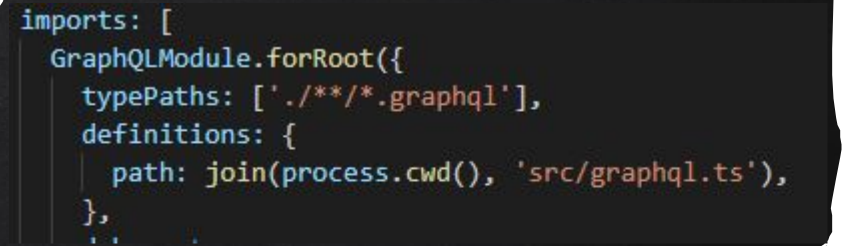
•

Generate Typings


```
import { GraphQLDefinitionsFactory } from '@nestjs/graphql';
import { join } from 'path';

const definitionsFactory = new GraphQLDefinitionsFactory();
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
});
```

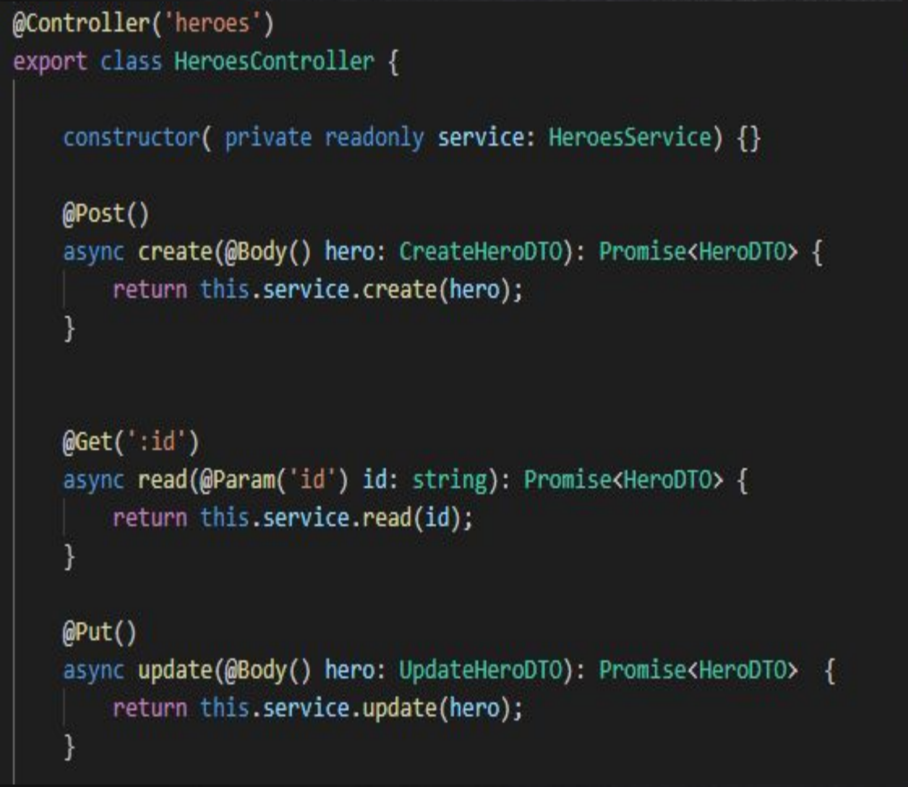
First approach

Graphql Module



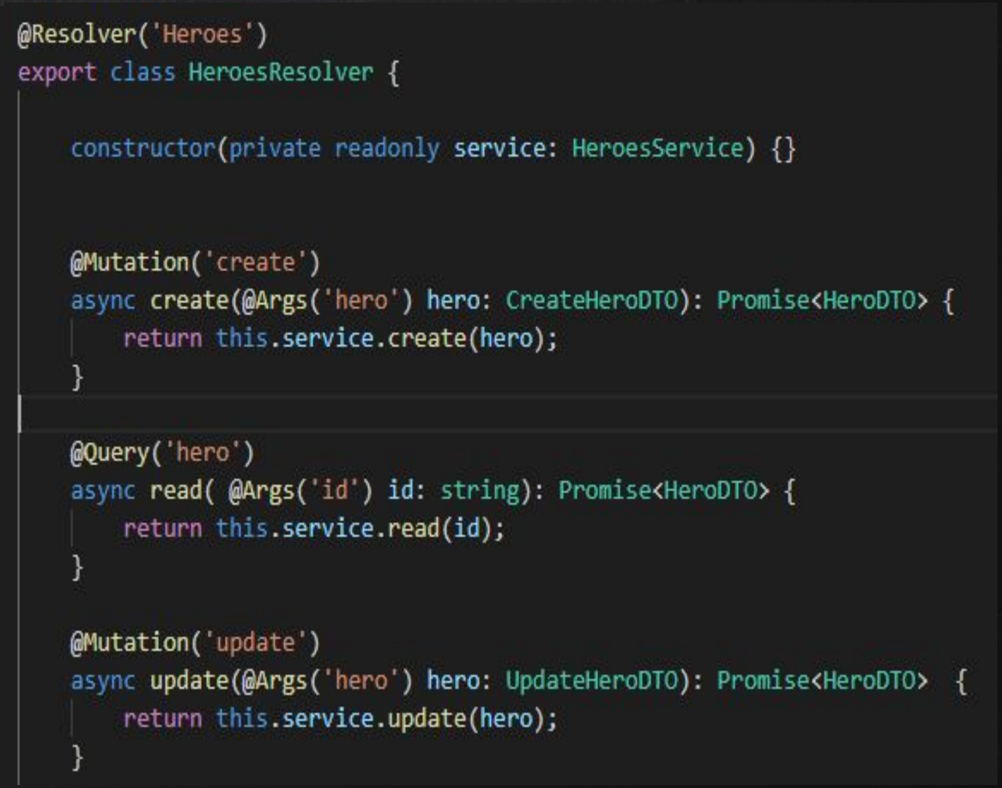
```
imports: [
  GraphQLModule.forRoot({
    typePaths: ['./**/*.graphql'],
    definitions: {
      path: join(process.cwd(), 'src/graphql.ts'),
    },
```

CONTROLLER



```
@Controller('heroes')
export class HeroesController {

  constructor( private readonly service: HeroesService) {}

  @Post()
  async create(@Body() hero: CreateHeroDTO): Promise<HeroDTO> {
      return this.service.create(hero);
  }


  @Get(':id')
  async read(@Param('id') id: string): Promise<HeroDTO> {
      return this.service.read(id);
  }

  @Put()
  async update(@Body() hero: UpdateHeroDTO): Promise<HeroDTO>  {
      return this.service.update(hero);
  }
```

RESOLVER



```
@Resolver('Heroes')
export class HeroesResolver {

  constructor(private readonly service: HeroesService) {}


  @Mutation('create')
  async create(@Args('hero') hero: CreateHeroDTO): Promise<HeroDTO> {
      return this.service.create(hero);
  }

  @Query('hero')
  async read( @Args('id') id: string): Promise<HeroDTO> {
      return this.service.read(id);
  }

  @Mutation('update')
  async update(@Args('hero') hero: UpdateHeroDTO): Promise<HeroDTO>  {
      return this.service.update(hero);
  }
```

Nice but…

I don't want to learn a new syntax

I don't want to generate typings

Just keep focus on my TS code

SECOND APPROACH

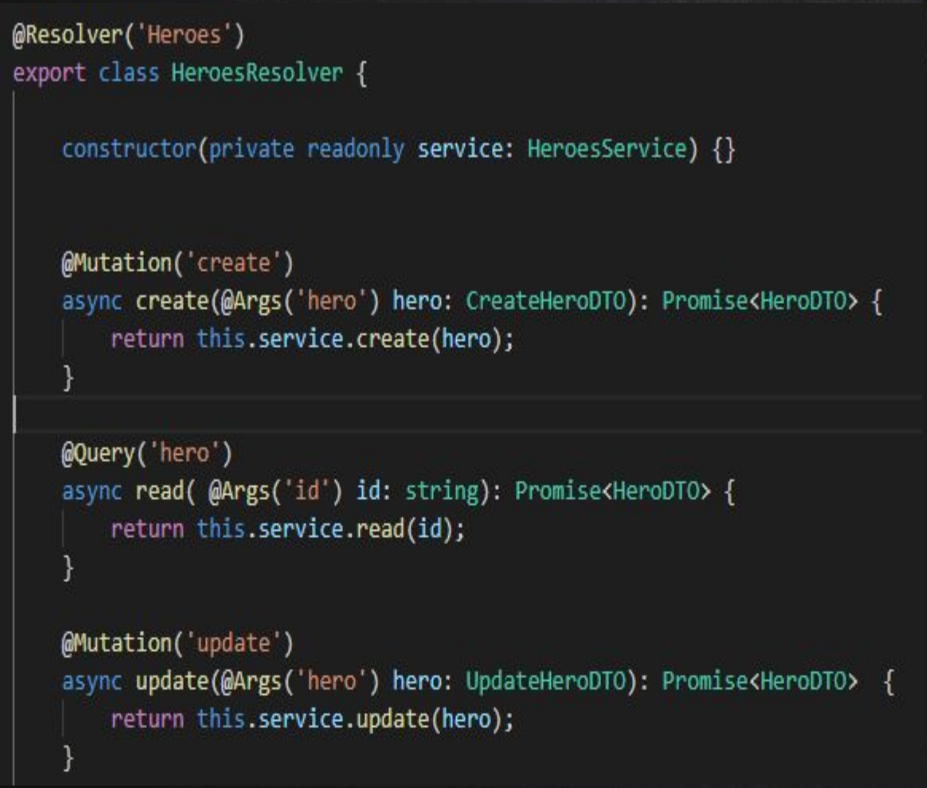
✘ Install some dependencies (Appolo Server & @nestjs/graphql)

✘ Import and configure a Nest Graph module

✘ Decorate your DTO types with type-graphql decorators

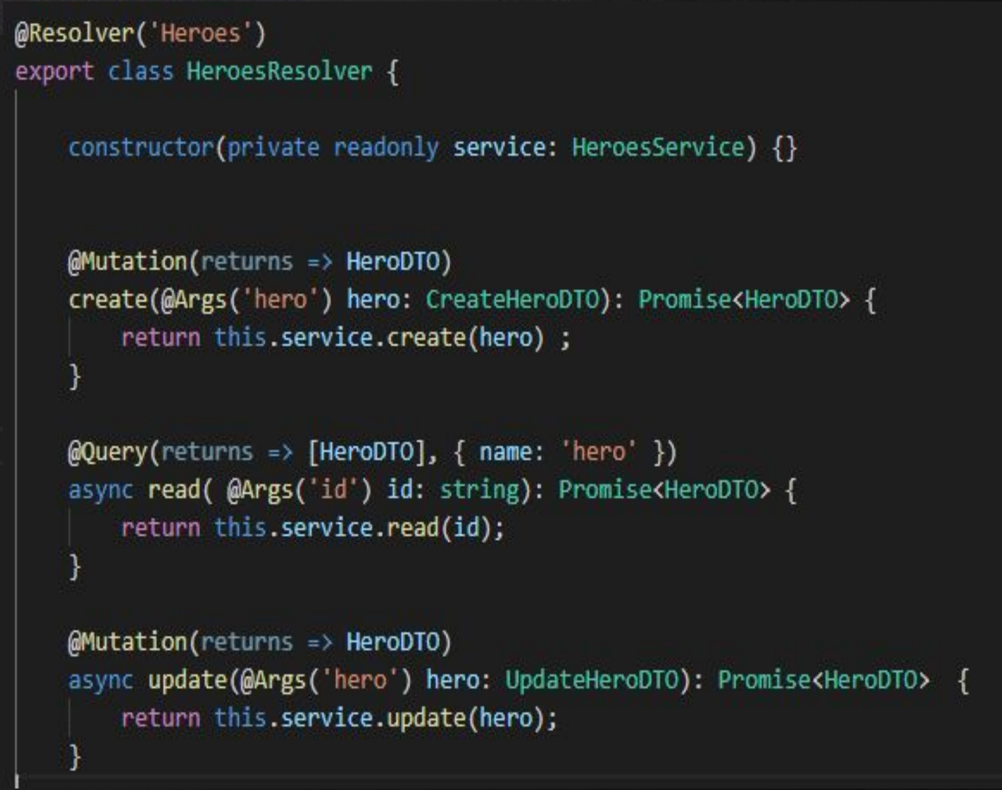
✘ Convert controller to graphql resolver with a bit of further description

RESOLVER



```
@Resolver('Heroes')
export class HeroesResolver {

    constructor(private readonly service: HeroesService) {}


    @Mutation('create')
    async create(@Args('hero') hero: CreateHeroDTO): Promise<HeroDTO> {
        return this.service.create(hero);
    }

    @Query('hero')
    async read( @Args('id') id: string): Promise<HeroDTO> {
        return this.service.read(id);
    }

    @Mutation('update')
    async update(@Args('hero') hero: UpdateHeroDTO): Promise<HeroDTO>  {
        return this.service.update(hero);
    }
```

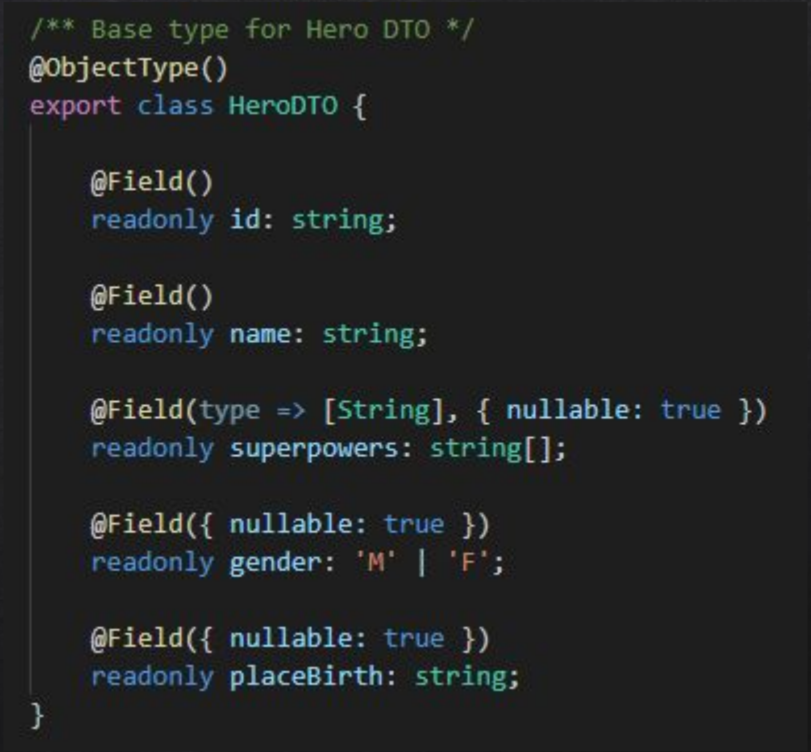
New resolver



```
@Resolver('Heroes')
export class HeroesResolver {

    constructor(private readonly service: HeroesService) {}


    @Mutation(returns => HeroDTO)
    create(@Args('hero') hero: CreateHeroDTO): Promise<HeroDTO> {
        return this.service.create(hero) ;
    }

    @Query(returns => [HeroDTO], { name: 'hero' })
    async read( @Args('id') id: string): Promise<HeroDTO> {
        return this.service.read(id);
    }

    @Mutation(returns => HeroDTO)
    async update(@Args('hero') hero: UpdateHeroDTO): Promise<HeroDTO>  {
        return this.service.update(hero);
    }
```

DTO Class as graphql data type

-



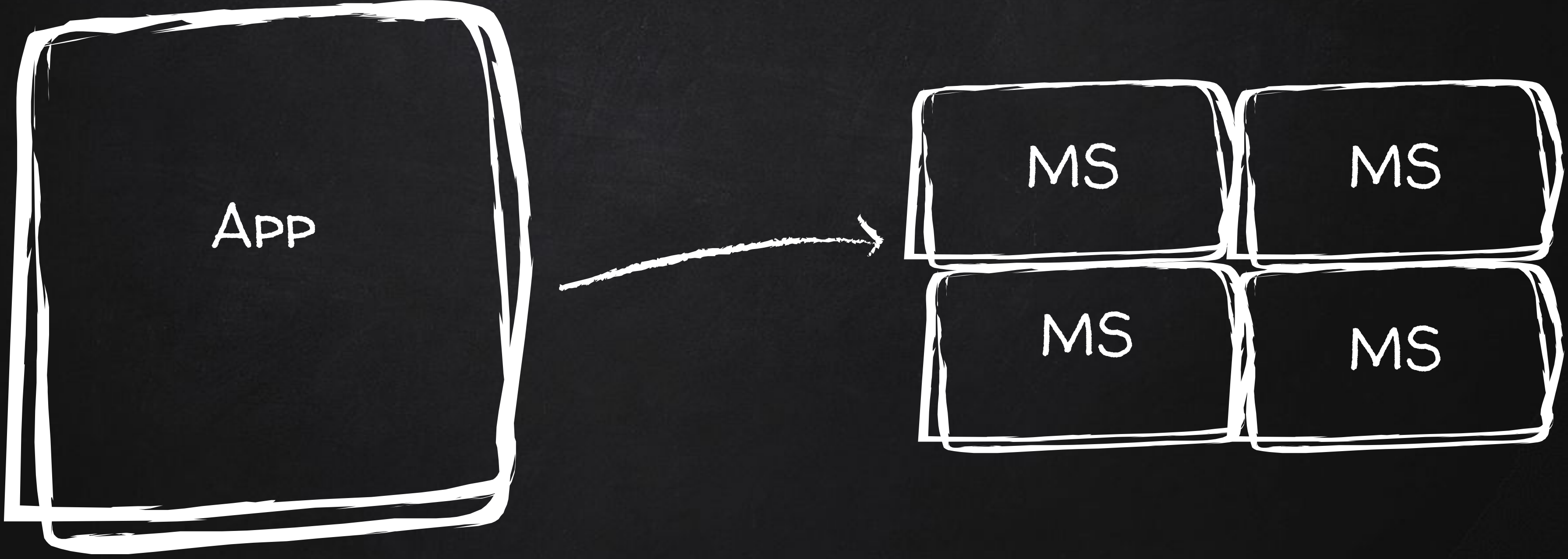
```
/** Base type for Hero DTO */
@ObjectType()
export class HeroDTO {

    @Field()
    readonly id: string;

    @Field()
    readonly name: string;

    @Field(type => [String], { nullable: true })
    readonly superpowers: string[];

    @Field({ nullable: true })
    readonly gender: 'M' | 'F';

    @Field({ nullable: true })
    readonly placeBirth: string;
}
```

5.

MICROSERVICES

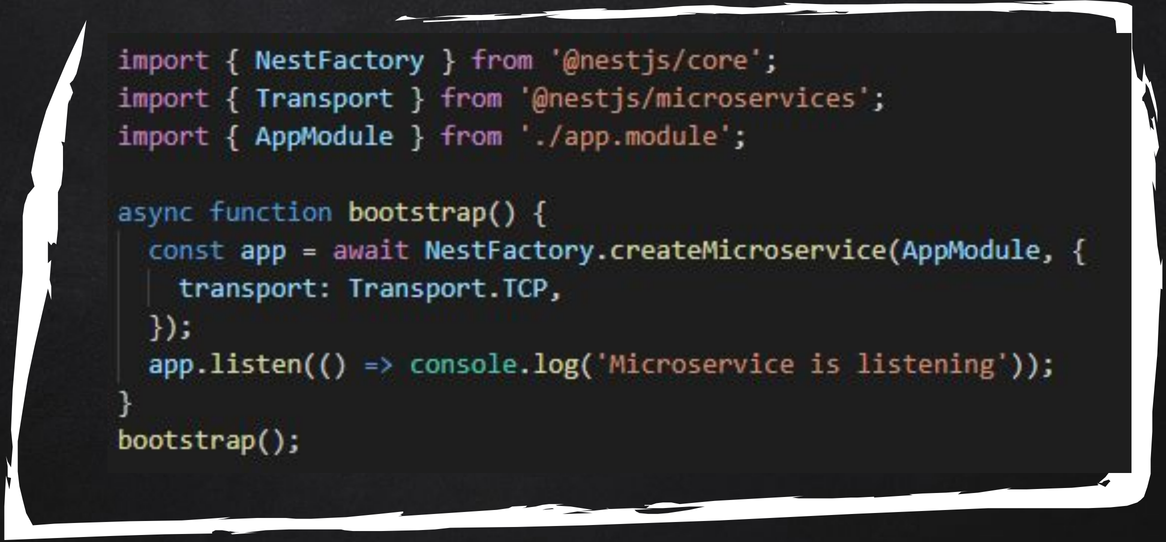
Microservices are a software development technique that structures an application as a collection of loosely coupled services

Define a microservice



Define a microservice

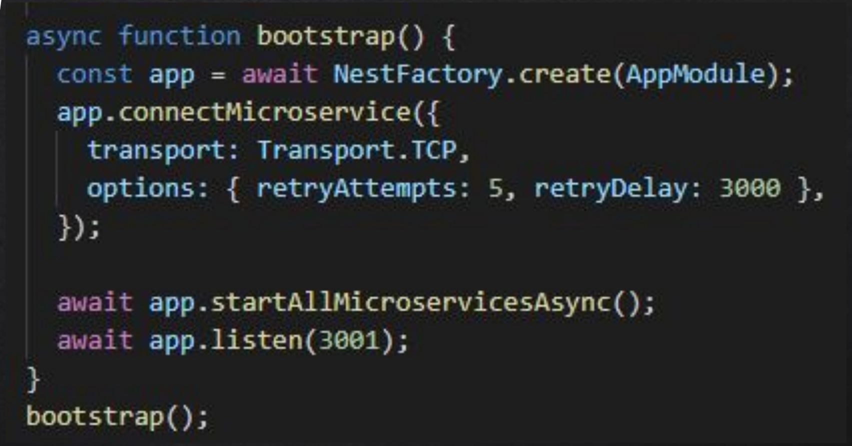
•



```
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.TCP,
  });
  app.listen(() => console.log('Microservice is listening'));
}
bootstrap();
```

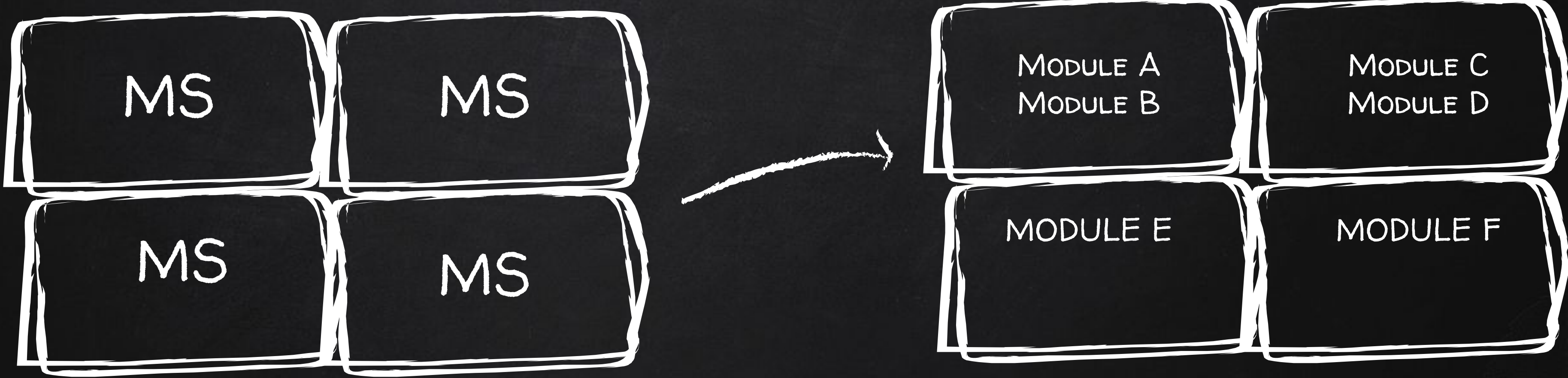
Define a microservice

•



```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.connectMicroservice({
    transport: Transport.TCP,
    options: { retryAttempts: 5, retryDelay: 3000 },
  });

  await app.startAllMicroservicesAsync();
  await app.listen(3001);
}
bootstrap();
```

Define a microservice



Communication approach

Abstract the communication by using pattern

Pattern is just a js object !

Communication transports

TCP

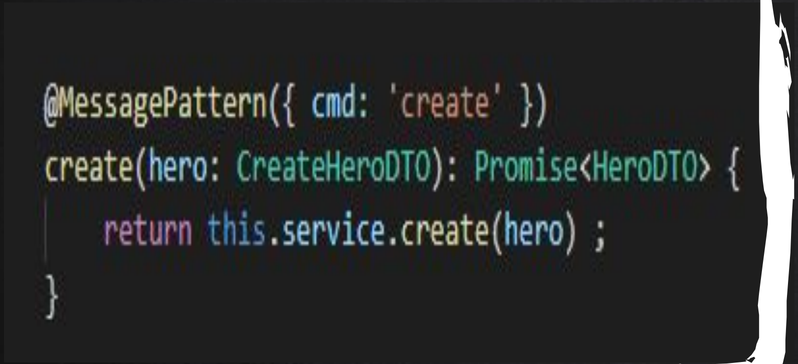
Redis

MQTT

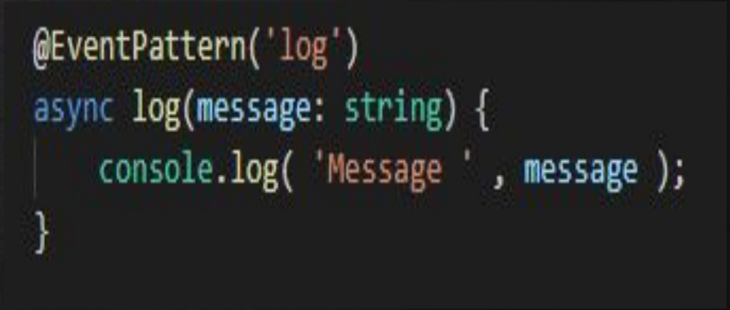
NATS

gRPC

Listening messages

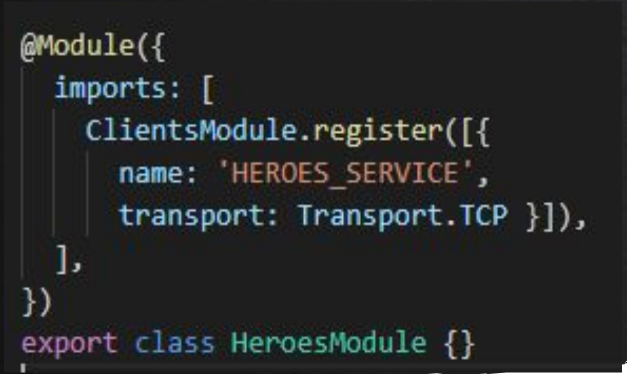


```
@MessagePattern({ cmd: 'create' })
create(hero: CreateHeroDTO): Promise<HeroDTO> {
    return this.service.create(hero) ;
}
```

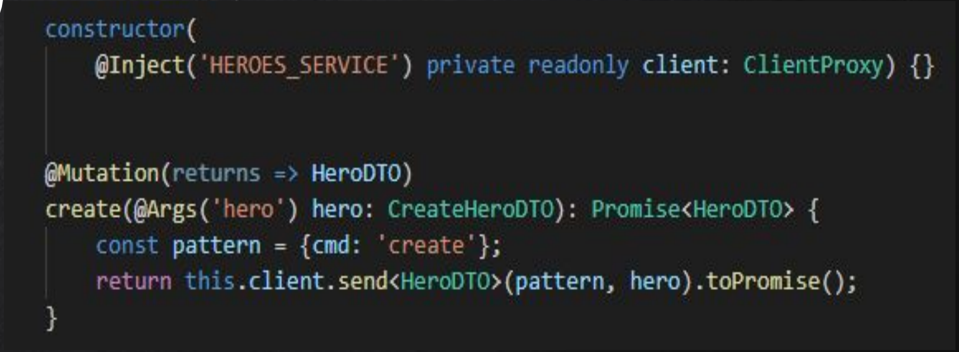


```
@EventPattern('log')
async log(message: string) {
    console.log( 'Message ' , message );
}
```

Sending messages



```
@Module({
  imports: [
    ClientsModule.register([{
      name: 'HEROES_SERVICE',
      transport: Transport.TCP }]),
  ],
})
export class HeroesModule {}
```



```
constructor(
    @Inject('HEROES_SERVICE') private readonly client: ClientProxy) {}


@Mutation(returns => HeroDTO)
create(@Args('hero') hero: CreateHeroDTO): Promise<HeroDTO> {
    const pattern = {cmd: 'create'};
    return this.client.send<HeroDTO>(pattern, hero).toPromise();
}
```

THANKS!

Any questions?

You can find me at
@karl.devooght
karl.devooght@gmail.com

HELP

Looking for a new freelance opportunity

Front-end Development / Leading (Angular)

Voxxed days luxembourg

Microfrontend

or how to break your frontend in thousand pieces