

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

آموزش یادگیری فریم ورک Nestjs

نگارنده:

سورنا سعیدی

فهرست مطالب

4	مقدمه
5	ایجاد پروژه ی Nestjs
6	ایجاد ماژول
7	کنترلر
8	سرویس
10	برقراری ارتباط با دیتا بیس (Mongodb)
10	خواندن اطلاعات از دیتا بیس
14	نوشتن اطلاعات در دیتا بیس
18	ایجاد یک Middleware
20	ایجاد یک سیستم شناسایی کاربر با nestjs
20	فرآیند ایجاد یک کاربر (register)
24	فرآیند ورود یک کاربر (Login)
28	تعریف گارد برای API ها براساس توکن دسترسی
31	ارزیابی داده های ورودی
34	فرآیند ایجاد یک گارد برای محدود سازی نرخ درخواست ها به یک API
37	مستندسازی API ها با Swagger
48	کش کردن داده های یک API
52	ایجاد session برای کاربران
60	تعیین مجوز و نقش برای کاربران (Authorization)
71	اقدامات امنیتی
71	Helmet
77	Cross-Origin Resource Sharing (CORS)
81	معماری میکروسرویس
84	تفاوت در معماری Arciteture
85	مرزبندی سرویس ها
85	نحوه ذخیره سازی داده ها
85	حاکمیت در سرویس های مختلف
85	اندازه دامنه

ارتباط بین سرویس‌ها..... 86

Deployment 86

دسترسی به سرویس‌ها 86

پیاده سازی معماری میکروسرویس 87

مقدمه

فریم ورک Nest.js یکی دیگر از پلتفرم‌ها و چهارچوب‌های طراحی شده برای کسانی است که می‌خواهند بر پایه انگولار کار کنند. با استفاده از چهارچوب نست جی اس به راحتی می‌توانید از تکنولوژی‌های برنامه‌نویسی شی گرا و تابع نویسی و همین‌طور برنامه‌نویسی تابعی واکنشی استفاده کنید که در حال حاضر یکی از محبوب‌ترین ساختارها برای توسعه جاوا اسکریپت به حساب می‌آید. در واقع برای کسانی که قصد برنامه‌نویسی حرفه‌ای در حوزه وب را دارند. می‌توانند از فریم ورک Nest.js استفاده کنند. از طرفی این فریم ورک بر پایه پروگروسیو نود جی اس است با استفاده از تایپ اسکریپت طراحی شده است.

اوپن سورس بوده و به شدت قدرتمند است. شرکت‌های بزرگ حوزه توسعه کسب و کارهای دیجیتال از جمله گوگل از آن استفاده می‌کنند. در واقع فریم ورک Nest.js بک‌اند است و با استفاده از آن شما می‌توانید بر پایه نود جی اس، به راحتی وب‌سایت و اپلیکیشن‌های تحت وب استفاده کنید.

در این گزارش سعی شده تا آموزش فریم ورک Nest.js برای کاربری‌های عمومی قرار داده شود تا کسانی که قصد شروع به یادگیری این فریم ورک دارند بتوانند به عنوان شروع از آن بهره ببرند.

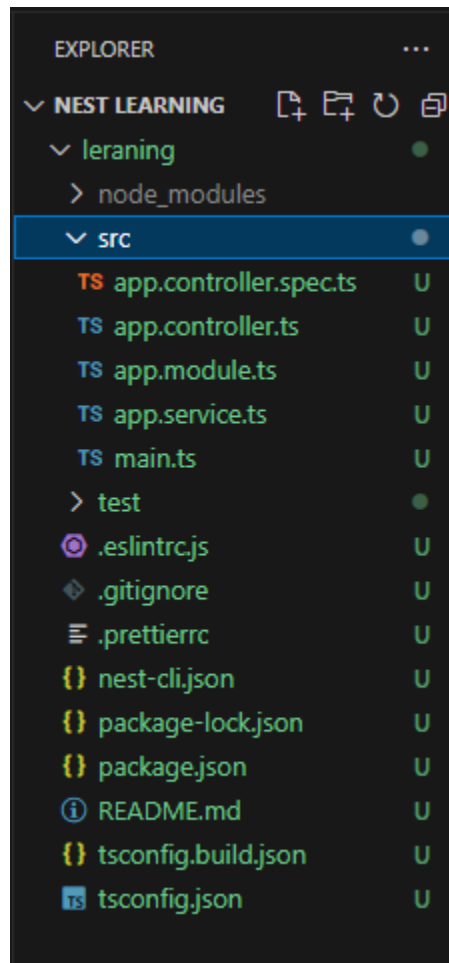
ایجاد پروژه ی Nestjs

برای ایجاد یک پروژه ی Nestjs کافی است دستور زیر را در بخش ترمینال اجرا کنید:

```
npm i -g @nestjs/cli
```

```
nest new project-name
```

با اجرای کد فوق شالوده ی کلی پروژه به شکل زیر ایجاد می گردد:



ایجاد ماژول

ماژول ها در Nestjs در واقع برای هسته ی اصلی سرویس ها هستند هر قاعدا برای هر گروه از سرویس ها یک ماژول تعریف می گردد. زمانی که یک پروژه Nestjs را ایجاد می کنیم یک ماژول به نام app داریم که ماژول اصلی سرویس است و تمامی ماژول های دیگر باید در این بخش تعریف گردند مثال زیر یک نمونه از ماژول app که درواقع نشان دهنده ی صفحه ی اصلی سایت است.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

برای ایجاد یک ماژول جدید از دستور زیر استفاده می شود:

nest g res users

با اجرای این کد یک ماژول کامل به همراه سرویس و کنترلر را در اختیار می می گذارد و ماژول مربوطه اتوماتیک به ماژول های app اضافه می گردد.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UsersModule } from './users/users.module';

@Module({
  imports: [UsersModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

▼	users	●
>	dto	●
>	entities	●
TS	users.controller.spec.ts	U
TS	users.controller.ts	U
TS	users.module.ts	U
TS	users.service.spec.ts	U
TS	users.service.ts	U

کنترلر

در این بخش باید نوع API تعریف شده و پردازش مد نظر بر روی داده ها تعیین گردد گیرد به عنوان مثال ابتدا یک API از نوع GET تعریف می کنیم که با اجرای آن یک متن ارسال گردد:

```
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UsersService) {}

  @Get()
  simpleGet() {
    return this.userService.simpleGet();
  }
}
```

حال اگر بخواهیم یک API از نوع GET بنویسیم که بتواند از کاربر ID دریافت نماید به صورت زیر نوشته می شود:

```
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UsersService) {}

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.userService.findOne(+id);
  }
}
```

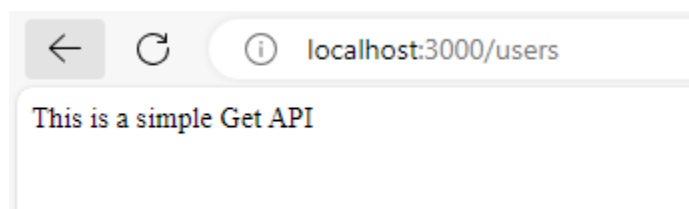
برای نوشتن API از نوع پست نیز به صورت زیر عمل می کنیم:

```
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UsersService) {}

  @Post()
  createUser(@Body() createUserDto: CreateUserDto) {
    return this.userService.create(createUserDto);
  }
}
```

در این بخش پردازش مد نظر بر روی API بخش کنترلر را تعیین می کنیم:

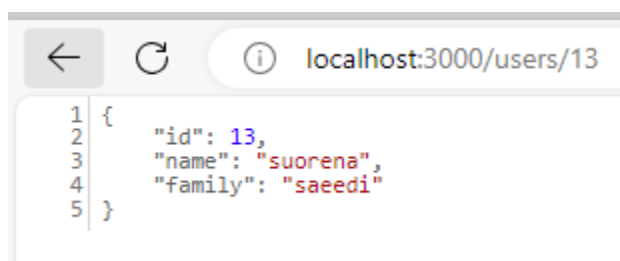
```
@Injectable()
export class UsersService {
  simpleGet() {
    return `This is a simple Get API`;
  }
}
```



برای گرفتن یک آیدی از کاربر و نمایش اطلاعات یا API تعریف شده با نام findOne در بخش قبل به صورت زیر عمل می کنیم:

```
@Injectable()
export class UsersService {
  create(createUserDto: CreateUserDto) {
    return 'This action adds a new user';
  }

  FindOne(id:number){
    return{id,
      name:"suorena",
      family:"saeedi"}
  }
}
```



برای پست کردن اطلاعات یک کاربر و نمایش آن ها یا API تعریف شده با نام UsersController در بخش قبل به صورت زیر عمل می کنیم:

```
@Injectable()
export class UsersService {
  create(createUserDto: CreateUserDto) {
    return {statusCode:"201",createUserDto};
  }
}
```

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:3000/users
- Body Type:** JSON
- Request Body:**

```
1 { "name": "suorena",
2   "family": "saeedi",
3   "age": 50 }
```
- Response:** 201 Created, 43 ms, 322 B
- Response Body (Pretty):**

```
1 {
2   "statusCode": "201",
3   "createUserDto": {
4     "name": "suorena",
5     "family": "saeedi",
6     "age": 50
7   }
8 }
```

برقراری ارتباط با دیتابیس (Mongodb)

در این داکيومنت جهت برقراری ارتباط با Mongodb از کتابخانه ی استفاده شده است. برای برقراری ارتباط با این دیتابیس ابتدا باید این کتابخانه به صورت زیر نصب گردد:

```
npm i @nestjs/mongoose mongoose
```

در مرحله ی بعد به کمک این کتابخانه باید دیتابیس را به کمک این کتابخانه در بخش app.module وارد نماییم:

```
@Module({
  imports: [MongooseModule.forRoot('*****'),
    UsersModule],
  controllers: [AppController],
  providers: [AppService],
})
```

سپس با اجرای کد می توانیم مطمئن شویم که به دیتا بیس وصل شده ایم یا نه اگر خطایی در اجرای برنامه وجود نداشت یعنی اتصال به درستی برقرار شده است.

خواندن اطلاعات از دیتا بیس

در تایپ اسکریپ تایپ تمامی متغیرها با تعریف گردد برای پایبندی به این مسیر چه برای خواندن چه نوشتن داده در دیتا بیس باید شکل ده های ورودی را تعیین نماییم برای این کار در پوشه سورس یک فایل ts ایجاد کرده و در درون آن شکل تایپ مدنظر برای دادهایی که قصد خواندن آن ها را داریم تعیین می کنیم:

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import * as mongoose from 'mongoose';

@Schema({ collection: 'users' })
export class User extends Document {
  @Prop()
  _id: string;

  @Prop()
  id: number;

  @Prop()
  inner_id: string;

  @Prop()
```

```

    name: string;

    @Prop()
    slug: string;

    @Prop()
    shortName: string;

    @Prop()
    nameCode: string;

    @Prop()
    persian_name: string;

    @Prop()
    flag: string;
}

export const UserSchema = SchemaFactory.createClass(User);

export const UserModel = mongoose.model<User>('User', UserSchema);

```

در درون دکوراتور Schema می توان نام کالکشنی را که قصد خواندن داده های آن را داریم تعیین نماییم اگر چنین نکنیم یک کالکشن به صورت جمع نام مدل تعریف شده برای ما در دیتابیس ایجاد می گردد.

در داخل دکوراتور Prop می توان ویژگی های داده ها را تعریف کرد

```

@Prop({ required: true })
_id: string

```

پس از تعریف این مدل و Schema در مدول user که می خواهیم از طریق آن API برای خواندن دیتابیس مدنظر بنویسیم مدل ایجاد شده را به صورت زیر وارد می کنیم:

```

@Module({
  imports: [MongooseModule.forFeature([{ name: User.name, schema: UserSchema }])],
  controllers: [UsersController],
  providers: [UsersService],
})

```

اکنون قصد داریم از این کالکشن یکبار کل داده ها را بخوانیم و بار دیگر داده ها را براساس آیدی بخوانیم برای این منظور ابتدا در بخش سرویس کد را به شکل زیر تغییر می دهیم:

```
@Injectable()
export class UsersService {
  constructor(@InjectModel(User.name) private readonly userModel: Model<User>) {}

  async findAll(): Promise<User[]> {
    const result = await this.userModel.find().exec()
    return result;
  }

  async findone(id: number): Promise<object | null> {
    const result = await this.userModel.findOne({ id });
    return result;
  }
}
```

ابتدا در کانستراکتور مدل ایجاد شده را اینجکت می کنیم در این توابع به کمک `async` و `await` فرایند دریافت اطلاعات را سنکرونایز می کنیم و تایپ هردو تابع را از نوع `Promise` تعریف می کنیم حال باید این توابع را در بخش کنترلر فراخوانی کنیم.

```
@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get('/:id')
  async findOne(@Param('id') id: number) {
    return await this.usersService.findone(id);
  }

  @Get()
  async simpleGet() {
    return await this.usersService.findAll();
  }
}
```

به این ترتیب می توانیم داده ها را با یک API دریافت کنیم:

```
← ↻ ⓘ localhost:3000/users
1 [
2   {
3     "id": "65421aa35dfc553a6c349a84",
4     "id": 4819,
5     "inner_id": "None",
6     "name": "Argentina",
7     "slug": "argentina",
8     "shortName": "Argentina",
9     "nameCode": "ARG",
10    "persian_name": "None",
11    "flag": "flag/4819.png"
12  },
13  {
14    "id": "65421aa35dfc553a6c349a85",
15    "id": 4481,
16    "inner_id": "None",
17    "name": "France",
18    "slug": "france",
19    "shortName": "France",
20    "nameCode": "FRA",
21    "persian_name": "None",
22    "flag": "flag/4481.png"
23  },
24  {
25    "id": "65421aa35dfc553a6c349a86",
26    "id": 4748,
27    "inner_id": "None",
28    "name": "Brazil",
29    "slug": "brazil",
30    "shortName": "Brazil",
31    "nameCode": "BRA",
32    "persian_name": "None",
33    "flag": "flag/4748.png"
34  },
35  {
36    "id": "65421aa35dfc553a6c349a87",
37    "id": 4713,
38    "inner_id": "None",
39    "name": "England",
40    "slug": "england",
41    "shortName": "England",
42    "nameCode": "ENG",
43    "persian_name": "None",
44    "flag": "flag/4713.png"
45  },
46  {
47    "id": "65421aa35dfc553a6c349a88",
48    "id": 4717,
49    "inner_id": "None",
50    "name": "Belgium",
51    "slug": "belgium",
52    "shortName": "Belgium",
53    "nameCode": "BEL",
54    "persian_name": "None",
55    "flag": "flag/4717.png"
56  },
57 ]
```

```
← ↻ ⓘ localhost:3000/users/4819
1 {
2   "id": "65421aa35dfc553a6c349a84",
3   "id": 4819,
4   "inner_id": "None",
5   "name": "Argentina",
6   "slug": "argentina",
7   "shortName": "Argentina",
8   "nameCode": "ARG",
9   "persian_name": "None",
10  "flag": "flag/4819.png"
11 }
```

نوشتن اطلاعات در دیتابیس

در این بخش مدلی که تعریف کرده ایم در بخش قبل اهمیت دو چندان می یابد و باید داده ها را به شکل آن مدل به API بدهیم. در این بخش تعیین میکنیم که داده باید دارای id باشد:

```
@Schema({ collection: 'users' })
export class User extends Document {

  @Prop({ required: true })
  id: number;

  @Prop()
  inner_id: string;

  @Prop()
  name: string;

  @Prop()
  slug: string;

  @Prop()
  shortName: string;

  @Prop()
  nameCode: string;

  @Prop()
  persian_name: string;

  @Prop()
  flag: string;
}
```

در ادامه یک تایپ هم برای داده های ورودی به صورت یک کلاس در یک فایل جداگانه تعریف می کنیم:

```
export class CreateUserDto {
  readonly id: number;
  readonly inner_id: string;
  readonly name: string;
  readonly slug: string;
  readonly shortName: string;
  readonly nameCode: string;
  readonly persian_name: string;
  readonly flag: string;}
}
```

سپس در بخش سرویس ابتدا در کانستراکتور مدل ایجاد شده را اینجکت می کنیم در این تابع مربوط به پست را به کمک `async` و `await` فرایند دریافت اطلاعات را سنکرونایز می کنیم و تایپ تابع را از نوع `Promise` که داخل آن به شکل مدل `User` تعریف می کنیم حال باید این تابع را در بخش کنترلر فراخوانی کنیم.

```
@Injectable()
export class UsersService {
  constructor(@InjectModel(User.name) private readonly userModel: Model<User>) {}

  async create(createUserDto: CreateUserDto): Promise<User> {
    const createdUser = new this.userModel(createUserDto);
    return await createdUser.save();
  }

  async findAll(): Promise<User[]> {
    const result = await this.userModel.find().exec()
    return result;
  }

  async findone(id: number): Promise<object | null> {
    const result = await this.userModel.findOne({ id });
    return result;
  }
}
```

در نهایت این در بخش کنترلر API پست را تعریف می کنیم:

```
@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get('/:id')
  async findOne(@Param('id') id: number) {
    return await this.usersService.findone(id);
  }

  @Post()
  async createUser(@Body() createUserDto: CreateUserDto): Promise<User> {
    return await this.usersService.create(createUserDto);
  }
}
```

حال API پست را تست می کنیم:

The screenshot shows the Postman interface for a POST request to `http://localhost:3000/users`. The request body is a JSON object with the following fields:

```
1 { "id": 13,
2   "inner_id": "test",
3   "name": "test_user",
4   "slug": "TU",
5   "shortName": "tU",
6   "nameCode": "test_U",
7   "persian_name": "یوزر تست",
8   "flag": "تستی" }
```

The response is displayed in the bottom section, showing a 201 status code and the following JSON body:

```
1 {
2   "id": 13,
3   "inner_id": "test",
4   "name": "test_user",
5   "slug": "TU",
6   "shortName": "tU",
7   "nameCode": "test_U",
8   "persian_name": "یوزر تست",
9   "flag": "تستی",
10  "_id": "65c0e51bbe728557dff19f89",
11  "__v": 0
12 }
```

FootService.users

0 1
DOCUMENTS INDEXES

Documents Aggregations Schema Indexes Validation

Filter {id:13} Generate query Explain Reset Find Options

ADD DATA EXPORT DATA

1 - 1 of 1

```
_id: ObjectId('65c0e51bbe728557dff19f89')
id: 13
inner_id: "test"
name: "test_user"
slug: "TU"
shortName: "tU"
nameCode: "test_U"
persian_name: "یوزر تست"
flag: "تستی"
__v: 0
```


اما اگر json ارسالی فاقد id باشد همانطور که در Schema تعریف کردیم داده ها ذخیره نمی شود و خطای 500 میگیریم:

The screenshot shows a Postman interface with a POST request to `http://localhost:3000/users`. The request body is a JSON object with the following fields:

```
1 {
2   "inner_id": "test",
3   "name": "test_user",
4   "slug": "TU",
5   "shortName": "tU",
6   "nameCode": "test_U",
7   "persian_name": "یوزر تست",
8   "flag": "تستی"
9 }
```

The response status is `500 Internal Server Error` with a response time of `20 ms` and a size of `388 B`. The response body is a JSON object:

```
1 {
2   "statusCode": 500,
3   "message": "Internal server error"
4 }
```

ایجاد یک Middleware

Middleware ها در واقع در مسیر درخواست کاربر و سرویس قرار می گیرند و با انجام پردازش یا بررسی درخواست اجازه ی عبور و رسیدن آن به سرویس را می دهند در این بخش قصد داریم یک Middleware را جهت لاگ کردن اطلاعات درخواست بنویسیم که اطلاعاتی نظیر ip درخواست دهنده، مدت زمان ارسال پاسخ و حجم درخواست را برای ما نمایش دهد. برای ایجاد یک Middleware در پروژه کافی است از دستور زیر استفاده کنیم:

nest g mi logger

با اجرای این کد یک فایل ts برای ما ساخته می شود که در آن یک کلاس ارث برده از NestMiddleware وجود دارد کلاس را به صورت زیر باز نویسی می کنیم:

```
import { Injectable, Logger, NestMiddleware } from '@nestjs/common';
import { Request, Response } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private logger = new Logger('HTTP');
  use(req: Request, res: Response, next: () => void) {
    const {ip, method, baseUrl}=req;
    const userAgent = req.get('user-agent') || '';
    const startAt = process.hrtime();
    res.on('finish',()=>{
      const {statusCode} = res
      const contentLenght = res.get('content-length');
      const dif = process.hrtime(startAt);
      const responseTime = dif[0]*1e3 + dif[1]*1e-6;
      this.logger.log(
        `Method :${method} , baseUrl:${baseUrl} , statusCode:${statusCode} ,
contentLenght:${contentLenght} , responseTime:${responseTime}ms ,
userAgent:${userAgent} , ip:${ip},`
      )
    })
    next();
  }
}
```

حال باید این کلاس را در بخش ماژول های برنامه تعریف کنیم:

```
@Module({
  imports: [MongooseModule.forRoot(*****),
    UsersModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(LimitMiddleware).forRoutes('*');
  }
}
```

متد configure برای فریم ورک nest بوده و جهت تعریف Middleware استفاده می گردد. حال این Middleware برای تمام سرویس ها فعال شده است کافی است یک درخواست به API ها بزنیم تا لاگ مربوط به آن ایجاد گردد:

localhost:3000/users:

LOG [HTTP] Method :GET , baseUrl:/users , statusCode:304 , contentLenght:undefined ,
responseTime:40.34679999999995ms , userAgent:Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36 Edg/121.0.0.0 , ip:::1

ethod :POST , baseUrl:/users , statusCode:201 , contentLenght:187 ,
responseTime:13.544699999999999ms , userAgent:PostmanRuntime/7.36.0 , ip:::1,

localhost:3000/users/13:

Method :GET , baseUrl:/users/13 , statusCode:200 , contentLenght:187 ,
responseTime:7.724399999999999ms , userAgent:Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36 Edg/121.0.0.0 , ip:::1,

ایجاد یک سیستم شناسایی کاربر با nestjs:

فراآیند ایجاد یک کاربر(register)

برای این پروژه ابتدا باید کتابخانه های زیر را نصب کنیم:

```
npm i @nestjs/jwt
npm i passport
npm i bcryptjs
npm i @nestjs/passport
npm i passport-jwt
npm i type/passport-jwt
npm install express-session
```

سپس یک سرویس به نام Auth برای این منظور با دستور زیر می سازیم:

```
nest g res auth
```

در این بخش باید Api های مربوط به Login و register را تعریف کنیم در گام اول باید یک API برای تعریف کاربر یا به اصطلاح Register تعریف کنیم. برای این منظور ابتدا در بخش کنترلر این سرویس را تعریف می کنیم:

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @Post('register')
  register(@Body() RegisterDto: RegisterDto) {
    return this.authService.register(RegisterDto);
  }
}
```

API های Login و Register هر دو از نوع پست هستند. در این بخش کاربر را به شکل RegisterDto باید وارد کنیم که آن را به صورت زیر تعریف می کنیم:

```
export class RegisterDto {
  id: number;
  email: string;
  first_name: string;
  last_name: string;
  age: number;
  password: string;
}
```

سپس در بخش سرویس باید منطق ساخت کاربر را تعریف کنیم:

```
@Injectable()
export class AuthService {
  [x: string]: any;
  constructor(@InjectModel(User.name) private readonly userModel: Model<User>,
    private readonly usersService: UsersService,
    private readonly JwtService: JwtService) {}
  async register(registerDto: RegisterDto) {
    const user = await this.userModel.findOne({email: registerDto.email
  }).exec();
    if(user){
      throw new HttpException("User already exist",400);
    }
    registerDto.password = await bcrypt.hash(registerDto.password,10);
    return await this.usersService.create(registerDto)
  }
}
```

در اینجا فرض بر این است که کاربرها در دیتابیس وجود دارند بنابراین ابتدا در باید چک کنیم که کاربر در دیتابیس مجمود نباشد اگر نبود ابتدا به کمک `bcrypt.hash` پسورد کاربر را هش می کنیم و سپس با استفاده از سرویس `usersService.create` کاربر را ایجاد می کنیم لازم به ذکر است که برای این منظور باید `usersService` در `provider` های ماژول مربوطه ایجاد نماییم. همچنین `CreateUserDto` را نیز مطابق نیاز مسئله به شکل زیر تغییر می دهیم و همچنین `schema` دیتابیس را نیز به شکل زیر تغییر می دهیم:

CreateUserDto

```
export class CreateUserDto {
  id: number
  email: string;
  first_name: string;
  last_name: string;
  age: number;
  password: string;
}
```

Schema:

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import * as mongoose from 'mongoose';

@Schema({ collection: 'users' })
export class User extends Document {
  @Prop()
  id: number;

  @Prop()
  email: string;

  @Prop()
  first_name: string;

  @Prop()
  last_name: string;

  @Prop()
  age: string;

  @Prop()
  password: string;
}

export const UserSchema = SchemaFactory.createForClass(User);

export const UserModel = mongoose.model<User>('User', UserSchema);
```

حال به کمک این API می توانیم کاربر جدید تعریف کنیم و در دیتابیس ذخیره کنیم:

The screenshot shows a REST client interface for a 'Football Service'. The request is a POST to 'http://localhost:3000/Auth/register'. The body is a JSON object with user registration details. The response is a 201 status code with a JSON body containing the created user's ID, email, first name, last name, age, password, a JWT token, and a UUID.

```
1 {
2   "id": 111,
3   "email": "learn_nest@gmail.com",
4   "first_name": "suorena",
5   "last_name": "saeedi",
6   "age": 28,
7   "password": "12345"
8 }
```

Body Cookies Headers (7) Test Results (0/1) 201 Created 150 ms 452 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 111,
3   "email": "learn_nest@gmail.com",
4   "first_name": "suorena",
5   "last_name": "saeedi",
6   "age": "28",
7   "password": "$2a$10$LZXYstAAelpMIfS/uS40puhRK7W1p9JETthweWT0RQJ/hW77ZB7y",
8   "_id": "65cb17a71dc0686ebd51226d",
9   "__v": 0
10 }
```

حال اگر مجددا همین فایل json را بفرستیم با خطای تکراری بودن کاربر مواجه خواهیم شد:

The screenshot shows the same REST client interface, but the response is a 400 Bad Request status code. The JSON body contains an error message: 'User already exist'.

```
1 {
2   "statusCode": 400,
3   "message": "User already exist"
4 }
```

Body Cookies Headers (7) Test Results (0/1) 400 Bad Request 19 ms 293 B Save as example

etty Raw Preview Visualize JSON

فرآیند ورود یک کاربر (Login)

برای این کار یک API جهت Login کردن کاربر می نویسیم ابتدا بخش کنترلر این API پست را می نویسیم:

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @Post('register')
  register(@Body() registerDto: RegisterDto) {
    return this.authService.register(registerDto);
  }

  @Post('login')
  login(@Body() LoginDto: LoginDto) {
    return this.authService.login(LoginDto);
  }
}
```

سپس LoginDto را مطابق منطقی که برای ورود در نظر داریم تعریف می کنیم:

```
export class LoginDto {
  email: string;
  password: string;
}
```

نکته ای که در اینجا وجود دارد این است که ما پسورد کاربر را در بخش قبل به دلیل مسائل امنیتی به صورت هش شده در دیتابیس ذخیره کردیم بنابراین برای چک کردن پسورد باید این نکته را دزر نظر بگیریم. با این فرض به سراغ نوشتن منطق API در بخش سرویس ها می شویم:

```
@Injectable()
export class AuthService {
  [x: string]: any;
  constructor(@InjectModel(User.name) private readonly userModel: Model<User>,
    private readonly usersService: UsersService,
    private readonly JwtService: JwtService) {}
  async register(registerDto: RegisterDto) {
    const user = await this.userModel.findOne({"email": registerDto.email
  }).exec();
    if(user){
      throw new HttpException("User already exist",400);
    }
    registerDto.password = await bcrypt.hash(registerDto.password,10);
    return await this.usersService.create(registerDto)
  }
}
```



```

}
async login(LoginDto: LoginDto) {
  const user = await this.userModel.findOne({email: LoginDto.email }).exec();
  if(!user){
    throw new HttpException("User not found",404);
  }
  const ispasswordMatch = await bcrypt.compare(LoginDto.password,user.password)
  if(!ispasswordMatch){
    throw new HttpException("Password is wrong",400);
  }
  const accessToken = this.JwtService.sign({
    sub: user.id,
    email: user.email
  })
  console.log(ispasswordMatch)
  return {access_Token:accessToken};
}
}

```

در این بخش ابتدا از طریق ایمیل موجود بودن کاربر را در دیتابیس بررسی می کنیم سپس به کمک `bcrypt.compare` اگر کاربر موجود بود چک می کنیم که آیا پسورد را درست وارد کرده است یا خیر اگر پسورد درست وارد شده باشد به کمک `JwtService` از `id` و `email` کاربر یک توکن ساخته و توکن را در هدر ذخیره می کنیم با این توکن قصد داریم در آینده سطح دسترسی کاربر را تعیین نماییم.

برای کار با این سرویس باید در بخش ماژول های این API کانفیگ شده و افزوده گردد:

```

@Module({
  imports:[MongooseModule.forFeature([ { name:User.name, schema: UserSchema } ]),
    UsersModule,
    JwtModule.register({
      secret:'secret2',
      signOptions:{expiresIn:'1d'}
    }),],
  controllers: [AuthController],
  providers: [AuthService,UsersService,JwtStrategy],
})
export class AuthModule {}

```

حال باید عملکرد این API را چک کنیم:

Football Service / Get tractor in nearest event

From Get not started event data

Save

POST

http://localhost:3000/Auth/login

Send

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautify

```
1 {
2   ... "email": "learn_nest@gmail.com",
3   ... "password": "12345"
4 }
```

Body

Cookies

Headers (7)

Test Results (0/1)

201 Created

119 ms

443 B

Save as example

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJleXMSwizW1haWwiOiJsZWYybm9uZXR0b3R0dWlsLmNvbSIsIm1hdCI6MTcwNzg4MDgyMCwiZXhwIjoxNzA3ODk3MjIwLnQ.1XPZKRDAZzhkMsiNfb005y1jgmhz0bTNUbZlqg40"
3 }
```

می‌توان دید که توکن دسترسی کاربر به خوبی تهیه و ارسال گردید حال اگر کاربر وجود نداشته باشد بررسی می‌کنیم که عملکرد API چگونه است:

POST http://localhost:3000/Auth/login Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ...."email": "learn_nest15@gmail.com",
3   ...."password": "12345"
4 }
```

Body Cookies Headers (7) Test Results (0/1) 404 Not Found 21 ms 287 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "statusCode": 404,
3   "message": "User not found"
4 }
```

در این بخش نیز عملکرد قابل قبولی را شاهد بودیم حال اگر کاربر پسورد خود را نادرست وارد کند:

Football Service / Get tractor in nearest event From Get not started event data Save

POST http://localhost:3000/Auth/login Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ...."email": "learn_nest@gmail.com",
3   ...."password": "1234567"
4 }
```

Body Cookies Headers (7) Test Results (0/1) 400 Bad Request 92 ms 292 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "statusCode": 400,
3   "message": "Password is wrong"
4 }
```

بنابراین می توان دید که API مربوط به Login به خوبی منطق مد نظر ما را در بر می گیرد در ادامه به سراغ توکن دسترسی رفته و برای API های خود یک guard می نویسیم.

تعریف گارد برای API ها براساس توکن دسترسی

در بخش قبل توکن دسترسی را تولید کردیم اکنون قصد ایجاد گارد را برای API ها داریم. برای این منظور باید یک استراتژی و یک گارد تعریف نماییم. برای ایجاد گارد از دستور زیر استفاده می کنیم:

npm i gu

با ایجاد فایل گارد به صورت دستی نیز یک فایل استراتژی ایجاد می کنیم:

```
import { Injectable } from "@nestjs/common";
import { PassportStrategy } from "@nestjs/passport";
import { Strategy, ExtractJwt } from 'passport-jwt';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: 'secret2'
    });
  }

  async validate(payload: any) {
    return {
      id: payload.sub,
      email: payload.email,
    };
  }
}
```

در بخش استراتژی تعیین می کنیم که گارد باید توکن دسترسی را از کجا بخواند و کلید دیکد کردن این توکن چیست و همچنین اگر این کد اکسپایر شده بود نیز باز هم آنرا تایید کند و همچنین به متد `validate` ایجاد می کنیم که در آن تعیین می کنیم در توکن دسترسی کاربر چه اطلاعاتی نهفته است. پس از تعریف استراتژی نوبت به تعریف گارد میرسد:

```
import { Injectable, ExecutionContext } from "@nestjs/common";
import { AuthGuard } from "@nestjs/passport";
import { UnauthorizedException } from "@nestjs/common";

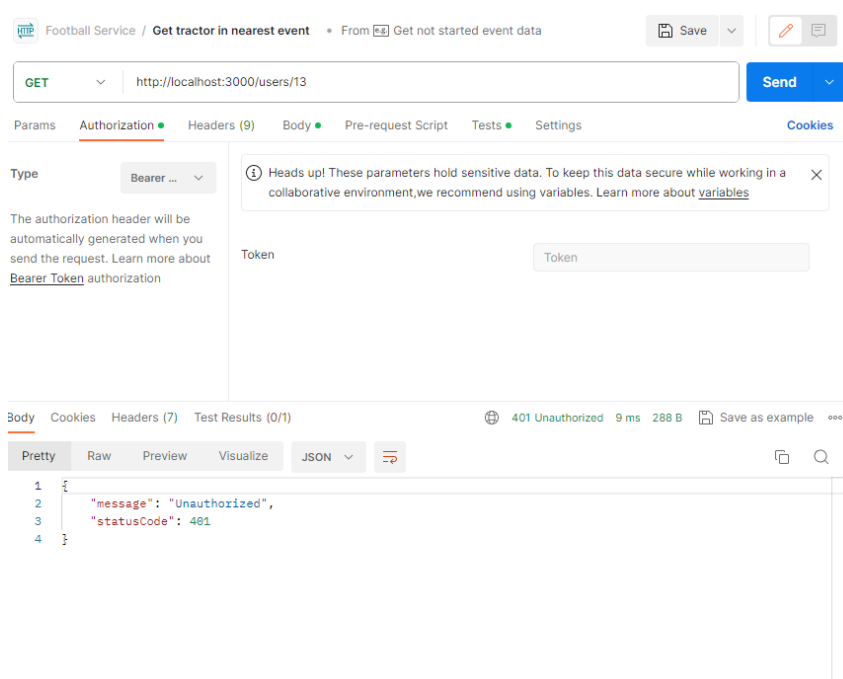
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  canActivate(context: ExecutionContext) {
    return super.canActivate(context);
  }

  handleRequest(err, user, info) {
    // You can throw an error based on certain conditions
    if (err || !user) {
      throw err || new UnauthorizedException();
    }
    return user;
  }
}
```

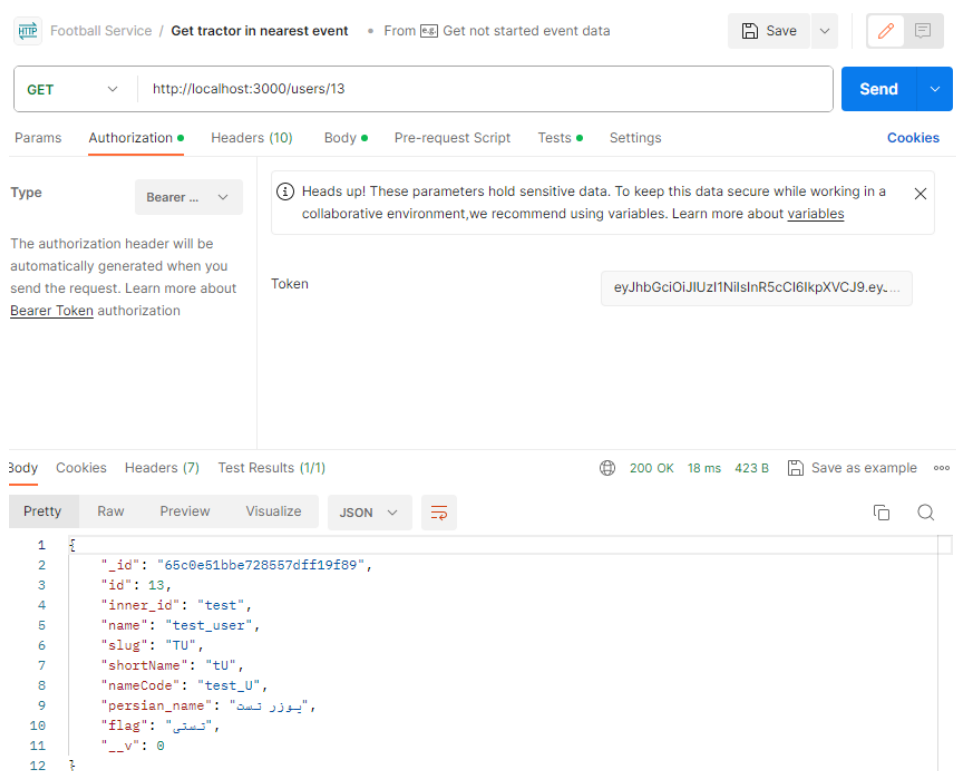
این گارد با استراتژی تعریف شده در ارتباط است اگر توکن کاربر غلط باشد یا موجود نباشد اجازه دسترسی را به او نمی دهد حال کافی است این گارد را به عنوان یک دکوراتور برای API مد نظلمان تعریف کنیم:

```
@Get('/:id')
@UseGuards(JwtAuthGuard)
async find(@Param('id') id: number) {
  return await this.usersService.findone(id);
}
```

حال باید امتحان کرد که آیا API مد نظر بدون توکن دسترسی کار می کند یا خیر:



حال توکن دسترسی را به هدر درخواست اضافه می کنیم:



به این ترتیب کسی بدون در اختیار داشتن توکن مربوطه امکان استفاده از این API را ندارد.

ارزیابی داده های ورودی :

برای این منظور ابتدا پکیج های زیر را نصب می کنیم:

```
npm i class-validator
```

```
npm i class-transformer
```

سپس برای این که امکان استفاده از ولیدیشن وجود داشته باشد در پوشه ی main پروژه باید کانفیگ اولیه صورت پذیرد:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as passport from 'passport';
import * as session from 'express-session';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({
    whitelist:true
  }))
  app.use(
    session({
      secret: '1317abcd', // Change this to your desired secret key
      resave: false,
      saveUninitialized: false,
    }),
  );
  app.use(passport.initialize());
  app.use(passport.session());
  await app.listen(3000);
}
bootstrap();
```

همانطور که در کد فوق پیداست یک ValidationPipe ساخته شده و تنها تنظیمی که بر روی آن اعمال کرده ایم تنظیمات whitelist:true به این ترتیب در روند ولیدیشن اگر کاربر پارامتر اضافه تر از dto را ارسال کند نیز جلوی ارسال این پارامتر گرفته خواهد شد.

پس از تعریف این مسیر تنها کافی است در `dto` مربوطه با استفاده از دکوراتور های این کتابخانه ها ولیدشن های مد نظرمان را اعمال کنیم به عنوان مثال می توان کد زیر را بررسی کرد:

```
import { IsEmail, IsNotEmpty, IsNumber, IsOptional, IsString, MaxLength,
MinLength } from "class-validator";

export class CreateUserDto {
  @IsNumber()
  @IsNotEmpty()
  id: number
  @IsString()
  @IsNotEmpty()
  @IsEmail()
  email: string;
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  first_name: string;
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  last_name: string;
  @IsNumber()
  @IsOptional()
  @Max(150)
  age: number;
  @IsNotEmpty()
  @IsString()
  password: string;
}
```


حال با این روند ولیدشن ورودی سعی می کنیم یک کاربر جدید ایجاد کنیم:

Football Service / Get tractor in nearest event • From Get not started event data

POST http://localhost:3000/auth/register

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ... "id": 1121,
3   ... "email": "learn_nest78@gmail.com",
4   ... "first_name": "suorena",
5   ... "last_name": "saeedi",
6   ... "age": 180,
7   ... "password": "12345"
8 }
```

Body Cookies Headers (7) Test Results (0/1) 400 Bad Request 45 ms 357 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": [
3     "email must be an email",
4     "age must not be greater than 160"
5   ],
6   "error": "Bad Request",
7   "statusCode": 400
8 }
```

حال اگر ایرادات وارده را اصلاح کنیم باید کاربر ایجاد گردد:

Football Service / Get tractor in nearest event • From Get not started event data

POST http://localhost:3000/auth/register

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ... "id": 1121,
3   ... "email": "learn_nest79@gmail.com",
4   ... "first_name": "suorena",
5   ... "last_name": "saeedi",
6   ... "age": 18,
7   ... "password": "12345"
8 }
```

Body Cookies Headers (7) Test Results (0/1) 201 Created 134 ms 455 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1121,
3   "email": "learn_nest79@gmail.com",
4   "first_name": "suorena",
5   "last_name": "saeedi",
6   "age": "18",
7   "password": "$2a$10$uxVHNZ3JfFadWLC3d2AVbeeIaAy8mGPe7V5FYzeSLZrEJkFhMobB.",
8   "_id": "65cc6698f04b11606e22e9cb",
9   "__v": 0
10 }
```

فرآیند ایجاد یک گارد برای محدود سازی نرخ درخواست ها به یک API

یکی از راه های جلوگیری از حملات احتمالی به سایت اعمال محدودیت برروی تعداد درخواست هایی است که کاربران می توانند به یک API بدهند برای این کار باید ابتدا کتابخانه ی زیر را نصب کنیم:

```
npm i --save @nestjs/throttler
```

سپس این کتابخانه را در بخش ماژول سرویسی که قصد استفاده از آن را داریم کانفیگ می کنیم(ما قصد داریم در سرویس users از این کتابخانه استفاده کنیم):

users.module.ts

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { UserSchema, User } from 'src/NatinalteamModel';
import { CacheModule } from '@nestjs/cache-manager';
import { ThrottlerModule } from '@nestjs/throttler';

@Module({
  imports: [MongooseModule.forFeature([{ name: User.name, schema: UserSchema }]),
    CacheModule.register(),
    ThrottlerModule.forRoot([
      {
        ttl: 5 * 60 * 1000,
        limit: 10
      }
    ]),
  ],
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule {}
```

در کنترلر سرویس مدنظر ThrottlerGuard را به عنوان یک گارد به صورت زیر اضافه می کنیم به این صورت API مد نظر محدودیت نرخ پاسخ خواهد داشت و اگر در 5 دقیقه بیشتر از یک درخواست به آن ارسال گردد خطا باز می گرداند:

users.controller.ts

```
import { ThrottlerGuard } from '@nestjs/throttler';

@Controller('users')
@ApiTags("Users")
export class UsersController {
  constructor(private readonly usersService: UsersService,
    @Inject(CACHE_MANAGER) private cacheManager: Cache
  ) {}

  @Get()
  @UseGuards(ThrottlerGuard)
  // @UseGuards(JwtAuthGuard)
  @ApiHeader({
    name: "Authorization",
    description: "Send Authorization Token"
  })
  @ApiResponse({
    status: 200, description: "Data send properly", type: CreateUserDto
  })
  @ApiBearerAuth()
  @ApiOperation({ summary: 'This API return all users ' })
  async simpleGet() {
    // Check if c has a value
    const allUsers = await this.cacheManager.get('users')
    console.log(allUsers)
    if (allUsers) {
      return allUsers
    }
    else{
      const allUsers = await this.usersService.findAll();
      await this.cacheManager.set('users', allUsers, 100000)
      return allUsers
    }
  }
}
```

پس از ده بار درخواست داریم:

The screenshot displays the Postman web application interface. At the top, a green notification bar states: "An update has been downloaded for Postman. Restart now to install the update." with a "Restart" button. The main workspace shows a collection named "Football Service" with a request titled "Get tractor in nearest event". The request is a GET method to the URL "http://localhost:3000/users/". The "Authorization" tab is selected, showing a "Bearer ..." token type and a text input field containing a long token string. Below the request details, the "Body" tab is active, displaying the response in "Pretty" JSON format. The response is a 429 status code with the message "ThrottlerException: Too Many Requests". The status bar at the bottom indicates the response details: "429 Too Many Requests 5 ms 336 B".

Home Workspaces API Network Upgrade

An update has been downloaded for Postman. Restart now to install the update. **Restart**

Get t Football POST Upl Upload Upload Upload Upload No Environment

Football Service / Get tractor in nearest event From Get not started event data Save

GET http://localhost:3000/users/ Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Type Bearer ...

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about [variables](#).

Token eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...

The authorization header will be automatically generated when you send the request. Learn more about [Bearer Token](#) authorization.

Body Cookies Headers (8) Test Results (0/1) 429 Too Many Requests 5 ms 336 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "statusCode": 429,
3   "message": "ThrottlerException: Too Many Requests"
4 }
```

Console Postbot Runner

مستندسازی API ها با Swagger :

ابتدا باید کتابخانه ی swagger را نصب کنیم:

```
npm i @nestjs/swagger
```

پس از نصب کتابخانه ی مربوطه باید swagger را در بخش main برنامه کانفیگ کنیم:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as passport from 'passport';
import * as session from 'express-session';
import { ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true
  }));
  app.use(
    session({
      secret: '1317abcd', // Change this to your desired secret key
      resave: false,
      saveUninitialized: false,
    }),
  );
  app.use(passport.initialize());
  app.use(passport.session());
  const config = new DocumentBuilder()
    .setTitle("Nestjs Learning")
    .setDescription("This is document for test APIs generated in Nestjs learning course")
    .setVersion('1.0')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('docs', app, document)

  await app.listen(3000);
}
bootstrap();
```

در این پخش عنوان، توضیح و URL سند تولید شده توسط Swagger را تعریف می کنیم برای این منظور ما سند را در URL تحت عنوان docs/ قرار می دهیم.

با اجرای این تنظیمات تمامی API ها به صورت خام به سند اضافه می گردد در گام بعدی به سراغ Dto ها رفته و از طریق دکوراتو های کتابخانه Swagger ورودی API ها را تنظیم می کنیم:

Login:

```
import { ApiProperty } from "@nestjs/swagger";
import { IsEmail, IsNotEmpty, IsString } from "class-validator";

export class LoginDto {
  @IsEmail()
  @ApiProperty({
    description: "Email of user",
    example: "ssuorena@gmail.com",
    format: "email"
  })
  email: string;

  @IsNotEmpty()
  @IsString()
  @ApiProperty({
    description: "Password of user",
    example: "12345",
  })
  password: string;
}
```

POST /auth/login

Parameters Try it out

No parameters

Request body required application/json

Example Value | Schema

```
{
  "email": "ssuorena@gmail.com",
  "password": "12345"
}
```

```
LoginDto {
  email* string($email)
  example: ssuorena@gmail.com
  Email of user

  password* string
  example: 12345
  Password of user
}
```

Register:

```
import { ApiProperty } from "@nestjs/swagger";
import { IsEmail, IsNotEmpty, IsNumber, IsOptional, IsString, Max, MaxLength,
MinLength } from "class-validator";

export class RegisterDto {
  @IsNumber()
  @IsNotEmpty()
  @ApiProperty({
    description: "ID of users in mongo db database",
    maximum: 10000,
    minimum: 100,
    example: 150
  })
  id: number
  @IsString()
  @IsNotEmpty()
  @IsEmail()
  @ApiProperty({
    description: "Email of user",
    example: "ssuorena@gmail.com",
    format: "email"
  })
  email: string;
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  @ApiProperty({
    description: "First name of user",
    example: "Suorena",
    minLength: 2
  })
  first_name: string;
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  @ApiProperty({
    description: "Last name of user",
    example: "Saeedi",
    minLength: 2
  })
  last_name: string;
  @IsNumber()
```

```

@IsOptional()
@Max(150)
@ApiProperty(
  {
    description: "Age of user",
    example:28,
    maximum:150
  }
)
age: number;
@IsNotEmpty()
@IsString()
@ApiProperty({
  description: "Password of user",
  example:"12345",

})
password: string;
}

```

POST /auth/register ^

Parameters Try it out

No parameters

Request body required application/json

Example Value | Schema

```

{
  "id": 150,
  "email": "ssuorena@gmail.com",
  "first_name": "Suorena",
  "last_name": "Saeedi",
  "age": 28,
  "password": "12345"
}

```

```

RegisterDto {
  id*      number
           maximum: 10000
           minimum: 100
           example: 150
           ID of users in mongo db database

  email*   string($email)
           example: ssuorena@gmail.com
           Email of user

  first_name* string
           example: Suorena
           minLength: 2
           First name of user

  last_name* string
           example: Saeedi
           minLength: 2
           Last name of user

  age*     number
           example: 28
           maximum: 150
           Age of user

  password* string
           example: 12345
           Password of user
}

```


Create User:

```
import { ApiProperty } from "@nestjs/swagger";
import { IsEmail, IsNotEmpty, IsNumber, IsOptional, IsString, Max, MaxLength,
MinLength } from "class-validator";

export class CreateUserDto {
  @IsNumber()
  @IsNotEmpty()
  @ApiProperty({
    description: "ID of users in mongo db database",
    maximum: 10000,
    minimum: 100,
    example: 150
  })
  id: number
  @IsString()
  @IsNotEmpty()
  @IsEmail()
  @ApiProperty({
    description: "Email of user",
    example: "ssuorena@gmail.com",
    format: "email"
  })
  email: string;
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  @ApiProperty({
    description: "First name of user",
    example: "Suorena",
    minLength: 2
  })
  first_name: string;
  @IsNotEmpty()
  @IsString()
  @MinLength(2)
  @ApiProperty({
    description: "Last name of user",
    example: "Saeedi",
    minLength: 2
  })
  last_name: string;
  @IsNumber()
```

```

@IsOptional()
@Max(150)
@ApiProperty(
  {
    description: "Age of user",
    example: 28,
    maximum: 150
  }
)
age: number;
@IsNotEmpty()
@IsString()
@ApiProperty({
  description: "Password of user",
  example: "12345",
})
password: string;
}

```

POST /users ^

Parameters Try it out

No parameters

Request body required application/json v

Example Value | Schema

```

{
  "id": 150,
  "email": "ssuorena@gmail.com",
  "first_name": "Suorena",
  "last_name": "Saeedi",
  "age": 28,
  "password": "12345"
}

```

CreateUserDto v {

id* number
maximum: 10000
minimum: 100
example: 150
ID of users in mongo db database

email* string(email)
example: ssuorena@gmail.com
Email of user

first_name* string
example: Suorena
minLength: 2
First name of user

last_name* string
example: Saeedi
minLength: 2
Last name of user

age* number
example: 28
maximum: 150
Age of user

password* string
example: 12345
Password of user

}

در ادامه به سراغ آپشن های دیگر API ها می رویم برای این که تعیین کنیم خروجی API ها به چه شکلی باشد از دکوراتور ApiResponse در کنترلر مانند زیر استفاده می کنیم:

```
@Get()
@UseGuards(JwtAuthGuard)
@ApiResponse({
  status:200,description:"Data send properly",type:CreateUserDto
})
async simpleGet() {
  return await this.usersService.findAll();
}
```

برای اینکه تعیین کنیم یک API نیاز به توکن شناسایی دارد از دکوراتور ApiBearerAuth استفاده می کنیم همچنین به کمک ApiHeader می توانیم مشخص کنیم که برای استفاده از این API باید از هدر استفاده گردد:

```
@Get()
@UseGuards(JwtAuthGuard)
@ApiHeader({
  name:"Authorization",
  description: "Send Authorization Token"
})
@ApiResponse({
  status:200,description:"Data send properly",type:CreateUserDto
})
@ApiBearerAuth()
async simpleGet() {
  return await this.usersService.findAll();
}
```

همچنین برخی API ها ممکن است در URL پارامتری را دریافت نمایند که این موارد را می توان به کمک دکوراتور ApiParam به صورت زیر تعریف کرد:

```
@ApiParam({
  name:"id",
  description:"ID of The user"
})
async find(@Param('id') id: number) {
  return await this.userService.findone(id);
}
```

در نهایت نیز انواع دکوراتورها را تعریف می کنیم و بخش کنترلر برای User و auth به صورت زیر می گردد:

User

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UseGuards } from
 '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { User } from 'src/NatinalteamModel';
import { JwtAuthGuard } from 'src/jwt-auth/jwt-auth.guard';
import { ApiBearerAuth, ApiBody, ApiHeader, ApiOperation, ApiParam, ApiResponse,
 ApiTags } from '@nestjs/swagger';

@Controller('users')
@ApiTags("Users")
export class UsersController {
  constructor(private readonly userService: UsersService) {}

  @Get('/:id')
  @UseGuards(JwtAuthGuard)
  @ApiHeader({
    name:"Authorization",
    description: "Send Authorization Token"
  })
  @ApiBearerAuth()
  @ApiResponse({
    status:200,description:"Data send properly",type:CreateUserDto
  })
  @ApiParam({
```

```

        name:"id",
        description:"ID of The user"
    })
    @ApiOperation({ summary: 'This API return user with certain id' })
    async find(@Param('id') id: number) {
        return await this.userService.findone(id);
    }

    @Post()
    // @UseGuards(JwtStrategy)
    @ApiOperation({ summary: 'This API Create user' })
    @ApiBody({
        type:CreateUserDto
    })
    async createUser(@Body() createUserDto: CreateUserDto):Promise<User> {
        return await this.userService.create(createUserDto);
    }

    @Get()
    @UseGuards(JwtAuthGuard)
    @ApiHeader({
        name:"Authorization",
        description: "Send Authorization Token"
    })
    @ApiResponse({
        status:200,description:"Data send properly",type:CreateUserDto
    })
    @ApiBearerAuth()
    @ApiOperation({ summary: 'This API return all users ' })
    async simpleGet() {
        return await this.userService.findAll();
    }

    @Delete('/:id')
    @ApiOperation({ summary: 'This API remove user with certain id' })
    remove(@Param('id') id: string) {
        return this.userService.remove(+id);
    }
}

```

Auth:

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from
 '@nestjs/common';
import { AuthService } from './auth.service';
import { RegisterDto } from './dto/register.dto';
import { LoginDto } from './dto/login.dto';
import { ApiOperation, ApiTags } from '@nestjs/swagger';

@Controller('auth')
@ApiTags("Authorization")
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @Post('register')
  @ApiOperation({ summary: 'This API create user' })
  register(@Body() registerDto: RegisterDto) {
    return this.authService.register(registerDto);
  }

  @Post('login')
  @ApiOperation({ summary: 'This API if for user login' })
  login(@Body() LoginDto: LoginDto) {
    return this.authService.login(LoginDto);
  }
}
```

در نهایت در صفحه ی بعد می توانید یک نمای کلی از صفحه ی Swagger را مشاهده نمایید:

Nestjs Learning 1.0 QA 1.0

This is document for test APIs generated in Nestjs learning course

[Authorize](#)

default

GET /

Users

GET /users/{id} This API return user with certain id

DELETE /users/{id} This API remove user with certain id

POST /users This API Create user

GET /users This API return all users

Authorization

POST /auth/register This API create user

Parameters

[Try it out](#)

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "id": 123,
  "email": "ssorena@gmail.com",
  "first_name": "Sorena",
  "last_name": "Sorena",
  "age": 20,
  "password": "12345"
}
```

Responses

Code	Description	Links
201		No links

POST /auth/login This API for user login

Schemas

CreateUserDto >

RegisterDto >

LoginDto >

کش کردن داده های یک API

در این بخش قصد داریم به کمک کتابخانه ی cach-manger داده های یک API را کش کنیم برای این منظور ابتدا باید این کتابخانه را نصب کنیم:

```
npm install @nestjs/cache-manager cache-manager
```

سپس باید CacheModule را در بخش ماژول های پروژه وارد کنیم:

app.module.ts

```
import { Module , NestModule, MiddlewareConsumer } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UsersModule } from './users/users.module';
import { MongooseModule } from '@nestjs/mongoose';
import { LoggerMiddleware } from './logger/logger.middleware';
import { AuthModule } from './auth/auth.module';
import { JwtModule } from '@nestjs/jwt';
import { CacheModule } from '@nestjs/cache-manager';

@Module({
  imports: [MongooseModule.forRoot(*****),
    CacheModule.register(),
    UsersModule,
    AuthModule],

  controllers: [AppController],

  providers: [AppService],
})

export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(LoggerMiddleware).forRoutes('*')
  }
}
```


سپس باید CacheModule را در بخش ماژول های API مدنظر نیز وارد کنیم در این پروژه قصد داریم که برای API که تمام کاربران را به ما می دهد کش بگذاریم برای این منظور به پوشه ی users می رویم:

users.module.ts

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { UserSchema, User } from 'src/NatinalteamModel';
import { CacheModule } from '@nestjs/cache-manager';

@Module({
  imports: [MongooseModule.forFeature([{ name: User.name, schema: UserSchema }]),
    CacheModule.register()],
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule {}
```

حال باید کش را در بخش کنترلر API مربوط به users اضافه کنیم:

users.controller.ts

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UseGuards,
  UseInterceptors, Inject } from '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { User } from 'src/NatinalteamModel';
import { JwtAuthGuard } from 'src/jwt-auth/jwt-auth.guard';
import { ApiBearerAuth, ApiBody, ApiHeader, ApiOperation, ApiParam, ApiResponse,
  ApiTags } from '@nestjs/swagger';
import { CACHE_MANAGER } from '@nestjs/cache-manager';
import { Cache } from 'cache-manager';

@Controller('users')
@ApiTags("Users")
export class UsersController {
  constructor(private readonly usersService: UsersService,
    @Inject(CACHE_MANAGER) private cacheManager: Cache
```

```

    ) {}

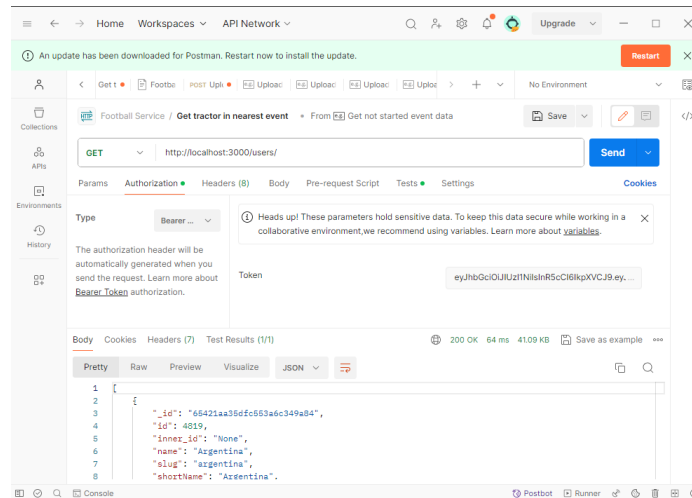
    @Get()
    @UseGuards(JwtAuthGuard)
    @ApiHeader({
      name: "Authorization",
      description: "Send Authorization Token"
    })
    @ApiResponse({
      status: 200, description: "Data send properly", type: CreateUserDto
    })
    @ApiBearerAuth()
    @ApiOperation({ summary: 'This API return all users ' })
    async simpleGet() {
      const allUsers = await this.cacheManager.get('users')
      console.log(allUsers)
      if (allUsers) {
        return "ok"
      }
      else{
        const allUsers = await this.userService.findAll();
        await this.cacheManager.set('users', allUsers, 100000)
        return allUsers
      }
    }
  }
}

```

در این بخش ابتدا ماژول `CACHE_MANAGER` را از `nestjs/cache-manager` و ماژول `Cache` را از `cache-manager` وارد می کنیم و `CACHE_MANAGER` را در `constructor` های کنترلر وارد می کنیم باید دقت شود که خود `type script` یک ماژول `Cache` دارد ولی در اینجا باید ماژول `Cache` را از کتابخانه ی مدنظر وارد کنیم.

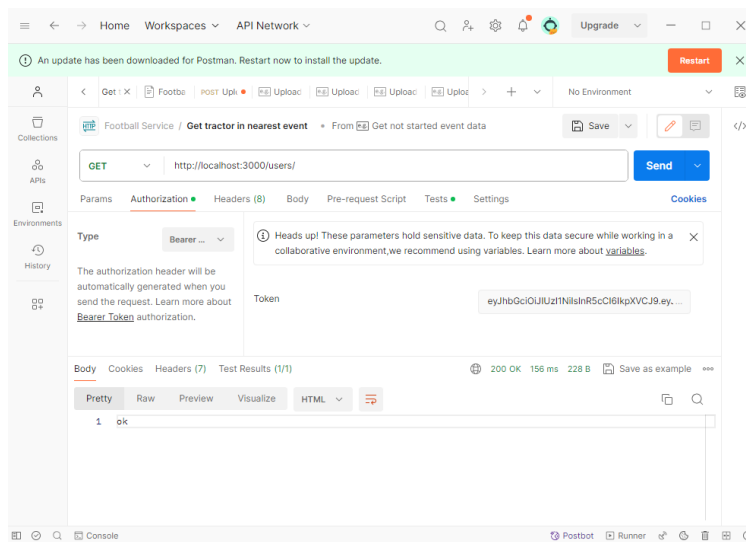
نهایتاً کنترلر `GET` مدنظر را با این منطق می نویسیم که اگر داده ای در حافظه ی کش با کلید `users` وجود داشت `Ok` برگرداند و اگر داده ای در کش وجود نداشت به سراغ تابع `findAll` در بخش سرویس ها رفته و مقدار خروجی آن را به مدت `100` ثانیه در حافظه ی کش ذخیره کند. باید به این نکته دقت شود که تمامی حروف کلید مربوط به حافظه ی کش باید کوچک باشند مثلاً اگر کلید را `'Users'` در نظر بگیریم چیزی در حافظه ی کش ذخیره نخواهد شد.

حال برای بار نخست این API را فراخوانی می کنیم:



```
undefined
[Nest] 9340 - 03/02/2024, 11:03:11 AM LOG [HTTP] Method :GET , baseUrl:/users , statusCode:200 , contentLength:41834 , responseTime:51.5789ms , userAgent:PostmanRuntime/7.36.3 , ip:::1,
```

همانطور که دیده می شود در مرحله ی اول خروجی API تمامی کاربرها می باشد و چیزی در حافظه ی کش وجود ندارد حال بار دیگری API را فراخوانی می کنیم:



```
  shortName: 'Belarus',
  nameCode: 'BLR',
  persian_name: 'None',
  flag: 'Flag/4743.png'
},
... 137 more items
]
[Nest] 9340 - 03/02/2024, 11:04:43 AM LOG [HTTP] Method :GET , baseUrl:/users , statusCode:200 , contentLength:2 , responseTime:150.4214ms , userAgent:PostmanRuntime/7.36.3 , ip:::1,
```

حال می توان دید که تمامی اطلاعاتی که در مرحله ی قبل فراخوانی API بازگشت در حافظه کش ذخیره شده (این اطلاعات در لاگ کنسول چاپ شده است) و همچنین خروجی API به ok تغییر پیدا کرده است. تا صد ثانیه ی دیگر نیز هربار که API فراخوانی شود فقط ok باز خواهد گشت چراکه داده ها در کش ذخیره شده اند.

ایجاد session برای کاربران

در ای بخش قصد داریم به معرفی جلسات در Nestjs بپردازیم. جلسات (sessions) در برنامه‌های وب به طور گسترده استفاده می‌شوند و ویژگی‌های مفیدی برای ایجاد اتصال پایدار و حفظ وضعیت کاربر ارائه می‌دهند. در زیر، به برخی از کاربردهای اصلی جلسات در برنامه‌های وب اشاره شده است:

1. احراز هویت و ورود کاربران:

جلسات برای احراز هویت کاربران و مدیریت ورود و خروج آنها استفاده می‌شوند. وقتی کاربر با موفقیت وارد سیستم می‌شود، یک جلسه برای آن ایجاد می‌شود که اطلاعات احراز هویتی (مانند شناسه کاربری یا نام کاربری) را در خود نگهداری می‌کند. این اطلاعات برای تشخیص هویت کاربر در هر درخواست استفاده می‌شوند.

2. حفظ وضعیت بین درخواست‌ها:

با استفاده از جلسات، می‌توانید وضعیت بین درخواست‌های مختلف یک کاربر را حفظ کنید. این اطلاعات می‌توانند از نوع متغیرهای جلسه (session variables) باشند که مانند یک سبد خرید، تنظیمات کاربری یا دیگر اطلاعات مورد نیاز برنامه‌ی شما هستند.

3. مدیریت دسترسی‌ها و سطوح دسترسی:

جلسات می‌توانند برای مدیریت دسترسی‌ها و سطوح دسترسی کاربران به بخش‌های مختلف برنامه مورد استفاده قرار بگیرند. اطلاعات مربوط به دسترسی‌ها می‌توانند در متغیرهای جلسه ذخیره شده و با توجه به آنها، برنامه می‌تواند تصمیم بگیرد که کاربر به چه منابعی دسترسی داشته باشد.

4. پیگیری فعالیت‌ها و آمارها:

با استفاده از جلسات، می‌توانید فعالیت‌ها و رفتار کاربران را پیگیری کنید و اطلاعات آماری مربوط به استفاده از برنامه را جمع‌آوری کنید. این اطلاعات می‌توانند برای بهبود عملکرد و تجربه کاربری برنامه مفید باشند.

5. بازیابی رمز عبور و بازنشانی کلمه عبور:

جلسات می‌توانند برای فرایند بازیابی رمز عبور و بازنشانی کلمه عبور کاربران استفاده شوند. با ارسال یک پیوند یا کد امنیتی به کاربران از طریق ایمیل یا پیام متنی، کاربران می‌توانند وارد حساب کاربری خود شوند و رمز عبور جدید تعیین کنند. با استفاده از جلسات، می‌توانید تجربه کاربری بهتری را در برنامه‌های وب خود فراهم کنید و از امکانات امنیتی و کاربردی آنها بهره‌مند شوید.

برای راه اندازی جلسات ابتدا باید کتابخانه های زیر را نصب کرد:

```
npm i express-session
```

```
npm i -D @types/express-session
```

حال باید در فایل main.ts برنامه session را وارد کرده و کانفیگ کنیم:

```
// Session configure
app.use(
  session({
    name: "Nestjs Learning Session",
    secret: 'session_secret',
    resave: false,
    saveUninitialized: false,
    cookie: {
      maxAge: 30*60*1000,
    }
  })
);
```

در این بخش تنظیم می کنیم که session به صورت یک کوکی با عنوان Nestjs Learning Session و با عمر 30 دقیقه برای هر کاربر ذخیره گردد و با کلید 'session_secret' هش گردد.

گزینه 'resave' در تنظیمات جلسه مشخص می کند که آیا جلسه مجدداً ذخیره شود یا خیر. وقتی که 'resave' به 'true' تنظیم شود، این به این معنی است که موقعیت های جلسه در هنگامی که درخواست هایی انجام می شود، دوباره ذخیره می شوند، حتی اگر تغییری در آنها ایجاد نشده باشد. این موضوع می تواند برای برخی متون از مرورگر منجر به مشکلاتی مانند ذخیره سازی اضافی و زیادی در پایگاه داده ها باشد. اگر 'resave' را به 'false' تنظیم کنید، جلسات فقط زمانی که تغییری در آنها ایجاد شود، مجدداً ذخیره می شوند. این کمک می کند تا بار اضافی بر روی پایگاه داده ها کاهش یابد و عملکرد سرور بهبود یابد. معمولاً تنظیم 'resave' را به 'false' ترجیح می دهند تا مشکلات ذخیره سازی اضافی حین اجرای برنامه بروز نکند. اما لازم به ذکر است که برخی از میان افزارها ممکن است نیاز به تنظیم 'resave' به 'true' داشته باشند. این نیاز به وابستگی به نوع میان افزار و رفتار برنامه است.

گزینه 'saveUninitialized' در تنظیمات جلسه مشخص می کند که آیا جلسه هایی که اطلاعاتی در آنها ذخیره نشده است، ذخیره شوند یا خیر. وقتی که 'saveUninitialized' به 'true' تنظیم شود، این به این معنی است که جلسه های جدیدی برای کاربرانی که اطلاعات جلسه ای مشخصی ندارند، ایجاد می شوند و به پایگاه داده ذخیره

می‌شوند. اگر `saveUninitialized` را به `false` تنظیم کنید، جلسه‌هایی که اطلاعاتی در آنها ذخیره نشده است، ذخیره نمی‌شوند. این موضوع به عنوان یک راه‌حل امنیتی مطرح می‌شود، زیرا کاربرانی که به صورت خودکار یک جلسه ایجاد می‌کنند اما اطلاعاتی در آنها ذخیره نشده است، جلسه‌های بی‌معنی ایجاد می‌کنند که باعث مصرف اضافی منابع سرور می‌شوند و امنیت را کاهش می‌دهند. معمولاً تنظیم `saveUninitialized` را به `false` ترجیح می‌دهند تا از ایجاد جلسه‌های بی‌معنی جلوگیری شود و منابع سرور بهینه‌تر استفاده شود. اما برای برخی از میان‌افزارها و سناریوها، ممکن است نیاز به ذخیره کردن جلسه‌های بی‌معنی باشد، به ویژه اگر بخواهید اطلاعاتی از کاربران بدون لاگین جمع‌آوری کنید.

پس از ایجاد این تنظیمات برای بررسی عملکرد session آن را بر روی یکی از کنترلرها تعریف می‌کنیم با این سناریو که زمانی که شخصی url این API را کال کرد یک session برای او ایجاد گردد و اطلاعات این session در یک کوکی ذخیره گردد همچنین در API بازگردانده شود و همچنین لاگ گرفته شود.

app.controller

```
import { Controller, Get, Session } from '@nestjs/common';
import { AppService } from '../app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  async getAuthsession(@Session() session: Record<string, any>){
    console.log(session)
    console.log(session.id)
    session.athenticated = true
    return session
  }
}
```




حال اگر این API را کال کنیم یک session برای ما ایجاد می‌گردد که اطلاعات آن به ما نمایش داده می‌شود و همچنین در بخش کوکی‌های مرورگر نیز یک کوکی با اطلاعات sessionId ذخیره می‌گردد.

```
localhost:3000
1 {
2   "cookie": {
3     "originalMaxAge": 1800000,
4     "expires": "2024-03-10T07:49:07.868Z",
5     "httpOnly": true,
6     "path": "/"
7   },
8   "authenticated": true
9 }
```

Cookies in use

Allowed Blocked

The following cookies were set when you viewed this page

-  Nestjs Learning Session
-  _xsrf
-  ajs_anonymous_id

Name	Nestjs Learning Session
Content	s%3AA5QUBmvEX7x98PhZzYD8Bj1HyTxiOy9d.FoYsP...
Domain	localhost
Path	/
Send for	Same-site connections only
Created	Sunday, March 10, 2024 at 10:48:04 AM
Expires	Sunday, March 10, 2024 at 11:18:04 AM

Block

Remove

Done

در ادامه قصد داریم تا در session ایجاد شده امکان شمارش تعداد دفعات دیدن صفحه ی نخست یا در واقع کال شدن API صفحه ی را داشته باشیم برای این منظور ابتدا باید یک فیچر جدید تحت عنوان visit به session خود اضافه کنیم برای این کار یک فایل جدید به صورت زیر ایجاد می کنیم:

src/ session.visit.ts

```
import { Session, SessionData } from 'express-session';

declare module 'express-session' {
  interface SessionData {
    visit?: number;
  }
}
```

حال کافی است در کنترلر این API تعریف کنیم که هر بار کال شد یک عدد به مقدار بازدیدهای کاربر افزوده گردد:

app.controller

```
import { Controller, Get, Session } from '@nestjs/common';
import { AppService } from '../app.service';

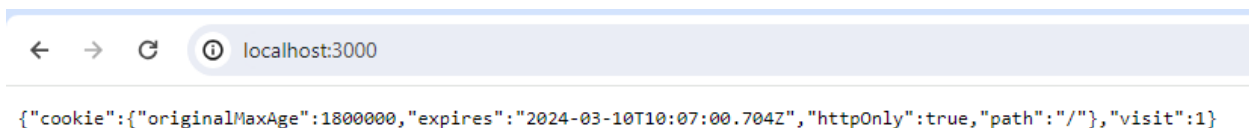
@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  async getAuthsession(@Session() session: Record<string,any>){
    session.visit = session.visit ? session.visit + 1 : 1;
    return session
  }
}
```

اکنون عملکرد آن را چک می کنیم برای این کار سه بار صفحه را رفرش می کنیم:



نکته اینجاست که در این حین اگر با مرورگر کروم هم به این API درخواست بدهیم یک session کاملاً جدید برای ما ایجاد می‌کند:



حال قصد داریم یک گام جلو تر رفته و زمانی که کاربر login کرد اطلاعات مربوط به نام کاربری و پسورد او را نیز در session مربوطه ذخیره سازی کنیم برای این منظور لازم است که ابتدا یک فیچر دیگر به session مربوطه تحت عنوان user اضافه کنیم. برای این کار یک فایل جدید به صورت زیر ایجاد می‌کنیم:

src/ Auth/ session.user.ts

```
import { Session, SessionData } from 'express-session';
import { LoginDto } from '../dto/login.dto';

declare module 'express-session' {
  interface SessionData {
    user?: LoginDto; // Assuming 'LoginDto' is your user model
  }
}
```

اکنون فقط باید هرزمان که کاربر ورود موفق به سایت داشت آیدی و پسورد او را در session ذخیره کنیم برای این کار به ورودی های کنترلر Login یک ورودی Request از نوع express اضافه می کنیم و آن را به صورت زیر تغییر می دهیم:

```
@Post('login')
@ApiOperation({ summary: 'This API if for user login' })
login(@Body() LoginDto: LoginDto, @Req() req: Request) {
  return this.authService.login(LoginDto, req);
}
```

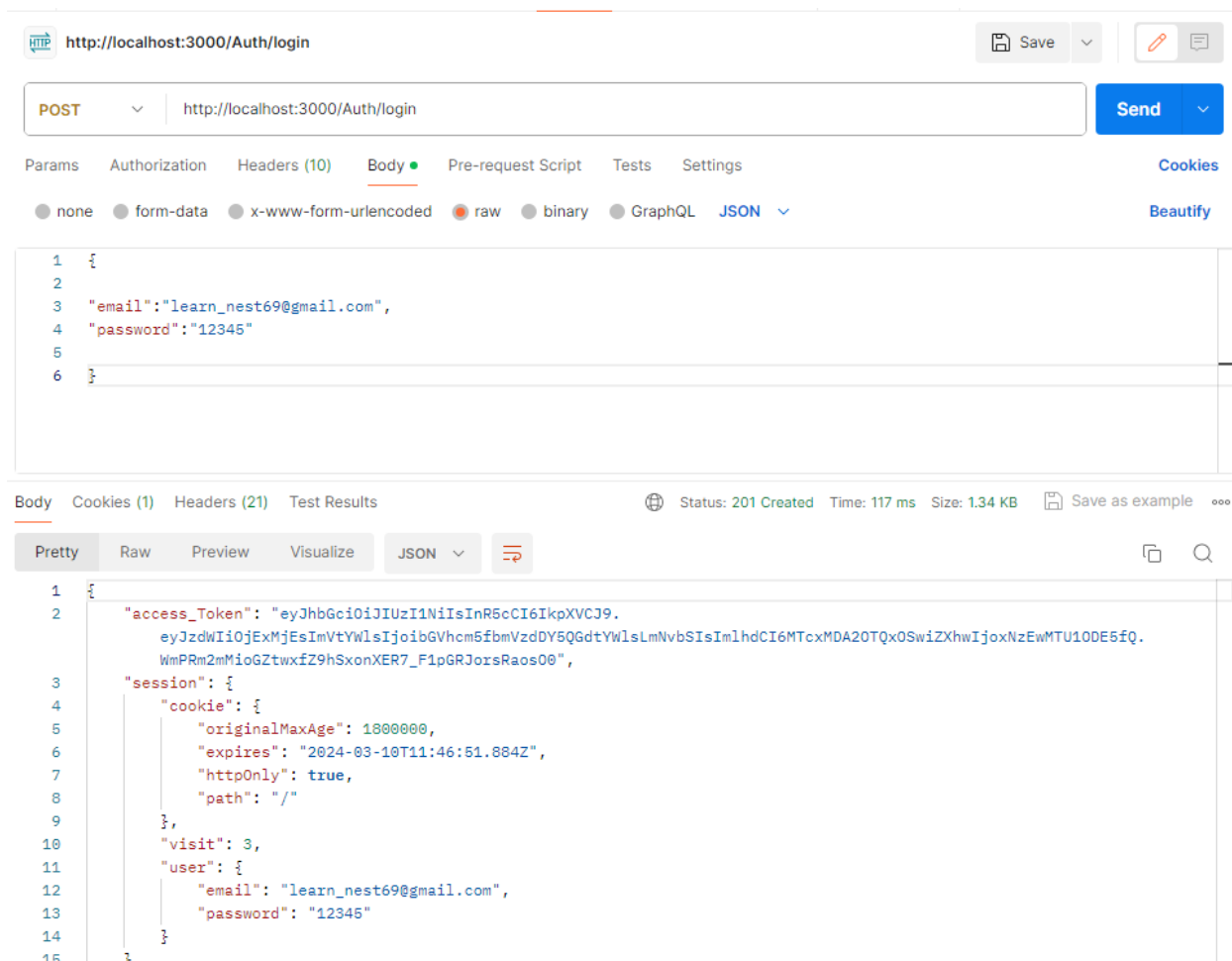
اکنون وارد بخش سرویس Auth شده و اگر چنانچه کاربر نام کاربری و پسورد صحیحی را وارد کرد این نام کاربری و پسورد را در session او ذخیره می کنیم:

```
async login(LoginDto: LoginDto , req: Request ) {
  const user = await this.userModel.findOne({email: LoginDto.email }).exec();
  if(!user){
    throw new HttpException("User not found",404);
  }

  const ispasswordMatch = await bcrypt.compare(LoginDto.password,user.password)
  if(!ispasswordMatch){
    throw new HttpException("Password is wrong",400);
  }

  const accessToken = this.JwtService.sign({
    sub: user.id,
    email: user.email
  })
  req.session.user = LoginDto //Save user information in session
  return {access_Token:accessToken,session:req.session};
}
```

حال عملکرد این کد را با Postman تست می کنیم. برای این منظور ابتدا سه بار با Postman به API صفحه ی نخست درخواست می دهیم و سپس به API مربوط به login ورود می کنیم خروجی سرویس login به شکل زیر خواهد بود:



می توان دید که اطلاعات کاربر شامل نام کاربری و رمز عبور او به این session افزوده شده است. به این شکل می توان این کاربر را شناسایی کرده و اطلاعات وی را جمع آوری کرد.

تعیین مجوز و نقش برای کاربران (Authorization)

یکی از نکات مهم در امنیت سایت تعیین نقش برای کاربران مختلف و تعیین دسترسی ها برای هریک از این کاربران می باشد. در Nest.js، Authorization یکی از مفاهیم اساسی در امنیت و مدیریت دسترسی به منابع در برنامه های وب است. اهمیت Authorization در برنامه های وب از دیدگاه امنیتی و مدیریتی بسیار بالاست. Authorization به معنای احراز هویت و کنترل دسترسی کاربران به منابع مختلف در یک برنامه وب است. با استفاده از Authorization، تصمیم گیری می شود که چه کاربرانی اجازه دسترسی به چه منابعی را دارند و چه کاربرانی باید از دسترسی به آن منابع محروم شوند. بسیاری از برنامه های وب دارای منابع حساسی هستند که نیاز به حفاظت دارند، مانند اطلاعات کاربران، داده های مالی، و غیره Authorization به شما امکان می دهد تا مطمئن شوید که تنها کاربران مجاز به دسترسی به این منابع هستند. همچنین در برخی موارد، ممکن است نیاز باشد که برخی از کاربران دسترسی به بخش های خاصی از برنامه را داشته باشند، در حالی که دیگران محدود شوند. به عنوان مثال، مدیران ممکن است به بخش مدیریت کامل دسترسی داشته باشند، در حالی که کاربران عادی ممکن است تنها به بخش های عمومی دسترسی داشته باشند. برای پیاده سازی Authorization باید ابتدا یک پروسه ی احراز هویت کاربر تعریف شود که در این گزارش در بخش "ایجاد یک سیستم شناسایی کاربر با nestjs" به تفصیل روند آن توضیح داده شده است. پس از ایجاد آن روند تعریف نقش ها به این صورت است که هر کاربر در دیتابیس علاوه بر تمام صفاتی که دارد یک صفت به عنوان Role نیز باید داشته باشد که تعیین کننده ی سطح دسترسی این کاربر است، زمانی که کاربر به دامنه ی ما login می کند در توکنی که برای دسترسی به او می دهیم نقش کاربر را نیز ارسال می کنیم و در گارد مربوط به Authorization تعیین می کنیم پیش از دسترسی دادن به کاربر نقش او مورد بررسی قرار گیرد و سپس مجوز استفاده از API مدنظر صادر گردد.

برای این منظور ابتدا به سراغ فایل UserModel.ts رفته و Role را به صفات کاربران اضافه می کنیم:

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import * as mongoose from 'mongoose';

@Schema({ collection: 'users' })
export class User extends Document {
  @Prop()
  id: number;

  @Prop()
  email: string;
```

```

@Prop()
first_name: string;

@Prop()
last_name: string;

@Prop()
age: string;

@Prop()
password: string;

@Prop()
roles: string;
}

export const UserSchema = SchemaFactory.createClass(User);

export const UserModel = mongoose.model<User>('User', UserSchema);

```

سپس در Dto های register.dto.ts و create-user.dto.ts پارامتر roles را به صورت زیر اضافه می کنیم:

```

import { ApiProperty } from "@nestjs/swagger";
import { IsEmail, IsNotEmpty, IsNumber, IsOptional, IsString, Max, MaxLength, MinLength } from "class-validator";

export class CreateUserDto {
  @IsNumber()
  @IsNotEmpty()
  @ApiProperty({
    description: "ID of users in mongo db database",
    maximum: 10000,
    minimum: 100,
    example: 150
  })
  id: number;
  @IsString()
  @IsNotEmpty()
  @IsEmail()
  @ApiProperty({
    description: "Email of user",
    example: "ssuorena@gmail.com",
  })
  email: string;
}

```

```

        format:"email"
    })
    email: string;
    @IsNotEmpty()
    @IsString()
    @MinLength(2)
    @ApiProperty({
        description: "First name of user",
        example:"Suorena",
        minLength:2
    })
    first_name: string;
    @IsNotEmpty()
    @IsString()
    @MinLength(2)
    @ApiProperty({
        description: "Last name of user",
        example:"Saeedi",
        minLength:2
    })
    last_name: string;
    @IsNumber()
    @IsOptional()
    @Max(150)
    @ApiProperty(
        {
            description: "Age of user",
            example:28,
            maximum:150
        }
    )
    age: number;
    @IsNotEmpty()
    @IsString()
    @ApiProperty({
        description: "Password of user",
        example:"12345",
    })
    password: string;
    @IsNotEmpty()
    @IsString()
    @ApiProperty({
        description: "Role of user",
        example:"ADMIN",
    })

```

```

    })
    roles: string;
}

```

اکنون نوبت تعریف نقش ها است برای این گزارش صرفاً دو نقش Admin و User را در نظر میگیریم. لازمه به ذکر است که در پروژه های عملیاتی تعداد نقش ها بسیار بیشتر بوده و نقش ها جدول خاص خود را دارند و به صورت صفت مستقیماً در جدول کاربران جای نمی گیرند.

role.enum.ts

```

export enum Role {
  ADMIN = 'ADMIN',
  USER = 'USER',
}

```

حال باید دکوراتور Roles را به گونه ای تعریف کنیم که بتواند مجموعه ای از نقش ها را بگیرد و از آن در کنترلرهای مدنظرمان استفاده کنیم:

roles.decorator.ts

```

import { SetMetadata } from '@nestjs/common';
import { Role } from './role.enum';

export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);

```

در گام بعدی باید در کد های بخش احراز هویت تغییراتی را اعمال کنیم در این بخش قصد داریم تا در توکن احراز هویت نقش کاربر را نیز ارسال کنیم برای این منظور باید در سرویس login نقش کاربر را نیز در payload اسایی کاربر قرار دهیم بنابراین سرویس login را به صورت زیر تعریف می کنیم:

```

async login(LoginDto: LoginDto , req: Request ) {
  const user = await this.userModel.findOne({email: LoginDto.email }).exec();

  if(!user){
    throw new HttpException("User not found",404);
  }
}

```

```

const ispasswordMatch = await bcrypt.compare(LoginDto.password,user.password)

if(!ispasswordMatch){
  throw new HttpException("Password is wrong",400);
}

const accessToken = this.JwtService.sign({
  sub: user.id,
  email: user.email,
  roles: user.roles,
})
console.log(user)

req.session.user = LoginDto //Save user information in session

return {access_Token:accessToken,session:req.session};
}

```

و همچنین payload را در JwtStrategy نیز تعریف می کنیم:

jwt.strategy.ts

```

import { Injectable } from "@nestjs/common";
import { PassportStrategy } from "@nestjs/passport";
import { Strategy, ExtractJwt } from 'passport-jwt';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: 'secret2',
    });
  }

  async validate(payload: any) {

    return {
      id: payload.sub,
      email: payload.email,

```



```

        roles:payload.roles
    };
}
}

```

در نهایت نیز باید گارد مناسب را برای نقش ها بنویسیم به این صورت که اگر هیچ گاردی برای API مدنظر وجود نداشت دسترسی بدهد و اگر وجود داشت با نقش موجود در توکن بررسی کرده و اگر نقش موجود در توکن با یکی از نقش های موجود در گارد همخوانی داشت اجازه ی دسترسی بدهد:

role.guard.ts

```

import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Role } from './role.enum';
import { ROLES_KEY } from './roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles?.includes(role));
  }
}

```

در این کد **reflector** نشان دهنده ی عبارت نوشته شده درون گارد در کنترلر است و **context** نیز بیانگر محتوای توکن موجود در هدر می باشد. در نهایت باید در کنترلر **USER** دسترسی ها را تعریف کنیم سناریو مدنظر این است فقط کاربر **Admin** بتواند تمامی کاربران را دریافت کند و کاربر **Admin** و **User** بتوانند یک کاربر را براساس شناسه دریافت نمایند با این سناریو کنترلر کد ما به صورت زیر خواهد بود:

users.controller.ts

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UseGuards,
UseInterceptors, Inject } from '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { User } from 'src/UserModel';
import { JwtAuthGuard } from 'src/jwt-auth/jwt-auth.guard';
import { ApiBearerAuth, ApiBody, ApiHeader, ApiOperation, ApiParam, ApiResponse,
ApiTags } from '@nestjs/swagger';
import { CACHE_MANAGER } from '@nestjs/cache-manager';
import { Cache } from 'cache-manager';
import { ThrottlerGuard } from '@nestjs/throttler';
import { Roles } from 'src/authorization/roles.decorator';
import { Role } from 'src/authorization/role.enum';
import { RolesGuard } from 'src/authorization/role.guard';

@Controller('users')
@ApiTags("Users")
export class UsersController {
  constructor(private readonly usersService: UsersService,
    @Inject(CACHE_MANAGER) private cacheManager: Cache
  ) {}

  @Get('/:id')
  // @Roles(Role.ADMIN) // attaching metadata
  @Roles(Role.USER, Role.ADMIN) // attaching metadata
  @UseGuards(JwtAuthGuard, RolesGuard)
  @ApiHeader({
    name: "Authorization",
    description: "Send Authorization Token"
  })
  @ApiBearerAuth()
  @ApiResponse({
    status: 200, description: "Data send properly", type: CreateUserDto
  })
  @ApiParam({
    name: "id",
    description: "ID of The user"
  })
  @ApiOperation({ summary: 'This API return user with certain id' })
  async find(@Param('id') id: number) {
```

```

        return await this.userService.findone(id);
    }

    @Post()
    // @UseGuards(JwtStrategy)
    @ApiOperation({ summary: 'This API Create user' })
    @ApiBody({
        type: CreateUserDto
    })
    async createUser(@Body() createUserDto: CreateUserDto): Promise<User> {
        return await this.userService.create(createUserDto);
    }

    // now in milliseconds (1 minute === 60000)
    @Get()
    @Roles(Role.ADMIN) // attaching metadata
    @UseGuards(ThrottlerGuard)
    @UseGuards(JwtAuthGuard, RolesGuard)
    @ApiHeader({
        name: "Authorization",
        description: "Send Authorization Token"
    })
    @ApiResponse({
        status: 200, description: "Data send properly", type: CreateUserDto
    })
    @ApiBearerAuth()
    @ApiOperation({ summary: 'This API return all users ' })
    async simpleGet() {
        // Check if c has a value
        const allUsers = await this.cacheManager.get('users')
        console.log(allUsers)
        if (allUsers) {
            return allUsers
        }
        else {
            const allUsers = await this.userService.findAll();
            await this.cacheManager.set('users', allUsers, 100000)
            return allUsers
        }
    }
}

```

```

@Delete('/:id')
@ApiOperation({ summary: 'This API remove user with certain id' })
remove(@Param('id') id: string) {
    return this.userService.remove(+id);
}
}

```

RolesGuard بیانگر این است که گارد نقش ها باید بر روی این API اجرا گردد و در دکوراتور Roles@ نقش های مجاز برای این API را تعریف می کنیم در ادامه یک کاربر با دسترسی Admin و یک کاربر با دسترسی User از مسیر ریجستر ایجاد کرده و عملکرد این گارد را چک می کنیم:

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:3000/Auth/login
- Body:**

```

{
  "email": "learn_nest_authorize@gmail.com",
  "password": "12345"
}

```
- Status:** 201 Created
- Time:** 150 ms
- Size:** 1.54 KB
- Response Body (JSON):**

```

{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJlZS5lcjlbWpCI6Imx1YXJuX25lc3RfYXV0b3JpemVAZ221haWwY29tIiwicm9sZXMiOiJBRE1JTjIiIsImh0dCI6MTcxMDU5ODY5NSwiZXhwIjoxNzEwNjg1MDk1fQ.vFAE0mxs5-48J2oRQq13iHoRWVn0HyK90W8gUOWC47Y",
  "session": {
    "cookie": {
      "originalMaxAge": 1800000,
      "expires": "2024-03-16T14:48:15.284Z",
      "httpOnly": true,
      "path": "/"
    },
    "user": {
      "email": "learn_nest_authorize@gmail.com",

```

این کاربر دارای دسترسی Admin است حال با توکن آن قصد گرفتن کل کاربران را داریم:

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://localhost:3000/users/
- Authorization:** Bearer To... (Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...
- Status:** 200 OK
- Time:** 73 ms
- Size:** 43.18 KB
- Response Body (JSON):**

```
[{"_id": "65421aa35dfc553a6c349a84", "id": 4819, "inner_id": "None", "name": "Argentina", "slug": "argentina", "shortName": "Argentina", "nameCode": "ARG", "persian_name": "None", "flag": "flag/4819.png"}]
```

می توان دید که موفق شدیم کل کاربران را دریافت کنیم حال با یک کاربر User وارد می شویم:

می توان دید که سیستم به درستی به کاربر User امکان دریافت تمامی کاربران را نمی دهد و عملکرد مد نظر ما اجرا می گردد.

می توان دید که سیستم به درستی به کاربر User امکان دریافت تمامی کاربران را نمی دهد و عملکرد مد نظر ما اجرا می گردد.

اقدامات امنیتی

در این بخش قصد داریم مختصراً برخی از اقدامات در راستای بهبود امنیت سایت را معرفی کرده و پیاده سازی اولیه ی آن ها را مورد بررسی قرار دهیم. اجرای این اقدامات می تواند تا حد خوبی امکان نفوذ به سایت را کاهش دهد و عمده‌تا پیاده سازی آن ها نیز چندان زمان بر نخواهد بود.

Helmet

کتابخانه Helmet در NestJS وظیفه افزودن لایه‌ی امنیتی به برنامه‌ی شما را دارد. این کتابخانه از middleware می‌باشد که به عنوان یک لایه واسطه در پرچم و درخواست‌های HTTP اعمال می‌شود. Helmet در واقع یک wrapper است برای کلیه‌ی فراهم‌کنندگان middleware های امنیتی در Node.js. برای استفاده از Helmet ابتدا کتابخانه ی زیر را نصب کنید:

```
npm i --save helmet
```

پیش از پیاده سازی helmet اگر نگاهی به هدرهای پاسخ کد خود بیندازیم به‌صورت زیر خواهند بود:

▼ Response Headers	<input type="checkbox"/> Raw
Connection:	keep-alive
Content-Length:	121
Content-Type:	application/json; charset=utf-8
Date:	Tue, 05 Mar 2024 06:36:22 GMT
Etag:	W/"79-RJWG7PrDRBI4Nje/MUQcGWy6gfl"
Keep-Alive:	timeout=5
Set-Cookie:	Nestjs Learning Session=s%3ANA5_I7SPrciF1ON41lvqSRVTOX589ajs.XD6PA6bf%2FUVWAitxe GlgTPKGIq167ZH91rDdZEKrlbY; Path=/; Expires=Tue, 05 Mar 2024 07:06:22 GMT; HttpOnly
X-Powered-By:	Express

می توان دید که تقریباً هیچ تنظیمات امنیتی در هدر ها وجود ندارد یکی از وظایف اصلی Helmet اعمال تنظیمات هدر های امنیتی می باشد. حال قصد داریم حالت دیفالت کتابخانه ی Helmet را فعال کنیم برای این کار کافی است در فایل main.ts کتابخانه را وارد کرده و آن را فعال کنیم:

```
import helmet from 'helmet';  
app.use(helmet())
```

با ایجاد مقدار اولیه ی این کتابخانه در کد اکنون می توانیم مقادیر دیفالت هدرهای امنیتی را مشاهده کنیم اگر مجددا هدرهای پاسخ را بررسی کنیم می بینیم:

```
Connection: keep-alive
Content-Length: 121
Content-Security-Policy: default-src 'self';base-uri 'self';font-src 'self' https: data;;form-action 'self';frame-ancestors 'self';img-src 'self' data;;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Content-Type: application/json; charset=utf-8
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Resource-Policy: same-origin
Date: Tue, 05 Mar 2024 07:21:47 GMT
Etag: W/"79-a31TB9LlOhJpgIb2NIXqzMvkAg"
Keep-Alive: timeout=5
Origin-Agent-Cluster: ?1
Referrer-Policy: no-referrer
Set-Cookie: Nestjs Learning Session=s%3AO-WI2mRN14K1BsQGVMQqim_rSpP0f6en.www1v9vlqEQX2wLA2MtvEfl87wEoC HLFaK4hgTFanZo; Path=/; Expires=Tue, 05 Mar 2024 07:51:47 GMT; HttpOnly
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Content-Type-Options: nosniff
X-Dns-Prefetch-Control: off
X-Download-Options: noopen
X-Frame-Options: SAMEORIGIN
X-Permitted-Cross-Domain-Policies: none
X-Xss-Protection: 0
```

در ادامه برخی از این تنظیمات هدرهای امنیتی را مختصرا معرفی می کنیم

تنظیم هدرهای امنیتی:

Content Security Policy (CSP): CSP – به شما اجازه می دهد تا سیاست های امنیتی برای منابع مختلف (مانند اسکریپت ها، استایل ها، تصاویر و غیره) در صفحات خود تعیین کنید. این سیاست ها مشخص می کنند که کدام منابع مجاز هستند و کدام منابع غیرمجاز هستند.

X-Content-Type-Options – **X-Content-Type-Options**: یکی از هدرهای امنیتی است که توسط **Helmet** در **NestJS** قابل تنظیم است. این هدر به مرورگرها می گوید که نوع **MIME** (**Multipurpose Internet Mail Extensions**) محتوا را از طریق **MIME-sniffing** تعیین نکنند و به طور صریح از نوع محتوای تعیین شده استفاده کنند. زمانی که مرورگر **MIME-sniffing** را فعال دارد، ممکن

است در مواردی که سرور ناهمخوانی در تعیین نوع MIME دارد، مرورگر تلاش کند تا نوع MIME را به صورت خودکار تشخیص دهد. این عملیات ممکن است در برخی موارد باعث امنیت پایینی شود، زیرا ممکن است حملاتی مانند XSS (Cross-Site Scripting) از طریق این نوع MIME-sniffing انجام شود. با ارسال هدر X-Content-Type-Options با مقدار `nosniff` به مرورگر، ما به آن می‌گوییم که نوع MIME را از سرور بپذیرد و از MIME-sniffing خودداری کند. این به مرورگر اجازه می‌دهد که مطمئن شود که نوع محتوایی که از سرور دریافت می‌کند، دقیقاً همان محتوایی است که سرور تعیین کرده است. به طور کلی، استفاده از هدر X-Content-Type-Options با مقدار `nosniff` در NestJS با استفاده از کتابخانه Helmet می‌تواند به افزایش امنیت برنامه‌ی شما کمک کند و از برخی از حملات امنیتی مانند XSS جلوگیری کند.

- X-DNS-Prefetch-Control: هدر X-DNS-Prefetch-Control یکی از هدرهای امنیتی است که توسط Helmet در NestJS قابل تنظیم است. این هدر کنترل می‌کند که مرورگر آیا باید DNS prefetching را انجام دهد یا خیر. DNS prefetching فرآیندی است که مرورگرها از طریق آن می‌توانند DNS را به طور پیش فرض برای منابع جهانی مانند تصاویر، اسکریپت‌ها، فونت‌ها و غیره prefetch کنند. این عملیات prefetching به مرورگر کمک می‌کند که قبل از زمانی که نیاز به آن‌ها باشد، آدرس IP مربوط به منابع را بدست آورد. مقادیر ممکن برای هدر X-DNS-Prefetch-Control عبارتند از:

1. off: این مقدار به مرورگر می‌گوید که DNS prefetching را غیرفعال کند، به این معنی که مرورگر نباید سعی کند آدرس‌های IP منابع را پیش از نیاز prefetch کند.
2. on: این مقدار به مرورگر می‌گوید که DNS prefetching را فعال کند، به این معنی که مرورگر مجاز است آدرس‌های IP منابع را پیش از نیاز prefetch کند.

با تنظیم هدر X-DNS-Prefetch-Control، شما می‌توانید کنترل دقیق‌تری بر روی عملیات DNS prefetching در مرورگرهای کاربران خود داشته باشید. این کمک می‌کند تا از مصرف اضافی منابع شبکه جلوگیری کرده و بهینه‌سازی عملکرد وبسایت شما را بهبود بخشید.

- X-Frame-Options: هدر X-Frame-Options یکی دیگر از هدرهای امنیتی است که توسط Helmet در NestJS قابل تنظیم است. این هدر به مرورگرها می‌گوید که آیا محتوای وبسایت می‌تواند در یک یا چند frame یا iframe دیگر نمایش داده شود یا خیر. این هدر به طور مستقیم محافظتی در برابر حملات clickjacking فراهم می‌کند. Clickjacking یا UI redress attack یک نوع حمله است که هنگامی که حمله‌کننده یک صفحه وب را در یک frame یا iframe شفاف مخفی کند و محتوایی را که قابلیت تعامل با

کاربر را دارد (مثل دکمه‌هایی که کاربر می‌تواند کلیک کند) روی آن بگذارد، اجازه می‌دهد که اعمالی که کاربر انجام می‌دهد، به صورت ناخواسته بر روی صفحه مخفی شده انجام شود. با استفاده از هدر `X-Frame-Options`، می‌توانید کنترل کنید که آیا محتوای وبسایت شما در یک `frame` یا `iframe` دیگر می‌تواند نمایش داده شود یا نه. این هدر دارای سه مقدار اصلی است:

1. `DENY`: این مقدار ممنوعیت نمایش محتوا را در هر `frame` یا `iframe` دیگر اعمال می‌کند.
2. `SAMEORIGIN`: این مقدار فقط به `frame` یا `iframe` هایی اجازه نمایش محتوا می‌دهد که از همان دامنه درخواست می‌شود.
3. `ALLOW-FROM uri`: این مقدار به `frame` یا `iframe` هایی اجازه نمایش محتوا را می‌دهد که از آدرس `URI` مشخص شده در پارامتر `uri` درخواست شده باشند.

با تنظیم هدر `X-Frame-Options` به مقادیر مناسب، شما می‌توانید از حملات `clickjacking` جلوگیری کنید و امنیت برنامه‌ی خود را تقویت کنید.

- `X-Permitted-Cross-Domain-Policies`: هدر `X-Permitted-Cross-Domain-Policies` یک هدر امنیتی است که توسط `Helmet` در `NestJS` قابل تنظیم است. این هدر برای کنترل سیاست‌های ارتباط متن با متن (مانند `XML`، `HTML` و غیره) در یک وبسایت از طریق فایل‌های `Cross-Domain Policy` مورد استفاده قرار می‌گیرد. فایل‌های `Cross-Domain Policy` به وبسایت‌ها اجازه می‌دهند که سیاست‌هایی را تعیین کنند که مشخص می‌کند که چه ارتباطات `cross-domain` (از یک دامنه به دامنه دیگر) مجاز است. این فایل‌ها معمولاً در فرمت `XML` بوده و برای محدود کردن دسترسی به منابع وب از دامنه‌های دیگر استفاده می‌شوند. هدر `X-Permitted-Cross-Domain-Policies` به وبسرور اجازه می‌دهد تا سیاست‌های `Cross-Domain Policy` خود را تعیین کند. این هدر می‌تواند چندین مقدار مختلف داشته باشد، از جمله:

1. `none`: این مقدار بیان می‌کند که هیچ یک از فایل‌های `Cross-Domain Policy` مجاز نیستند.
2. `master-only`: این مقدار نشان می‌دهد که فقط فایل `Cross-Domain Policy` اصلی برای سیاست‌های `Cross-Domain Policy` مجاز است. فایل‌های دیگر `Cross-Domain Policy` توسط مرورگر نادیده گرفته می‌شوند.
3. `by-content-type`: این مقدار به مرورگر اجازه می‌دهد که فایل‌های `Cross-Domain Policy` را بر اساس نوع `MIME` آنها (مانند `text/x-cross-domain-policy`) شناسایی کند.
4. `all`: این مقدار به مرورگر اجازه می‌دهد تا هر نوع فایل `Cross-Domain Policy` را مجاز بداند.

با استفاده از هدر `X-Permitted-Cross-Domain-Policies`، می‌توانید به مرورگرها بگویید که کدام فایل‌های `Cross-Domain Policy` مجاز هستند و کدام نه. این کمک می‌کند تا به امنیت برنامه‌ی شما از حملاتی مانند از دور آوردن محدودیت‌های دسترسی به منابع وب از دامنه‌های دیگر، کمک کند.

Referrer-Policy - هدر `Referrer-Policy` یکی دیگر از هدرهای امنیتی است که توسط `Helmet` در `NestJS` قابل تنظیم است. این هدر کنترل می‌کند که مرورگر چه اطلاعاتی را به عنوان `referrer` (ارجاع دهنده) با درخواست‌ها ارسال کند و از طریق هدرهای `HTTP` انتقال دهد.

مقادیر مختلف هدر `Referrer-Policy` شامل:

1. `no-referrer`: مرورگر هیچ اطلاعات `referrer` را با درخواست ارسال نمی‌کند.
2. `no-referrer-when-downgrade`: این مقدار به مرورگر می‌گوید که `referrer` را برای ارتباط‌های امن ارسال کند، اما برای ارتباط‌های غیرامن، `referrer` را ارسال نکند.
3. `origin`: مرورگر فقط نام دامنه (`origin`) به عنوان `referrer` ارسال می‌کند.
4. `origin-when-cross-origin`: مرورگر برای ارتباط‌های داخلی فقط نام دامنه (`origin`) را به عنوان `referrer` ارسال می‌کند، اما برای ارتباط‌های بین دامنه‌ای، تمام `URL` را به عنوان `referrer` ارسال می‌کند.
5. `same-origin`: مرورگر فقط برای ارتباط‌های داخلی (از همان دامنه) `referrer` را ارسال می‌کند.
6. `strict-origin`: مرورگر فقط برای ارتباط‌های امن (`HTTPS`) نام دامنه (`origin`) را به عنوان `referrer` ارسال می‌کند، اما برای ارتباط‌های غیرامن (`HTTP`) `referrer` را ارسال نمی‌کند.
7. `strict-origin-when-cross-origin`: مرورگر برای ارتباط‌های امن (`HTTPS`) فقط نام دامنه (`origin`) را به عنوان `referrer` ارسال می‌کند، اما برای ارتباط‌های غیرامن (`HTTP`) تنها نام دامنه را به عنوان `referrer` ارسال می‌کند.
8. `unsafe-url`: مرورگر تمام `URL` را به عنوان `referrer` ارسال می‌کند، حتی اگر دامنه‌ی مقصد `HTTPS` باشد.

با تنظیم مقدار مناسب برای هدر `Referrer-Policy`، شما می‌توانید کنترل دقیق‌تری بر روی اطلاعاتی که مرورگر به عنوان `referrer` ارسال می‌کند، داشته باشید. این کمک می‌کند تا امنیت برنامه‌ی شما از حملاتی مانند افشای اطلاعات حساس کاربران (مانند اطلاعات ورود به سیستم) حفظ شود.

X-Xss-protection - هدر `X-Xss-Protection` یکی دیگر از هدرهای امنیتی است که توسط `Helmet` در `NestJS` قابل تنظیم است. این هدر به مرورگر اطلاع می‌دهد که آیا باید فیلترینگ و حذف حملات `XSS` (`Cross-Site Scripting`) را انجام دهد یا خیر. حملات `XSS` زمانی رخ می‌دهند که مهاجمان کد اسکرپت را

در صفحات وب قرار داده و کاربران را متقاعد می‌کنند که اجرای آن را انجام دهند. این حملات می‌توانند به سرقت اطلاعات کاربران، دزدیدن کوکی‌ها، و تغییر محتوای صفحات منجر شوند. هدر `X-XSS-Protection` به مرورگر اجازه می‌دهد تا یک فیلتر XSS را اعمال کند. این فیلتر به طور خودکار کدهای اسکریپتی را که در صفحه درخواست شده‌اند، بررسی می‌کند و اگر شناسایی کند که اسکریپتی مخرب است، آن را حذف می‌کند یا اجازه اجرای آن را نمی‌دهد. مقادیر ممکن برای هدر `X-XSS-Protection` عبارتند از:

1. 0: غیرفعال کردن فیلتر XSS در مرورگر.
 2. 1: فعال کردن فیلتر XSS در مرورگر با رفتار پیش‌فرض. اگر مرورگر تشخیص دهد که یک حمله XSS در حال انجام است، آن را بلافاصله مسدود می‌کند.
 3. 1; mode=block: فعال کردن فیلتر XSS در مرورگر با رفتار بلاک کردن. اگر مرورگر تشخیص دهد که یک حمله XSS در حال انجام است، صفحه را به صورت کامل مسدود می‌کند.
- با اعمال هدر `X-XSS-Protection` به مقدار مناسب، شما می‌توانید از حملات XSS در برنامه‌ی خود جلوگیری کنید و امنیت کاربران را تضمین کنید.
- برای ایجاد هرکدام از این تغییرات از حالت دیفالت باید این‌ها را در فایل `main.ts` و در تابع `helmet` ایجاد کنیم به عنوان مثال در کد زیر هدر `Referrer-Policy` را از حالت دیفالت (`no-referrer`) به `no-referrer-when-downgrade` تغییر می‌دهیم و هدر `X-Frame-Options` را نیز از مقدار دیفالت (`SAMEORIGIN`) به `DENY` تغییر می‌دهیم:

```
app.use(helmet({
  xFrameOptions: {
    action: "deny"
  },
  referrerPolicy: {
    policy: "no-referrer-when-downgrade"
  }
}))
```

در ادامه می‌توان دید که تنظیمات در هدرها اعمال شده‌اند:

```
Connection: keep-alive
Content-Security-Policy: default-src 'self';base-uri 'self';font-src 'self' https: data;;form-action 'self';frame-ancestors 'self';img-src 'self' data;;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Resource-Policy: same-origin
Date: Sat, 09 Mar 2024 08:43:55 GMT
Etag: W/"a36a-jgZx9ypaMlaiR1SAD6Od5XxGEeY"
Keep-Alive: timeout=5
Origin-Agent-Cluster: ?1
Referrer-Policy: no-referrer-when-downgrade
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Content-Type-Options: nosniff
X-Dns-Prefetch-Control: off
X-Download-Options: noopen
X-Frame-Options: DENY
X-Permitted-Cross-Domain-Policies: none
X-Ratelimit-Limit: 10
X-Ratelimit-Remaining: 9
X-Ratelimit-Reset: 300
X-Xss-Protection: 0
```

به طور کلی، استفاده از کتابخانه **Helmet** در **NestJS** به شما کمک می‌کند تا برنامه‌ی خود را از نظر امنیتی تقویت کنید و از برخی از آسیب‌پذیری‌های معمول در وب‌سایت‌ها و برنامه‌های وب جلوگیری کنید. برای اطلاعات بیشتر راجع به این کتابخانه می‌توانید از این [لینک](#) استفاده کنید.

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) یک مکانیزم امنیتی است که در مرورگرهای وب پیاده‌سازی می‌شود و اجازه می‌دهد تا یک وب‌سایت به صورت امنیتی منابع خود را با سایت‌های دیگر به اشتراک بگذارد. در محیط وب، هر اصل (origin) توسط یک دامنه، پروتکل، و پورت مشخص می‌شود. اگر یک صفحه وب از یک منبع با اصلی متفاوت (یعنی دامنه، پروتکل، یا پورت متفاوت) درخواست کند، این درخواست به عنوان یک درخواست cross-origin (از دامنه دیگر) شناخته می‌شود.

قبل از CORS، مرورگرها توسط سیاست (SOP) Same-Origin Policy، جلوی درخواست‌های cross-origin را می‌گرفتند و اجازه دسترسی به منابع محلی در یک صفحه وب را به صورت پیش‌فرض می‌دادند. اما با افزایش پیچیدگی برنامه‌های وب و نیاز به اشتراک گذاری منابع با دامنه‌های دیگر، نیاز به یک راه حل برای مدیریت امنیتی این اشتراک گذاری بوجود آمد.

با استفاده از CORS، سرور می‌تواند به مرورگر اجازه دهد تا ببیند آیا یک درخواست cross-origin می‌تواند اجرا شود یا خیر. این کنترل به وب‌سرور اجازه می‌دهد تا سیاست‌های خود را برای اشتراک گذاری منابع با دامنه‌های دیگر مشخص کند. سپس مرورگرها این سیاست‌ها را اجرا می‌کنند و تصمیم می‌گیرند که آیا یک درخواست cross-origin باید اجازه دسترسی داشته باشد یا نه.

با استفاده از CORS، امنیت در اشتراک گذاری منابع میان دامنه‌های مختلف تضمین می‌شود و به برنامه‌های وب اجازه می‌دهد تا منابع خود را با اطمینان با دامنه‌های دیگر به اشتراک بگذارند، بدون آنکه از خطرات امنیتی مواجه شوند.

در NestJS، enableCors یک متد است که برای فعال کردن سیاست‌های Cross-Origin Resource Sharing (CORS) بر روی سرور شما استفاده می‌شود. این متد به شما امکان می‌دهد تا تنظیمات مختلفی را برای مدیریت دسترسی‌های cross-origin به منابع شما اعمال کنید. زیرا کنترل دسترسی به منابع cross-origin در سمت سرور انجام می‌شود، برخلاف کنترل‌های امنیتی مانند CORS که در مرورگرها پیاده‌سازی می‌شوند.

تابع 'enableCors' در NestJS در 'main.ts' (یا فایل‌ای که اولین بار NestJS شما را راه‌اندازی می‌کند) استفاده می‌شود. این تابع می‌تواند با تنظیمات مختلفی فراخوانی شود، که به شما امکان می‌دهد کنترل دقیق‌تری روی دسترسی‌های cross-origin داشته باشید. اکنون بیا ببینیم به توضیح هر یک از این تنظیمات بپردازیم:

- **Origin:** این تنظیم مشخص می‌کند که از کدام منابع cross-origin درخواست‌ها را قبول یا رد کنید. می‌توانید از رشته '*' استفاده کنید تا از همه‌ی منابع cross-origin درخواست‌ها را پذیرفته کنید یا یک تابع برای انتخاب دقیق‌تر کنترل اعمال کنید.
- **methods:** این تنظیم مشخص می‌کند که چه نوع متدهای HTTP برای درخواست‌های cross-origin قابل قبول هستند. به طور پیش‌فرض، GET، HEAD و POST اجازه‌شان داده می‌شود. اینجا می‌توانید متدهای دیگر مانند PUT، DELETE و PATCH را نیز اضافه کنید.

- **allowedHeaders**: این تنظیم مشخص می‌کند که چه هدرهای HTTP برای درخواست‌های cross-origin قابل قبول هستند. این می‌تواند یک رشته یا یک آرایه از رشته‌ها باشد. برای مثال، ممکن است بخواهید هدرهای مانند Content-Type و Authorization را برای درخواست‌های cross-origin قابل قبول کنید.
 - **exposedHeaders**: این تنظیم مشخص می‌کند که چه هدرهای HTTP قابل دسترسی از سمت کلاینت برای درخواست‌های cross-origin هستند. این می‌تواند یک رشته یا یک آرایه از رشته‌ها باشد.
 - **credentials**: این تنظیم یک بولین است که نشان می‌دهد آیا اجازه دسترسی به اطلاعات اعتبار احراز هویت (مانند کوکی‌ها و اعتبار دیگر) برای درخواست‌های cross-origin داده شود یا خیر.
 - **maxAge**: این تنظیم یک عدد صحیح است که مدت زمان معتبری را برای اطلاعات preflight درخواست‌های cross-origin OPTIONS مشخص می‌کند.
 - **preflightContinue**: این تنظیم یک بولین است که نشان می‌دهد آیا پردازش به درخواست‌های cross-origin ادامه داده شود یا از پردازش دستورالعمل‌های preflight استفاده شود.
 - **optionsSuccessStatus**: این تنظیم یک عدد صحیح است که کد وضعیت HTTP برای درخواست‌های cross-origin OPTIONS موفق را مشخص می‌کند.
- این تنظیمات به شما امکان می‌دهند که به صورت دقیق‌تر و سفارشی‌تر کنترل کنید که چگونه درخواست‌های cross-origin به منابع شما دسترسی دارند. هر تنظیم به طور جداگانه و با دقت باید تنظیم شود تا امنیت و عملکرد صحیح سیستم شما تضمین شود.

به عنوان مثال در پوشه ی `main.ts` به صورت زیر این تنظیمات را اعمال می‌کنیم:

```
app.enableCors({
  origin: 'localhost',
  methods: ["POST"],
  credentials: true
});
```

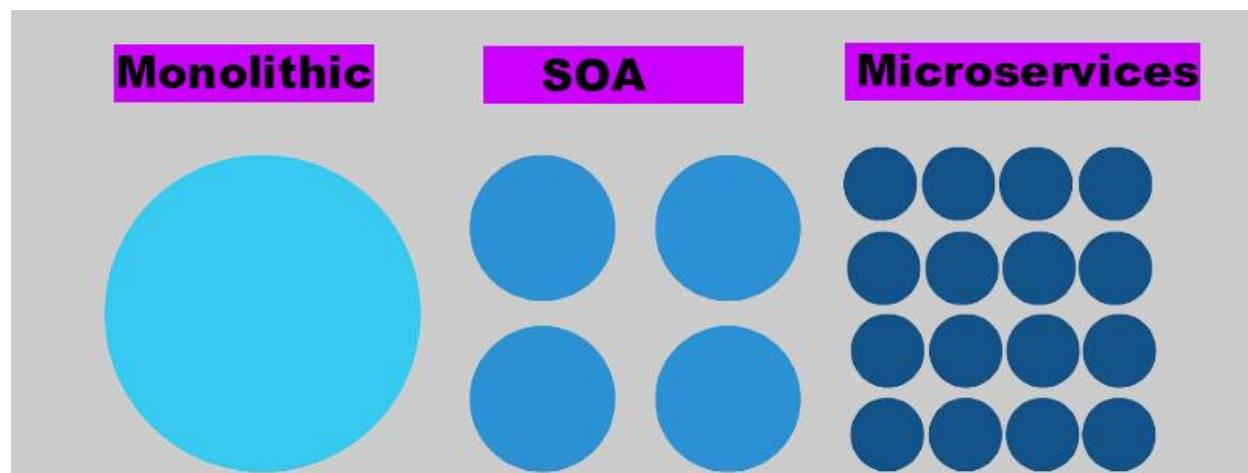
به این ترتیب تنها منبع مورد قبول برای درخواست را تعیین می‌کنیم که در اینجا تعریف شده و متد معتبر برای درخواست را نیز از نوع تعریف می‌کنیم **credentials** را نیز **true** می‌کنیم لازم به ذکر است که این تنظیمات زمانی قابل اجرا هستند که با پروتوکل HTTPS درخواست ارسال گردد حال این هدرها را می‌توان در درخواست‌ها مشاهده کرد. برای مشاهده ی جزئیات این تنظیمات و بررسی بیشتر به این [لینک](#) رجوع شود.

Access-Control-Allow-Credentials:	true
Access-Control-Allow-Origin:	localhost
Connection:	keep-alive
Content-Length:	121
Content-Security-Policy:	default-src 'self';base-uri 'self';font-src 'self' https: data;;form-action 'self';frame-ancestors 'self';img-src 'self' data;;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Content-Type:	application/json; charset=utf-8
Cross-Origin-Opener-Policy:	same-origin
Cross-Origin-Resource-Policy:	same-origin
Date:	Sat, 09 Mar 2024 10:00:13 GMT
Etag:	W/"79-aDuux1pl5sWydc/qCP/JCar1vDc"
Keep-Alive:	timeout=5
Origin-Agent-Cluster:	?1
Referrer-Policy:	no-referrer-when-downgrade
Set-Cookie:	Nestjs Learning Session=s%3ALw1QC4GXhKOyNWAmPZZxe7VOJ2-BSMv5.%2BnkDABJtW7V%2BRaJi8se83UPadlwXma%2FQXCL2RW04L2E; Path=/; Expires=Sat, 09 Mar 2024 10:30:13 GMT; HttpOnly
Strict-Transport-Security:	max-age=15552000; includeSubDomains
Vary:	Origin
X-Content-Type-Options:	nosniff
X-Dns-Prefetch-Control:	off
X-Download-Options:	noopen
X-Frame-Options:	DENY
X-Permitted-Cross-Domain-Policies:	none
X-Xss-Protection:	0

معماری میکروسرویس

اگر در سال‌های گذشته قصد راه‌اندازی یک پروژه نرم‌افزاری را داشتید، برای انتخاب معماری کار سختی در پیش نداشتید چون فقط یک گزینه به نام معماری مونولیتیک وجود داشت که برای تمامی پروژه‌ها از آن استفاده می‌کردیم. البته هنوز هم از معماری مونولیتیک استفاده می‌شود و در بسیاری از پروژه‌ها بهترین انتخاب برای ما می‌تواند همین معماری مونولیتیک باشد. اما امروزه نیاز نرم‌افزارها تغییر کرده. تعداد کاربرانی که از این وب اپلیکیشن استفاده می‌کنند نسبت به 20 سال پیش میلیون‌ها برابر شده است. پروژه‌های بزرگی مثل فیس‌بوک و آمازون و... داریم که روزانه به چندین میلیارد کاربر خدمات ارائه می‌کنند و همین نیازهای جدید باعث شد معماری‌های جدیدی به وجود بیایند که جایگزین معماری مونولیتیک شوند. معماری‌های مثل معماری سرویس‌گرا SOA و معماری میکروسرویس (Microservice).

شاید با بررسی‌ها که از پروژه خود انجام داده‌اید به این نتیجه رسیده باشید که معماری مونولیتیک برای پروژه شما مناسب نمی‌باشد، و تصمیم گرفته‌اید از یک معماری مدرن که جوابگوی نیازهای شما باشد استفاده کنید. با بررسی‌های که در اینترنت انجام داده‌اید به این نتیجه می‌رسید که دو معماری میکروسرویس و سرویس‌گرا SOA می‌تواند نیازهای شما را برطرف کند، و حالا باید بین این دو بتوانید یک گزینه را انتخاب کنید. در این مقاله تفاوت‌های معماری میکروسرویس و معماری سرویس‌گرا SOA را بررسی می‌کنیم و به شما کمک می‌کنیم به درک درستی از تفاوت‌های معماری میکروسرویس و SOA برسید که بتوانید انتخاب مناسب‌تری برای معماری پروژه‌های خود داشته باشید.



معماری مونولیتیک یک برنامه یکپارچه است که تمامی موارد نیاز پروژه به صورت یکپارچه کنار هم قرار می‌گیرد و توسعه داده می‌شود و در نهایت به دست مشتری می‌رسد.

این معماری دارای ۳ بخش مجزا است

User interface

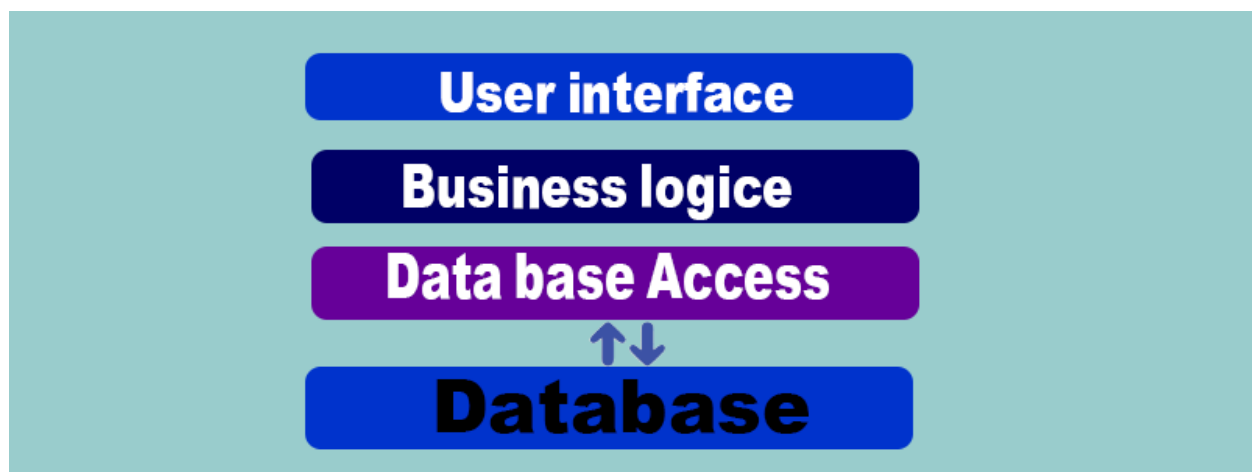
این بخش که ui پروژه است که با استفاده از html ,css نوشته می‌شود و یا ممکن است خروجی web API ها باشد که با JSON و یا XML نوشته شده باشد و در اختیار کاربر نهایی قرار می‌گیرد.

Business logice

در این بخش از منطق تجاری برنامه پیاده‌سازی می‌شود

Data Base Access

وظیفه این لایه ایجاد ارتباط با دیتابیس است، و کار ذخیره و بازیابی داده در دیتابیس را بر عهده دارد این لایه توسط لایه Business logice مورد استفاده قرار می‌گیرد.



کلمه مونولیتیک به معنای یکپارچه است و این بدین معنا است که برنامه شما یکپارچه است و بین اجزای آن اتصال محکم برقرار است.

مزایای معماری مونولیتیک:

مدیریت آسان پروژه

توسعه آسان‌تر پروژه

تست راحت‌تر نرم‌افزار

انتشار و استقرار برنامه ساده انجام می‌شود

پیچیدگی کمتر

مقیاس پذیری بهتر

معایب معماری مونولیتیک در پروژه‌های بزرگ:

1 - به سختی می‌توانید از فریم‌ورک‌ها و فناوری‌های جدید استفاده کنیم

2 - تحویل کندتر فیچرهای جدید به بازار

3 - پیچیدگی بالای کد

4 - نگهداری سخت پروژه

5 - در پروژه‌های بزرگ با معماری مونولیتیک توسعه آنها مشکل می‌شود

6 - استفاده نامناسب از منابع در زمان Scale پروژه

تا زمانی که پروژه شما بزرگ نشده معماری مونولیتیک بهترین گزینه برای شما است و می‌توانید از مزایای این معماری در پروژه‌های کوچک بهره‌مند شوید.

معماری سرویس‌گرا یا همان SOA برای رفع مشکلات معماری مونولیتیک ایجاد شد و در این معماری یک پروژه بزرگ یکپارچه که از معماری مونولیتیک استفاده می‌کند را به ماژول‌های مجزایی تقسیم می‌کنیم که هر ماژول خودش یک پروژه مونولیتیک کوچک‌تر است. این ماژول‌ها هر کدام خدمات خاصی را ارائه می‌کنند که بقیه ماژول‌ها می‌توانند از این خدمات استفاده کنند.

معماری میکروسرویس بعد از ارائه معماری SOA به وجود آمد و مزایایی بیشتری نسبت به معماری SOA دارد. می‌توانیم بگوییم که معماری میکروسرویس تکامل‌یافته معماری SOA می‌باشد که در این معماری ما سرویس‌های بیشتری داریم و خدمات خود را بسیار کوچک‌تر از معماری SOA طراحی می‌کنیم که هر کدام از این سرویس‌ها به صورت کاملاً مستقل از هم با دیتابیس اختصاصی خود توسعه داده می‌شوند.

میکروسرویس یک معماری جدید برای توسعه نرم افزار می باشد. در این معماری نرم افزار را به بخش های کوچک و مستقلی تقسیم می کنیم که هر کدام از این سرویس ها دیتابیس اختصاصی خود را دارند و از طریق api و یا Message broker ها با هم ارتباط برقرار می کنند.

معماری میکروسرویس و SOa شباهت های زیادی با هم دارند. در هر دوی این معماری ها پروژه یکپارچه را تقسیم می کنیم به بخش های کوچک تری که این بخش های جداگانه خدماتی که به دیگر بخش ها ارائه می کنند. در هر دوی این معماری ها می توانیم از کانتینرها و فضای ابری استفاده نماییم که همین باعث می شود با راهکارهای مدرن تر بتوانیم Scale برنامه را به خوبی انجام دهیم و به کاربران بیشتری بدون افت کیفیت خدمات ارائه کنیم . با این که این دو معماری شباهت های زیادی با هم دارند اما تفاوت های زیادی هم دارند که دانستن این تفاوت ها به شما کمک می کند انتخاب بهتری داشته باشید

تفاوت در معماری Arciteture

در معماری میکروسرویس سرویس های که توسعه داده می شوند بر خدمات واحد تمرکز می کنند و این یعنی که در میکروسرویس سرویس های ما بسیار کوچک تر هستند و به صورت مستقل کار می کنند و این یعنی باید دیتابیس اختصاصی خود را هم داشته باشند.

در سرویس های معماری میکروسرویس به دلیل کوچک بودنشان خدمات کمتری هم ارائه می دهند. پس در نتیجه ما در معماری میکروسرویس سرویس های بسیار بیشتری داریم، به عنوان مثال در یک پروژه فروشگاهی می توانیم یک سرویس برای نمایش پرفروش ترین محصولات داشته باشیم و یک سرویس برای نمایش نظرات هر محصول و... اما در معماری SOa سرویس های ما بزرگ تر هستند که می توانند مستقل از هم نباشند به عنوان مثال می توانیم یک سرویس برای کل حسابداری داشته باشیم، و یا یک سرویس برای کل انبارداری.

مرزبندی سرویس‌ها

مرزبندی و یا تقسیم‌بندی سرویس‌ها در این دو معماری با هم متفاوت است. در معماری SOA معمولاً سرویس‌های بزرگی داریم، سرویس‌های که شاید خودشان در حد یک برنامه مونولیتیک بزرگ باشند. اما در معماری میکروسرویس مرزبندی و یا همان تقسیم‌بندی که برای سرویس‌ها انجام می‌دهیم بسیار کوچک‌ترند. سرویس‌های معماری میکروسرویس این‌قدر باید کوچک باشند که در دوهفته تیم توسعه آن بتواند کامل آن را بازنویسی کند.

نحوه ذخیره‌سازی داده‌ها

در معماری میکروسرویس هر سرویس دیتابیس اختصاصی خودش را دارد و تمام دیتاهای مورد نیازش را در دیتابیس اختصاصی خود ذخیره می‌کند. به‌عنوان مثال در سرویسی که برای سبد خرید ایجاد می‌کنیم نیاز به اطلاعات محصول هم داریم، در این معماری باید تمامی اطلاعات هر محصولی که به سبد خرید اضافه می‌شود را در دیتابیس اختصاصی سرویس سبد خرید ذخیره کنیم که هر بار برای استفاده از آن داده‌ها نیاز نباشد به سرویس محصولات دسترسی داشته باشیم. این نوع نگرش باعث می‌شود در معماری میکروسرویس هر سرویس دیتاهای اختصاصی خاص خود را داشته باشد. اما در معماری SOA به این صورت نیست و سرویس‌ها می‌توانند دیتاهای اشتراکی بیس سرویس‌های مختلف را استفاده نمایند که خود این نیز مزایای برای ما می‌تواند داشته باشد.

حاکمیت در سرویس‌های مختلف

در معماری SOA از دیتاهای اشتراکی استفاده می‌کنیم و همین باعث می‌شود با حاکمیت و یا قوانینی که برای نحوه ایجاد و استفاده از داده‌ها وضع می‌کنیم برای تمامی سرویس‌ها یکسان باشد چون سرویس‌های مختلف به‌صورت اشتراکی از دیتاهای دیگر سرویس‌ها می‌توانند استفاده کنند. اما در مقابل، معماری میکروسرویس به این صورت نیست. چون هر سرویس کاملاً مستقل از سرویس‌های دیگر می‌باشد و می‌توانیم برای هر سرویس قوانین اختصاصی تعریف کنیم.

اندازه دامنه

در معماری میکروسرویس اندازه سرویس‌ها کوچک‌تر می‌باشد و در نتیجه دامنه فعالیت نیز بسیار کوچک‌تر می‌باشد، و از این‌رو برای توسعه دهندگان کار بر روی این سرویس لذت‌بخش‌تر می‌باشد. زیرا تیم توسعه هر سرویس به دلیل اندازه کوچک دامنه به سرعت تسلط کافی بر دامنه و دانش لازم بیزینس را به دست می‌آورند و خیلی بهتر می‌توانند سرویس را توسعه بدهند. اما در معماری SOA اندازه و دامنه سرویس‌ها بسیار بزرگ‌تر است و هرچه این اندازه بزرگ‌تر باشد پیچیدگی‌های بیشتر در توسعه برای ما ایجاد می‌کند.

ارتباط بین سرویس‌ها

در معماری soa ارتباط بین سرویس به طور سنتی توسط ESB انجام می‌شود و با استفاده از این روش سرویس‌ها می‌توانند با هم صحبت کنند که خود این یکی از دلایل کند بودن ارتباط در معماری سرویس‌گرا می‌باشد. معماری میکروسرویس از روش‌های ساده‌تری مانند API ها استفاده می‌کند که باعث افزایش سرعت در ارتباط‌ها می‌شود.

Deployment

Deployment یا استقرار نرم‌افزار در معماری میکروسرویس نسبت به معماری soa ساده‌تر انجام می‌شود. در معماری میکروسرویس سرویس‌ها کوچک و کاملاً مستقل از هم هستند و ما به راحتی می‌توانیم بدون آنکه مشکلی برای بقیه سرویس‌ها به وجود بیایید نسخه جدید یک سرویس را Deploy کنیم و که دیگر سرویس‌ها بتوانند از آن استفاده کنند. اما در معماری SOA معمولاً Deployment پیچیده است چون سرویس‌ها بزرگ هستند و تقریباً می‌توانیم بگوییم همان مشکلات Deployment معماری مونولیتیک را در معماری SOA هم داریم.

دسترسی به سرویس‌ها

معماری میکروسرویس و معماری سرویس‌گرا از پروتکل‌های مختلفی برای دسترسی به دیگر سرویس‌ها استفاده می‌کنند. معمولاً در معماری سرویس‌گرا پروتکل اصلی که مورد استفاده قرار می‌گیرد پروتکل (SOAP) می‌باشد و البته از پروتکل‌های مانند (AMQP) هم برای ارتباط استفاده می‌شود. اما در معماری میکروسرویس از پروتکل REST و یا از GRPC برای ارتباط‌های Sync استفاده می‌شود و از پروتکل‌های AMQP و دیگر پروتکل‌ها هم برای ارتباط‌های Async استفاده می‌شود.

هر دوی این معماری‌ها رویکردهای برای جداسازی سرویس‌ها از هم را ارائه می‌کنند و این سرویس‌ها را می‌توانیم در فضای ابری و در کانتینرها مستقر کنیم و همین باعث می‌شود بتوانیم از Auto Scale در این معماری‌ها بهره‌مند شویم. این که ما از کدام معماری استفاده کنیم دقیقاً به نیاز آن پروژه بستگی دارد که برای هر پروژه می‌تواند این متفاوت باشد و با توجه به مواردی که در این مقاله بررسی کردیم می‌توانید تصمیم بگیرید از کدام معماری در پروژه خود استفاده نمایید.

در ادامه قصد داریم یک مثال ساده از معماری میکروسرویس را به کمک Nestjs و kafka پیاده سازی کنیم...

پیاده سازی معماری میکروسرویس

برای پیاده سازی میکرو سرویس در این پروژه از کافکا استفاده می کنیم برای این منظور ابتدا کافدراپ را از ریپازیتوری زیر کلون می کنیم و به صورت زیر با استفاده از داکر آن را نصب و فعالسازی می کنیم: (دقت شود که برای فعالسازی باید نرم افزار داکر در سیستم فعال باشد)

```
git clone https://github.com/obsidiandynamics/kafdrop
```

```
cd kafdrop
```

```
cd docker-compose/kafka-kafdrop
```

```
docker-compose up
```

پس از اجرای این کد سه سرویس مجزا را به صورت زیر در سه ترمینال جداگانه ایجاد می کنیم:

```
nest new api-gateway
```

```
nest new billing
```

```
nest new auth
```

پس از راه اندازی این سه سرویس برای اجرای میکرو سرویس باید در هر سه ی آن ها کتابخانه ی زیر را نصب کنیم:

```
npm i @nestjs/microservices kafkajs
```

سپس هر سه سرویس را ران می کنیم (اولین سرویس باید gateway باشد.)

در بخش api-gateway قصد داریم یک API از نوع Post ایجاد کنیم بنابراین ابتدا برای آن یک dto ایجاد می کنیم:

```
export class createOrderRequest{
  userId:string;
  price:number;
}
```

سپس به کنترلر این سرویس رفته و به صورت زیر یک API پست را برای آن تعریف می کنیم:

```
@Post
createOrder(@Body() CreateOrderRequest:createOrderRequest){
  this.appService.createOrder(CreateOrderRequest);
}
```

حال باید تابع createOrder را در appService تعریف کنیم:

```
creatOrder({userId,price}:createOrderRequest){  
  }  
}
```

در آینده به کمک emit در این تابع یک رویداد را منتشر خواهیم کرد اما فعلا آن را رها کرده و به سراغ پوشه ی app.module.ts در سرویس gateway می رویم این سرویس ارتباطات میان سرویس ها را براساس کافکا وارد می کنیم.

```
import { Module } from '@nestjs/common';  
import { AppController } from './app.controller';  
import { AppService } from './app.service';  
import { ClientsModule, Transport } from '@nestjs/microservices';  
  
@Module({  
  imports: [  
    ClientsModule.register([  
      {  
        name: 'BILLING-SERVICE',  
        transport: Transport.KAFKA,  
        options: {  
          client: {  
            clientId: 'billing',  
            brokers: ['localhost:9092']  
          },  
          consumer: {  
            groupId: 'billing-consumer'  
          }  
        }  
      ]  
    ],  
  ],  
  controllers: [AppController],  
  providers: [AppService],  
})  
export class AppModule {}
```

دثر این بخش ابتدا نام سرویس را تعریف می کنیم BILLING-SERVICE سپس نوع ارتباط را از نوع KAFKA تعریف می کنیم حال در بخش آپشن های ارتباط آیدی کلاینت را که براساس آن ارتباط میان سرویس ها شکل می گیرد و بروکر را که دیفالت کافکا localhost:9092 است را تعریف می کنیم در بخش کانسومر نیز آیدی مصرف کنندگان این سرویس را billing-consumer تعریف می کنیم.....

مجدداً به `api-gateway` و سرویس باز می گردیم تا تابعی که نوشته بودیم را کامل کنیم پیش از آن اما باید تایپ رویداد را تعیین کنیم برای این منظور یک پوشه در `api-gateway` تعریف به نام `order-created.event.ts` به صورت زیر تعریف می کنیم:

```
export class orderCreatedEvent {
  constructor(
    public readonly orderId:string,
    public readonly userId:string,
    public readonly price:number,
  ){}

  toString(){
    return JSON.stringify({
      orderId: this.orderId,
      userId: this.userId,
      price: this.price,
    })
  }
}
```

در این کلاس از متد برای سریالایز کردن رویداد مد نظر استفاده می کنیم حال تابع `creatOrder` را به شکل زیر تکمیل می کنیم:

```
creatOrder({userId,price}:createOrderRequest){
  this.billingClient.emit('order_created',
    new orderCreatedEvent('123',userId,price))
}
```

سپس به بخش `main` در سرویس `billing` رفته و تعیین می کنیم که این سرویس بجای `HTTP` به `کافکا` گوش دهد:

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

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport:Transport.KAFKA,
      options:{
```

```

    client:{
      brokers:['localhost:9092']
    },
    consumer:{
      groupId:'billing-consumer'
    }
  }

}

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

```

با این تنظیمات سرویس billing از طریق 'billing-consumer' متوجه می شود که باید به این گروه از متقاضیان پاسخ دهد. سپس در بخش کنترلر سرویس billing یک API جدید می سازیم:

```

@EventPattern('order_created')
hanleOrderCreated(data:any){
  this.appService.hanleOrderCreated(data)
}

```

در بخش تعیین می کنیم که این API باید به چه API از gateway متصل شود در واقع پیام EventPattern همان پیامی است که در سرویس gateway تعریف شده است حال باید تابع hanleOrderCreated را در سرویس billing تعریف کنیم برای این منظور باید یک رویداد مشابه آنچه در gateway تعریف کردیم را در سرویس billing نیز تعریف کنیم با همان نام order-created.event.ts :

```

export class orderCreatedEvent {
  constructor(
    public readonly orderId:string,
    public readonly userId:string,
    public readonly price:number,
  ){
  }
}

```

سپس سرویس مربوطه را در billing می نویسیم لازم به ذکر است که در این رویداد نیازی به سریالایز کردن نداریم:

```
hanleOrderCreated(OrdeCreatedEvent:orderCreatedEvent){
  console.log(OrdeCreatedEvent);
}
```

ابتدا صرفا برای اینکه بررسی کنیم که آیا سیستم میکرو سرویسمان کار می کند یا خیر دیتای پست شده به api-gateway را در سرویس billing لاگ می گیریم به کمک postman داده های زیر را ارسال می کنیم:

```
{
  "userId":"Suorena",
  "price":100
}
```

سپس در خروجی ترمینال billing داریم:

LOG [NestMicroservice] Nest microservice successfully started +2ms

```
{orderId: '123', userId: 'Suorena', price: 100}
```

بنابراین میکروسرویس به درستی کار می کند. حال باید سرویس AUTH را تعریف کنیم و قصد داریم سرویس AUTH به عنوان ارائه دهنده ی خدمات به سرویس billing تعریف نماییم برای این کار همانند قبل به فایل app.module.ts در سرویس billing رفته سرویس AUTH را تعریف می کنیم:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ClientsModule, Transport } from '@nestjs/microservices';

@Module({
  imports: [ClientsModule.register([
    {
      name: 'AUTH-SERVICE',
      transport: Transport.KAFKA,
      options: {
        client: {
          clientId: 'auth',
          brokers: ['localhost:9092']
        },
        consumer: {
          groupId: 'auth-consumer'
        }
      }
    }
  ])]
})
```

```

    }
  ])
],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}

```

سپس همانند قبل این سرویس را در سرویس billing اینجکت می کنیم:

```

import { Injectable ,Inject } from '@nestjs/common';
import { orderCreatedEvent } from './order-created.event';
import { ClientKafka } from '@nestjs/microservices';

@Injectable()
export class AppService {
  constructor(
    @Inject('AUTH-SERVICE') private readonly authClient:ClientKafka,
  ){}
  getHello(): string {
    return 'Hello World!';
  }
  hanleOrderCreated(OrdeCreatedEvent:orderCreatedEvent){
    console.log(OrdeCreatedEvent);
  }
}

```

حال دقیقا همانند بخش قبل اینبار سرویس billing به سرویس AUTH متصل می کنیم برای این منظور باید ابتدا سرویس AUTH را در کد تعریف کنیم:

Billing (app.module.ts):

```

import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ClientsModule, Transport } from '@nestjs/microservices';

@Module({
  imports: [ClientsModule.register([
    {
      name: 'AUTH-SERVICE',
      transport:Transport.KAFKA,
      options:{

```

```

        client:{
            clientId:"auth",
            brokers:['localhost:9092']
        },
        consumer:{
            groupId:'auth-consumer'
        }
    }
}
])
],
controllers: [AppController],
providers: [AppService],
})
export class AppModule {}

```

حال سرویس billing را یازنویسی می کنیم و اینبار اطلاعات مدنظر را به سرویس AUTH از طریق emit ارسال می کنیم:

Billing (app.service.ts):

```

import { Injectable ,Inject } from '@nestjs/common';
import { orderCreatedEvent } from './order-created.event';
import { ClientKafka } from '@nestjs/microservices';
import { getUser } from './get_user.dto';

@Injectable()
export class AppService {
    constructor(
        @Inject('AUTH-SERVICE') private readonly authClient:ClientKafka,
    ){}
    getHello(): string {
        return 'Hello World!';
    }
    async hanleOrderCreated(OrdeCreatedEvent:orderCreatedEvent){

        await this.authClient.connect();
        console.log("connected")
        await this.authClient.emit('get_user' ,
            new
orderCreatedEvent(OrdeCreatedEvent.orderId,OrdeCreatedEvent.userId,OrdeCreatedEvent.price))

    }
}

```

```
}
```

عبارت 'get_user' بیانگر این است که ما از چه سرویسی در AUTH قصد استفاده را داریم. مجدداً در سرویس billing یک dto برای get_user می‌سازیم:

Billing(get_user.dto.ts):

```
export class getUser{
  constructor(public readonly userId:string){}
  toString(){
    JSON.stringify({
      userId: this.userId
    })
  }
}
```

حال باید در سرویس AUTH تعریف کنیم که به جایی HTTP به سرویس KAFKA گوش فرا دهد بنابراین در پوشه ی main.ts سرویس AUTH تنظیمات زیر را وارد می‌کنیم:

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

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport:Transport.KAFKA,
      options:{
        client:{
          brokers:['localhost:9092']
        },
        consumer:{
          groupId:'auth-consumer'
        }
      }
    }
  )
  app.listen();
}
bootstrap();
```

حال در بخش کنترلر سرویس AUTH باید یک سرویس بنویسیم که با پیام 'get_user' فعال شود برای این منظور هم همانند قبل از EventPattern استفاده می کنیم:

AUTH(app.controller.ts):

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from '../app.service';
import { EventPattern } from '@nestjs/microservices';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }

  @EventPattern('get_user')
  async getUser(data: any) {
    await this.appService.getUser(data)
  }
}
```

حال پیش از اینکه تابع getUser را در بخش سرویس تکمیل کنیم باید ابتدا یک dto برای getUser بنویسیم:

AUTH(get_user.dto.ts):

```
export class getUser {
  constructor(
    public readonly orderId:string,
    public readonly userId:string,
    public readonly price:number,
  ){}
}
```

حال باید بخش سرویس AUTH را تکمیل کنیم و تابع getUser را تعریف کنیم:

```
import { Injectable } from '@nestjs/common';
import { getUser } from './get_user.dto';

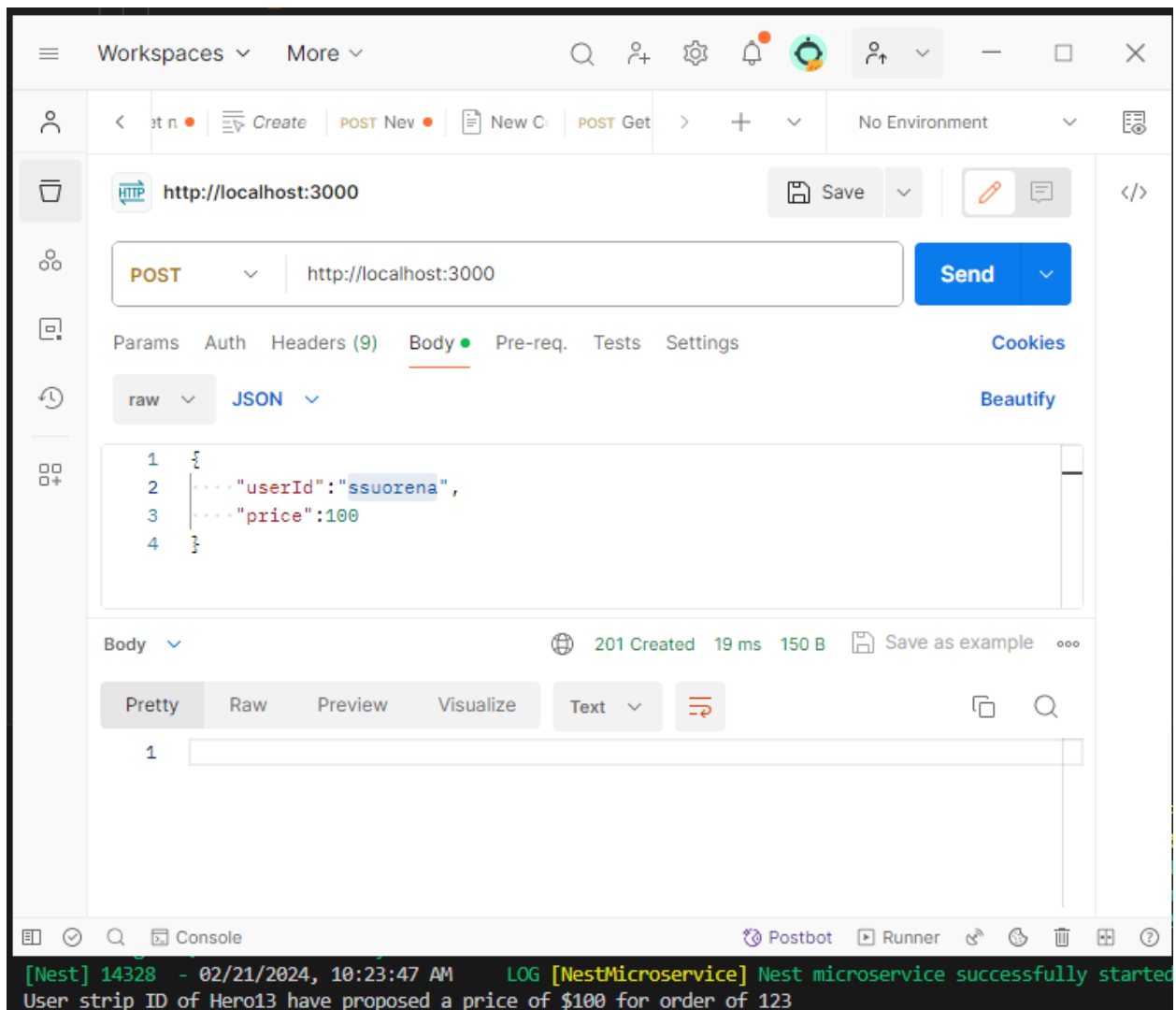
@Injectable()
export class AppService {
  private readonly users:any[]=[
    {
      userId:"ssuorena",
      stripeUserId:"Hero13"
    },
    {
      userId:"MSZ",
      stripeUserId:"The0M"
    },
    {
      userId:"hamed",
      stripeUserId:"esiblack"
    },
  ]

  getHello(): string {
    return 'Hello World!';
  }

  async getUser(GetUser:getUser){
    Userid:String
    const Userid = GetUser.userId
    const Orderid = GetUser.orderId
    const Price = GetUser.price
    const user = await this.users.find((user) => user.userId === Userid);
    console.log(`User strip ID of ${user.stripeUserId} have proposed a price of
    ${Price} for order of ${Orderid}`)

  }
}
```


در این بخش به جای استفاده از دیتاست از یک لیست با سه عضو استفاده می کنیم و در نهایت کاربر را در این لیست براساس آیدی پیدا کرده و یک لاگ را به عنوان گزارش براساس strip ID تهیه می کنیم به این ترتیب هر سه سرویس AUTH,Billing و Gateway در ایجاد این لاگ نقش داشته و باهم همکاری دارند.



آنچه تا این مرحله بررسی شد ارتباط عمقی میان سرویس ها بود یعنی ما از یک سرویس به سرویس دیگر صرفا پیام می فرستادیم و انتظار دریافت پاسخ از سرویس دوم را نداشتیم اما چه می شود اگر از سرویسی که به آن پیام داده باشیم انتظار خروجی وجود داشته باشد. در اینجا است که به جای استفاده از متد emit باید از متد send استفاده گردد. در ادامه قصد داریم لاگ ایجاد شده در سرویس AUTH را به عنوان پاسخ به سرویس billing

بفرستیم و در آنجا آن را نمایش دهیم برای این کار ابتدا باید در سرویس billing در بخش کنترلر ابتدا از کلاس را از onModuleInit ایمپلیمنت می کنیم و سپس متد onModuleInit را به صورت زیر ایجاد می کنیم.

Billing(app.controller.ts)

```
import { Controller, Get, Inject, OnModuleInit } from '@nestjs/common';
import { AppService } from '../app.service';
import { ClientKafka, EventPattern, MessagePattern } from '@nestjs/microservices';
import { orderCreatedEvent } from '../order-created.event';

@Controller()
export class AppController implements OnModuleInit {
  constructor(
    private readonly appService: AppService,
    @Inject('AUTH-SERVICE') private readonly authClient: ClientKafka,
  ) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }

  @EventPattern('order_created')
  async hanleOrderCreated(data: any) {
    await this.appService.hanleOrderCreated(data);
    console.log("emited", data);
  }

  onModuleInit() {
    this.authClient.subscribeToResponseOf("GET_USER");
  }
}
```

در اینجا نیز برای send کردن داده ها به سرویس AUTH از همان تابع hanleOrderCreated که برای emit کردن استفاده کردیم استفاده می کنیم. حال باید در این تابع تغییرات زیر را اعمال کنیم:

Billing(app.service.ts)

```
import { Injectable ,Inject } from '@nestjs/common';
import { orderCreatedEvent } from './order-created.event';
import { ClientKafka } from '@nestjs/microservices';
import { getUser } from './get_user.dto';

@Injectable()
export class AppService {
  constructor(
    @Inject('AUTH-SERVICE') private readonly authClient:ClientKafka,
  ){}
  getHello(): string {
    return 'Hello World!';
  }
  async hanleOrderCreated(OrdeCreatedEvent:orderCreatedEvent){

    await this.authClient.connect();
    console.log("connected")
    await this.authClient.emit('get_user' ,
      new
orderCreatedEvent(OrdeCreatedEvent.orderId,OrdeCreatedEvent.userId,OrdeCreatedEvent.price))
    const message = await this.authClient.send('GET_USER',
      new
orderCreatedEvent(OrdeCreatedEvent.orderId,OrdeCreatedEvent.userId,OrdeCreatedEvent.price)).
      subscribe((user)=>{console.log("reply message is:",user)})

  }
}
```

همانطور که می بینید اینبار علاوه بر emit کردن داده ها به کمک کد زیر داده ها را send کرده ایم و پاسخ ارسالی از سرویس AUTH خود به خود با استفاده از ماژول subscribe در متغیر user قرار می گیرد که ما در اینجا صرفا این پاسخ را چاپ کرده ایم. حال باید به سراغ سرویس AUTH برویم. ابتدا باید در بخش کنترلر داده های دریافتی از سرویس billing را به کمک دکوراتور MessagePattern("GET_USER") دریافت کرده پردازش کنیم و به کمک return به سرویس billing می فرستیم:

AUTH(app.controller.ts)

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from '../app.service';
import { EventPattern, MessagePattern } from '@nestjs/microservices';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService,
  ){}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }

  @EventPattern('get_user')
  async getUser(data:any){
    await this.appService.getUser(data)

  }

  @MessagePattern("GET_USER")
  async GETuser(data:any){
    const user = await this.appService.GETuser(data)
    return user
  }
}
```

بردازش داده های دریافتی همانطور که در کد فوق پیداست به عهده ی GETuser است بنابراین اکنون کافی است این تابع را در بخش سرویس AUTH تعریف کنیم:

AUTH(app.service.ts)

```
import { Injectable } from '@nestjs/common';
import { getUser } from './get_user.dto';

@Injectable()
export class AppService {

  private readonly users:any[]=[
    {
      userId:"ssuorena",
      stripeUserId:"Hero13"
    },
    {
      userId:"MSZ",
      stripeUserId:"The0M"
    },
    {
      userId:"hamed",
      stripeUserId:"esiblack"
    },
  ]

  getHello(): string {
    return 'Hello World!';
  }

  async getUser(GetUser:getUser){
    Userid:String
    const Userid = GetUser.userId
    const Orderid = GetUser.orderId
    const Price = GetUser.price
    const user = await this.users.find((user) => user.userId === Userid);
    console.log('Emitted message: ',`User strip ID of ${user.stripeUserId} have
proposed a price of $$${Price} for order of ${Orderid}`)

  }

  async GETuser(GetUser:getUser){
    Userid:String
    const Userid = GetUser.userId
```

```

const Orderid = GetUser.orderId
const Price = GetUser.price
const user = await this.users.find((user) => user.userId === Userid);
console.log('Sended message: ',user)
return `User strip ID of ${user.stripeUserId} have proposed a price of
${Price} for order of ${Orderid} (This message is from auth service)`
}
}

```

حال مجددا پیام را پست کرده اما اینبار ترمینال مربوط به سرویس billing را بررسی می کنیم:

The screenshot shows the Postman interface. The top bar indicates the workspace and environment. The left sidebar shows the 'Collections' and 'Environments' sections. The main area displays a POST request to `http://localhost:3000`. The request body is set to JSON and contains the following data:

```

{
  "userId": "MSZ",
  "price": 100
}

```

The response is displayed in the 'Body' tab, showing the following message:

```

reply message is: User strip ID of The0M have proposed a price of $100 for order of 123 (This message is from auth service)

```

