EMAILS NEST

- Creo la app con nest **new emails-app**
- Borro todos los archivos innecesarios como el app.service, app.controller, los tests .spec
- Instalo swagger con @nestjs/swagger, lo configuro.
- Instalo el class-validator class-transformer
- Lo configuro en el mail con **app.useGlobalPipes**

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

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

  app.useGlobalPipes(new ValidationPipe({
    transform: true
  }))
  const config = new DocumentBuilder()
  .setTitle('Emails example')
  .setDescription('emails app')
  .setVersion('1.0')
  .build();
const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('emails', app, document);
  await app.listen(3000);
}
bootstrap();
```

- Creo el módulo de email con **nest g res emails**
- Creo el dto

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

export class EmailDto {

    @ApiProperty({
        name: 'body',
        required: true,
        type: String,
        description: "Cuerpo del mensaje a enviar"
    })
    @IsString()
    @IsNotEmpty()
    body: string
```

```
    @ApiProperty({
        name: 'subject',
        required: true,
        type: String,
        description: "Asunto del mensaje a enviar"
    })
    @IsString()
    @IsNotEmpty()
    subject: string

    @ApiProperty({
        name: 'receivers',
        required: true,
        isArray: true,
        type: String,
        description: "Destinatarios del mensaje a enviar"
    })
    @IsArray()
    @IsNotEmpty()
    receivers: string[]
}
```

- Para validar el array de strings voy a crear otro dto que llamaré sender-dto

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

export class SenderDto{

    @ApiProperty({
        name: 'email',
        required: true,
        type: String,
        description: "Email del destinatario"
    })
    @IsEmail()
    @IsNotEmpty()
    email: string
}
```

- Uso el decorador **@ValidateNested** en el receivers MessageDto para validar lo que hay dentro del array
- Le digo each: true (dentro de cada elemento que se cumpla lo que te indico)
- Con el class-transformer le indico cin **@Type** que es de tipo SenderDto

```
@ApiProperty({
    name: 'receivers',
    required: true,
```

```
    type: String,
    description: "Destinatarios del mensaje a enviar"
})
@IsArray()
@IsNotEmpty()
@ValidateNested({each: true})
@Type(()=> SenderDto[])
receivers: SenderDto[]
```

EmailConfig

- Creo el archivo emai-config.ts dentro de la carpeta emails
- Le coloco el puerto 25 por defecto. No siempre hay que indicarle el puerto, lo pongo como opcional
- Le indico con secure si quiero que sea con seguridad
- Para los servicios creo un enum

```
export class EmailConfig{
    from: string

    password: string

    service: SERVICES

    port?: number = 25

    secure?: boolean = false
}

export enum SERVICES{
    GMAIL= 'gmail',
    OUTLOOK = 'outlook365'
}
```

Módulo dinámico

- Un módulo dinámico es un módulo al que **le podemos pasar parámetros** y le podemos cambiar los controllers, los providers, etc
- Se usa cuando un módulo necesita comportarse diferente en diferentes casos de uso, a modo de plugin
    - Un buen ejemplo es un módulo de configuración
- Creo un método static llamado register al que le paso unas opciones de tipo **EmailConfig** que devuelve algo de tipo **DynamicModule**
- Para cuando quiera pasar las opciones a mi EmailsService creo un objeto y lo llamo **CONFIG_OPTIONS**

```
import { DynamicModule, Module } from '@nestjs/common';
import { EmailsService } from './emails.service';
import { EmailsController } from './emails.controller';
import { EmailConfig } from './email-config';

@Module({
  controllers: [EmailsController],
  providers: [EmailsService]
})
export class EmailsModule {
  static register(options: EmailConfig): DynamicModule{
    return{
      module: EmailsModule,
      controllers:[EmailsController],
      providers: [
        {
          provide: 'CONFIG_OPTIONS',
          useValue: options
      },
        EmailsService]
    }
  }
}
```

- En el service lo obtengo usando **@Inject**
- Instalo Nodemailer con **npm i nodemailer**
- Lo importo en el servicio

```
import { Inject, Injectable } from '@nestjs/common';
import { UpdateEmailDto } from './dto/update-email.dto';
import { EmailConfig } from './email-config';
import * as nodemailer from 'nodemailer'

@Injectable()
export class EmailsService {

  constructor(
    @Inject('CONFIG_OPTIONS')
    private options: EmailConfig
  ){
    console.log(this.options)
  }
}
```

Creando un endpoint para enviar un email

- Creo un POST en el controller

```
@Post()
sendEmail(@Body() emailDto: EmailDto) {
  return this.emailsService.sendEmail(emailDto);
}
```

Configurando módulo para enviar correos con gmail

- En app.module, con la función register le paso los campos de email-config

```
import { Module } from '@nestjs/common';
import { EmailsModule } from './emails/emails.module';
import { SERVICES } from './emails/email-config';

@Module({
  imports: [EmailsModule.register({
    from: 'miemail@gmail.com',
    password: 'mi-password',
    service: SERVICES.GMAIL
  })],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Cuando arrancio el servidor, el console.log(options) en el constructor del servicio me devuelve este objeto

```
{
  from: 'miemail@gmail.com',
  password: 'mi-password',
  service: 'gmail'
}
```

- Con lo que lo está pillando. Quito el console.log
- Hay que **configurar la doble verificación en gmail**
- En Seguridad/Verificación en dos pasos (activarla)
- Hay que sincronizar la cuenta con el teléfono
- Ahora, en Iniciar sesión en Google, aparece **Contraseñas de aplicaciones**
- Selecciona la aplicación o dispositivo: Otro (y le pongo un nombre, por ej: test nest)
- Esto **me va a generar una contraseña**. La usaremos en lugar de mi password

---

Enviar correos con Nodemailer

- Usaremos los "well-known services" que ya tienen su host y puerto pre-definidos, no hace falta indicarlos
- Creo el transporter según la documentación de nodemailer en el método sendEmail
- A la función sendEmail no le puedo pasar el dto tal cual, lo configuro en emailOptions
- Para el to, al ser un array, uso el .map
- Lo coloco todo dentro de un try catch por si falla

```
  sendEmail(message: EmailDto){

    try {
      const transporter = nodemailer.createTransport({
        service: this.options.service,
        auth: {
          user: this.options.from,
          pass: this.options.password
        }
      })


      const to = message.receivers.map(e =>e.email )

      const mailOptions = {
        from: this.options.from,
        to,
        subject: message.subject,
        html: message.body
      }

      return transporter.sendEmail(mailOptions)
    } catch (error) {
      console.error(error)

      return null
    }
  }
```

- Ahora habría que testearlo con ThunderClient o POSTMAN, indicándole en un objeto el subject, el to, etc...

```
{
  "subject": "Test correo",
  "body": "<h1>Hola mundo! </h1>",
  "receivers": [
   { "email": "perico@gmail.com"}
  ]
}
```

- Para usar un correo de outlook solo tengo que cambiar el SERVICE e indicar mi correo de outlook

Documentación endpoint

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { EmailsService } from './emails.service';
import { EmailDto } from './dto/create-email.dto';
import { ApiBody, ApiOperation, ApiResponse } from '@nestjs/swagger';


@Controller('emails')
export class EmailsController {
  constructor(private readonly emailsService: EmailsService) {}

  @Post()
  @ApiOperation({
    description: 'envía un email',

  })
  @ApiBody({
    description: 'envía un email usando un emailDto',
    type: EmailDto,
    examples:{
      ejemplo1:{
        value: {
          subject: "Test correo",
          body: "<h1>Hola mundo! </h1>",
          receivers: [
           { email: "perico@gmail.com"}
          ]
        }
      }
    }
  })
  @ApiResponse({
    status: 201,
    description: "Correo enviado correctamente"
  })
  sendEmail(@Body() emailDto: EmailDto) {
    return this.emailsService.sendEmail(emailDto);
  }

}
```