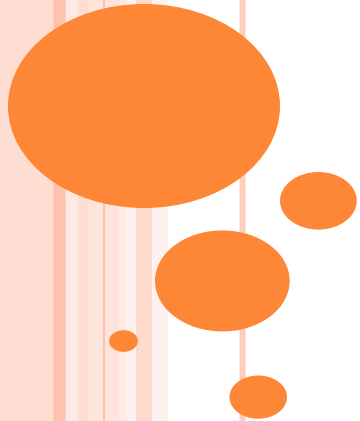


PROGRAMACIÓN EN EL SERVIDOR CON NODEJS/NEST



FUNDAMENTOS NEST

- **Framework para la creación de aplicaciones de lado de servidor (backend), basado en Node.js.**
- **Utiliza TypeScript como lenguaje de programación.**
- **Inspirado en la filosofía de Angular para el desarrollo de aplicaciones MVC**
- **Soporte para HTTP**
- **Integración con bases de datos**

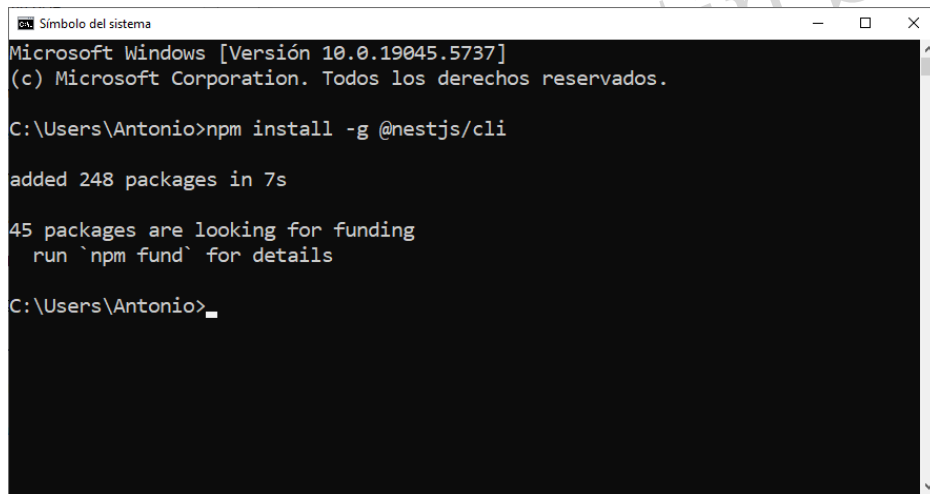


INSTALACIÓN

➤ Se requiere tener instalado Node.js con npm.

➤ Para instalar Nest:

```
>npm install -g @nestjs/cli
```



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19045.5737]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Antonio>npm install -g @nestjs/cli

added 248 packages in 7s

45 packages are looking for funding
  run `npm fund` for details

C:\Users\Antonio>
```



CREACIÓN DE UN PROYECTO NEST

➤ Para crear un proyecto Nest, nos colocaremos sobre la carpeta en la que queremos que esté y escribimos:

```
>nest new primer_proyecto
```

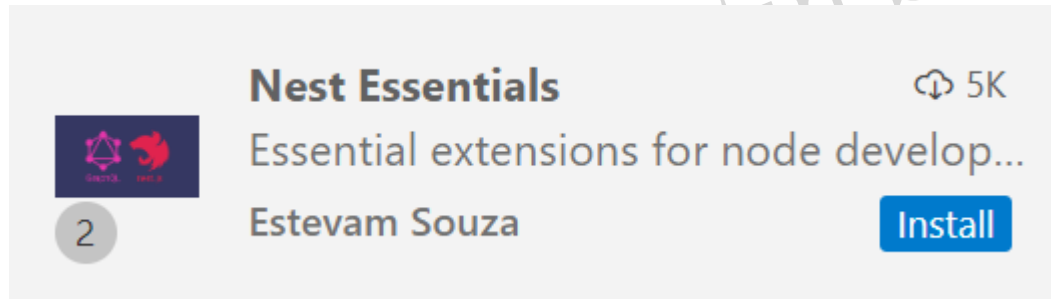
➤ Si nos pregunta el tipo de gestor de paquetes a utilizar, elegiremos npm

➤ Se creará un proyecto básico con esta composición de archivos en su carpeta src:

nest > primer_proyecto > src	
Nombre	Fecha
app.controller.spec.ts	02/05/2025 11:45
app.controller.ts	02/05/2025 11:45
app.module.ts	02/05/2025 11:45
app.service.ts	02/05/2025 11:45
main.ts	02/05/2025 11:45

NEST EN VISUAL STUDIO CODE

- Si queremos desarrollar proyectos Nest en Visual Studio Code, hay diversos plugins que nos pueden ayudar.
- El Nest Essentials agrupa varios de esos plugins que nos facilitan la programación de aplicaciones Nest:



EJECUCIÓN

➤ Para ejecutar una aplicación Nest, nos situamos en la carpeta raíz del proyecto y escribimos:

```
>npm run start:dev
```

➤ Se iniciará el servidor de aplicaciones en el puerto 3000 y podremos acceder a la dirección raíz:

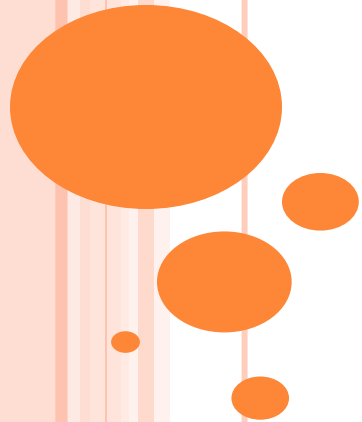
```
http://localhost:3000
```

➤ Se visualizará la respuesta generada por el controlador, que no es más que la página con un mensaje "Hello World!"

➤ Se puede modificar el puerto en main.js



CONTROLADOR



ESTRUCTURA Y FUNCIÓN

➤ El controlador se encarga de manejar las peticiones HTTP y generar respuestas

➤ Se define en una clase con la siguiente estructura:

Define la clase como un controlador y le asocia una dirección base

```
@Controller('libros')
export class LibrosController {
  @Post("alta")
  create(@Body() libro: LibroModel) {
  }
  @Get("catalogo")
  findAll() {
  }
  @Get("buscar/:id")
  findOne(@Param('id') id: string) {
  }
}
```

Mapea el cuerpo JSON a un objeto

Rutas específicas de cada recurso

PathVariable



RECOGIDA DE DATOS DE PETICIÓN

➤ **En los métodos del controlador se recogen los datos enviados en la petición a través de parámetros**

➤ **Estos datos pueden venir:**

- **Como Path variables (url/variable):**

```
@Get("buscar/:id")  
findOne(@Param('id') id: string) {..}
```

- **Como parámetros en querystring(url?param=value):**

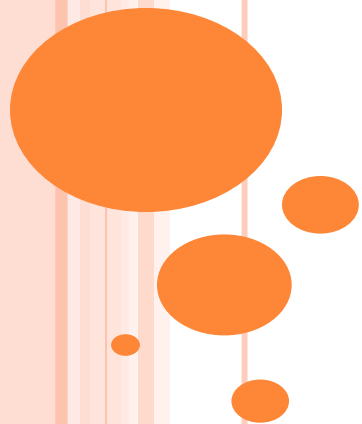
```
@delete("eliminar")  
deleteOne(@Query('id') id: string) {..}
```

- **Como JSON o Form url-encoded en el body:**

```
@post ("agregar")  
create(@Body() data: Persona) {..}
```



SERVICIO



ESTRUCTURA Y FUNCIÓN

- El servicio encapsula la lógica de negocio de la aplicación.
- Se define de forma similar a los servicios Angular:

```
@Injectable()
export class LibrosService {
  libros: LibroModel[]=[];
  create(libro: LibroModel):void {
    this.libros.push(libro);
  }

  findAll():LibroModel[] {
    return this.libros;
  }
  :
}
```

- Se inyecta en el controller a través del constructor.



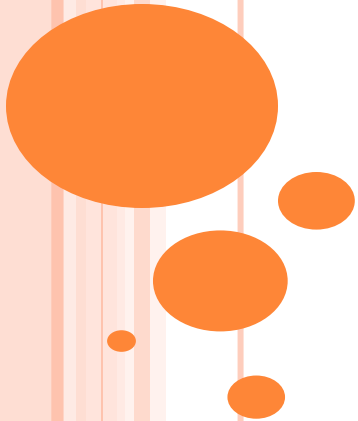
REGISTRO EN MÓDULO

➤ Para que puedan ser reconocidos por la aplicación, los servicios y controladores deben registrarse en el archivo de módulo AppModule:

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



ACCESO A BASES DE DATOS



TYPEORM

➤ **Librería para acceder a bases de datos relaciones desde TypeScript.**

➤ **Está basada en ORM.**

➤ **Para instalar TypeORM, junto con el controlador de MySQL:**

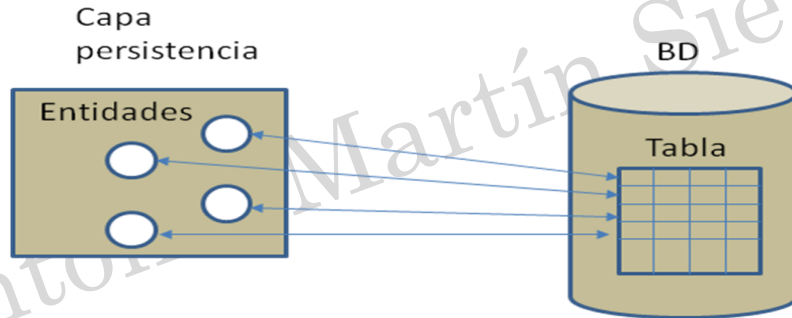
```
>npm install @nestjs/typeorm typeorm mysql2
```

➤ **Para MySQL, es necesario crear un usuario que use el plugin mysql_native_password, que es el compatible con TypeORM:**

```
CREATE USER 'nestuser'@'localhost' IDENTIFIED WITH mysql_native_password BY 'nestpass';  
GRANT ALL PRIVILEGES ON libros.* TO 'nestuser'@'localhost';  
FLUSH PRIVILEGES;
```

ENTIDADES

- El acceso a los datos se realiza a través de entidades.
- Una entidad es un objeto que representa una fila de una tabla de la base de datos:



CREACIÓN DE UNA ENTIDAD

➤ Se definen a través de una clase que encapsula los datos de la entidad.

➤ Se configura a través de una serie de decoradores:

campo clave
primaria

```
import {Entity, PrimaryColumn, Column,} from 'typeorm';
@Entity("libros")
export class LibroModel {
  → @PrimaryColumn()
    isbn:string;
  @Column()
    titulo:string;
  @Column()
    precio:number;
}
```



CONFIGURACIÓN DE LA CONEXIÓN

➤ Los datos de conexión se indican en el módulo de la aplicación en la sección de importaciones:

```
@Module({
  imports: [TypeOrmModule.forRoot({
    type: 'mysql',
    host: 'localhost',
    port: 3307,
    username: 'nestuser',
    password: 'nestpass',
    database: 'libros',
    entities: [LibroModel],
    synchronize: false,
  }), TypeOrmModule.forFeature([LibroModel])],
  controllers: [LibrosController],
  providers: [AppService],
})
```

a

También se
importa la
entidad



REPOSITORIO

➤ Para acceder a datos mediante ORM, el módulo TypeORM dispone del objeto Repository.

➤ Se debe inyectar en la capa service utilizando @InjectRepository:

```
constructor(@InjectRepository(LibroModel)  
    private readonly librosRepository: Repository<LibroModel>){  
  
}
```

➤ El objeto proporciona una serie de métodos para operar con entidades.



MÉTODOS DE REPOSITORY

- **save(entidad):Promise<Entidad>**. Guarda o actualiza la entidad en la base de datos
- **find():Promise<Entidad[]>**. Devuelve todas las entidades
- **findBy(where):Promise<Entidad[]>**. Recupera todas las entidades en función de una condición que se establece a través de un JSON:
- **findOneBy(where):Promise<Entidad>**. Igual que el anterior, pero devolviendo solo una entidad
- **remove(Entidad):Promise<Entidad>**. Elimina la entidad
- **delete(where)Promise<DeleteResult>**. Elimina las entidades que cumplen la condición



EJEMPLO SERVICE CON REPOSITORY

```
@Injectable()
export class AppService {
  constructor(@InjectRepository(LibroModel)
    private readonly librosRepository: Repository<LibroModel>){

  }
  create(libro: LibroModel):Promise<LibroModel> {
    return this.librosRepository.save(libro)
  }
  findAll():Promise<LibroModel[]> {
    return this.librosRepository.find();
  }
  findByIsbn(isbn: string) :Promise<LibroModel>{
    return this.librosRepository.findOneBy({isbn:isbn});
  }
  findByPrecioAndPaginas(precio:number, paginas:number) :Promise<LibroModel>{
    return this.librosRepository.findOneBy({where: {
      precio: LessThan(30),
      paginas: MoreThan(100),
    },});
  }
}
```