

# Integracion

## Carrera Programador full-stack

*Modelos de Persistencia*

Programación

Web

Programación  
básica

Front end

Programación  
orientada a  
objetos

Back end

Bases de  
datos

Integración

# Modelos de Persistencia

*¿Qué pasa con los datos de nuestra aplicación?*

Las aplicaciones trabajan con lo que se encuentra cargado en memoria RAM porque:

- La RAM es rápida
- El acceso a RAM es transparente

Pero:

- La RAM es limitada
- La RAM es volátil (una vez apagada la PC, se pierde el contenido que trabajamos en la RAM)

# Modelos de Persistencia

*¿Cómo guardar y recuperar el estado de la RAM?*

1. Utilizar medio de almacenamiento secundario
  - **Archivos planos** (como hicimos en el BackEnd)
  - **Bases de datos**
2. Se traduce a:
  - Manejo de archivos manual.
  - **Conexión directa a la base de datos**, mapeadores objeto-relacionales.
3. Existen otras opciones basadas en bases de datos No-SQL

# Mapeo Objeto Relacional (ORM)

- Los ORM tiene como objetivo abstraer al desarrollo de las cuestiones particulares de las bases de datos
- Transforman de forma automática los objetos a tuplas que se persisten en la base de datos cuando llega el momento de almacenarlos
- Transforman de forma automática las tuplas persistidas a objetos. De esta manera se puede operar con los datos
- Evitan el uso de SQL directo. Esto es una ventaja porque las diferentes bases de datos tiene diferencias en cómo está implementado SQL

# Paradigmas de Objetos y Relacional

	Programación Orientada a Objetos	Bases de Datos Relacionales
Elementos	Clases Objetos Atributos Métodos	Tablas Tuplas Atributos Triggers, Procedimientos, Funciones
Relaciones	Referencias en memoria	Foreign Keys
Identificación	Identificadores de Objetos	Primary Keys
Otros aspectos	Encapsulamiento, Herencia, Polimorfismo	Restricciones de Integridad, Transacciones
Diseño	Diagrama de Clases, secuencia	Diagramas de Entidad - Relación
Lenguajes	Java, C++, Python, Typescript,...	SQL

# Integracion

## Carrera Programador full-stack

*Conexión API-DB*

# Agenda

- Conexión con DB
- Uso de Queries Raw
  - Inconvenientes
- Asociación Clase-Tabla
- Operaciones sobre la DB
- Ejercicios



# Sistema Escolar

Basados en un modelo E-R que tiene las siguientes entidades y restricciones:

- . Estudiantes
- . Clases
- . Profesores
- . Escuelas
- . Ciudades
- . Asistencia

Los estudiantes pueden asistir a una o más clases de una misma escuela

Cada clase puede ser equivalente en diferentes escuelas

Un mismo profesor tiene asignadas muchas clases y en diferentes escuelas

Un estudiante se matricula en una o más clases de una misma escuela

Cada Estudiante puede tener más de un domicilio

Cada Profesor puede tener más de un domicilio

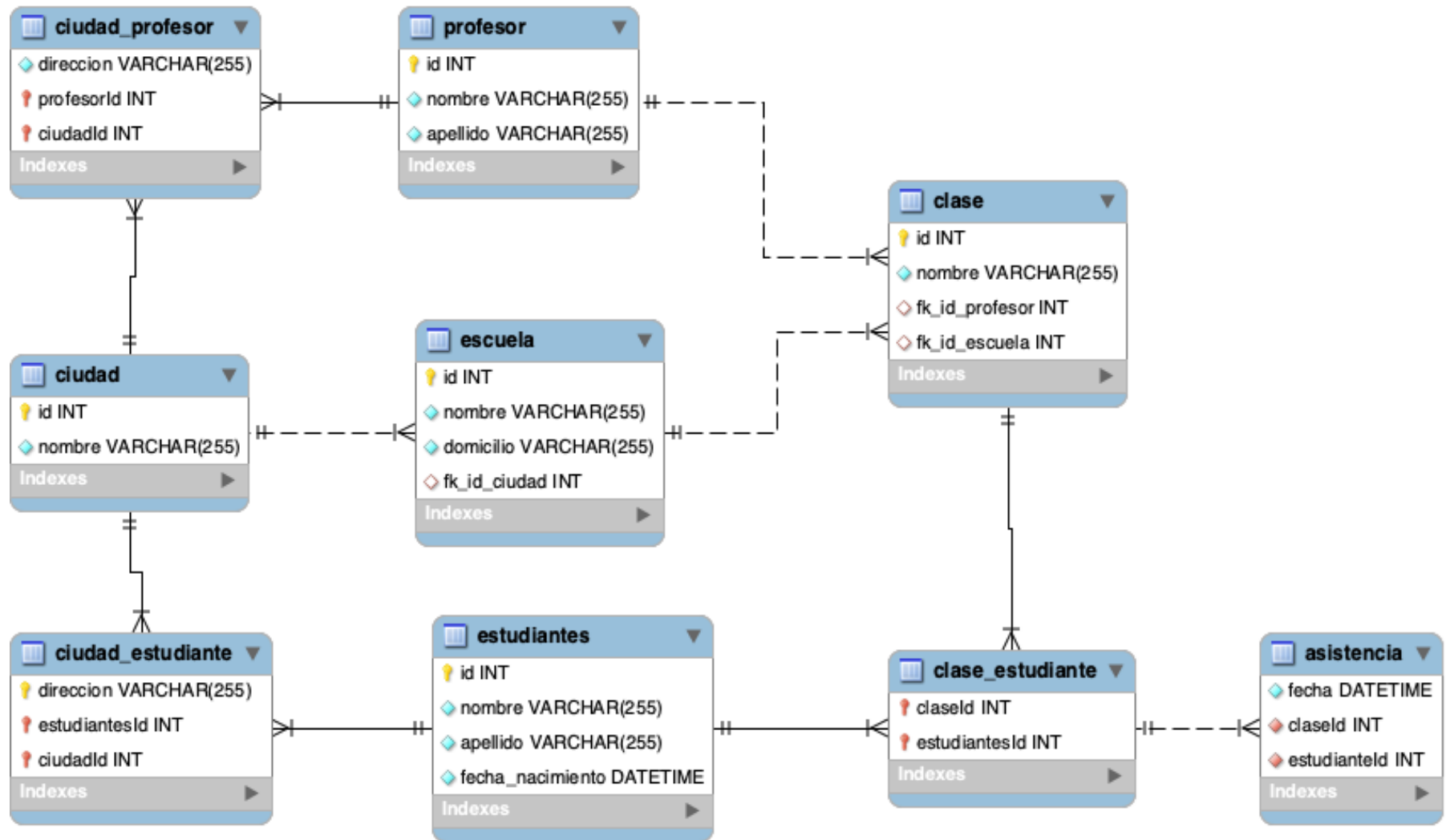
Cada Escuela tiene un único domicilio

Una ciudad puede tener más de una escuela

Se debe registrar la asistencia de los estudiantes a cada clase en las que están matriculados

Desarrollar un sistema que permita su administración.

# Esquema BD



# Comenzando...

- Usaremos NEST para gestionar nuestro BackEnd
- Crear proyecto NEST:

```
nest new pfs-escolar
```

- Entrar a la carpeta y commitear el código inicial:

```
cd pfs-escolar
```

```
git status
```

```
git add .
```

```
git status
```

```
git commit -m "Initial Project commit"
```

```
git remote add origin URL_DE_GITHUB
```

```
git push -u origin main
```

# Instalando TypeORM

- Necesitamos agregar el módulo de archivos estáticos

```
npm i --save @nestjs/serve-static
```

- La interacción entre nuestra API y MySQL la va a gestionar TypeORM

```
npm i --save @nestjs/typeorm typeorm mysql2
```

- Documentacion:

<https://docs.nestjs.com/techniques/database>

# Conexión con MySQL (1)

- Para conectarnos a MySQL necesitamos ciertos datos
  - Host
  - Puerto
  - Usuario
  - Contraseña
- Siempre que queramos conectarnos a una base de datos, vamos a necesitar este tipo de información
- En NestJS tenemos dos formas de dar esta data
  - En `app.module.ts` 👍
  - Haciendo manejo de conexiones

# Conexión con MySQL (2)

- Después, en el archivo app.module.ts
  - Importar **TypeOrmModule**
  - Agregar el módulo a los imports

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { CiudadModule } from './ciudad/ciudad.module';
...
@Module({
  imports: [
    TypeOrmModule.forRoot( ),
    CiudadModule,
  ],
  ...
})
export class AppModule { }
```

Aquí indicamos con qué base nos vamos a conectar:

```
{
  "type": "mysql",
  "host": "localhost",
  "port": 3306,
  "username": "root",
  "password": "*****",
  "database": "escolar",
  "entities": [
    "dist/**/**/*.entity{.ts,.js}"
  ],
  "synchronize": false
}
```

# Conexión con MySQL (3)

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { CiudadModule } from '../ciudad/ciudad.module';
```

```
...
@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: "mysql",
      host: "localhost",
      port: 3306,
      username: "root",
      password: "*****",
      database: "escolar",
      entities: [
        "dist/**/*.entity{.ts,.js}"
      ],
      "synchronize": false
    } ),
    CiudadModule,
  ],
  ...
})
export class AppModule { }
```

¿ Y esto ?

# Modules

- A medida que nuestra aplicación crece, se hace muy difícil mantener el código organizado
- Los módulos centralizan el código para cada funcionalidad.
- Permiten una fácil escalabilidad.
- Relacionan controladores con servicios

```
@Module({  
    controllers: [ CiudadController ],  
    providers: [ CiudadService ]  
})
```

```
export class CiudadModule { }
```



# Modificación de app.module.ts

- Aunque podemos simplificar todo utilizando el comando.

`nest g resource ciudad`

- Es una forma conveniente de crear rápidamente la estructura básica para un recurso en una aplicación NestJS, lo que ahorra tiempo y ayuda a mantener una organización consistente del código.

# Modificación de app.module.ts

- generará automáticamente un recurso llamado "ciudad". Este comando no solo creará el módulo, el servicio y el controlador para el recurso, sino que también generará un esqueleto de los métodos que se utilizarán dentro del controlador.
- Además, el comando también generará dos carpetas adicionales: Entity y DTO

# Asociación Clase-Tabla (1)

- El siguiente paso es asociar una clase con una tabla
- Cada clase puede verse como una tabla
- Cada variable de una clase, se puede asociar a una columna de la tabla
- Trabajamos con anotaciones
- Cada clase asociada a una tabla, la vamos a llamar *entity*

# Asociación Clase-Tabla (2)

```
import { Entity, PrimaryColumn, Column } from 'typeorm';
```

```
@Entity('ciudades')
```

```
export class Ciudad {
```

```
  @PrimaryColumn()
```

```
  private idCiudad : number;
```

```
  @Column()
```

```
  private nombre : string;
```

```
  constructor (id : number, nombre : string) {
```

```
    this.idCiudad = id;
```

```
    this.nombre = nombre;
```

```
  }
```

```
  public getIdCiudad(): number { return this.idCiudad; }
```

```
  public setIdCiudad(idCiudad: number): void { this.idCiudad = idCiudad; }
```

```
  public getNombre(): string { return this.nombre; }
```

```
  public setNombre(nombre: string): void { this.nombre = nombre; }
```

```
}
```

Nombre que va a tener la tabla que se cree/conecte al levantar la API

Definimos qué atributo será PK

ciudad.entity.ts

# Asociación Clase-Tabla (2)

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
```

```
@Entity('ciudades')
```

```
export class Ciudad {
```

```
  @PrimaryGeneratedColumn()
```

```
  private idCiudad : number;
```

```
  @Column()
```

```
  private nombre : string;
```

```
  constructor (nombre : string) {
```

```
    this.nombre = nombre;
```

```
  }
```

```
  public getIdCiudad(): number { return this.idCiudad; }
```

```
  public getNombre(): string { return this.nombre; }
```

```
  public setNombre(nombre: string): void { this.nombre = nombre; }
```

```
}
```

Nombre que va a tener la tabla que se cree/conecte al levantar la API

Definimos qué atributo será PK. Que será generada por la base

ciudad.entity.ts

# Importar Entity en el Módulo

- Además del controller y el service, hay que agregar las entities al módulo
- Se agregan todas las entities que va a utilizar el módulo

```
@Module({
  imports: [
    TypeOrmModule.forFeature( [
      Ciudad
    ]
  ],
  controllers: [ CiudadController ],
  providers: [ CiudadService ]
})
```

```
export class CiudadModule { }
```

```
ciudad.module.ts
```

# TypeORM - Repository

- La interacción con MySQL la hacemos a través de una variable de tipo Repository
  - La inyectamos en el constructor de nuestro servicio
- A través de esa variable hacemos todas las operaciones

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Ciudad } from './ciudad.entity';
```

```
@Injectable()
export class CiudadService {
  constructor (
    @InjectRepository(Ciudad)
    private readonly ciudadRepository : Repository<Ciudad> ) {}
  ...
}
```

Tener en cuenta que necesitamos pasar el tipo de la entity que armamos → Ciudad

ciudad.service.ts

# TypeORM - Consultas Raw

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Ciudad } from './ciudad.entity';
```

```
@Injectable()
export class CiudadService {
  private ciudades : Ciudad[] = [];

  constructor (@InjectRepository(Ciudad)
    private readonly ciudadRepository : Repository<Ciudad>) {}

  public async getAllRaw() : Promise<Ciudad[]> {
    let datos = await this.ciudadRepository.query("SELECT * FROM ciudades");

    datos.forEach(element => {
      let ciudad : Ciudad = new Ciudad(element['idCiudad'], element['nombre']);
      this.ciudades.push(ciudad);
    });
    return this.ciudades;
  }
}
```

Acá van las queries que  
venían haciendo directo  
sobre MySQL

ciudad.service.ts



# Consultas Raw - Inconvenientes

- Si bien es un recurso válido escribir las queries directo, tiene sus inconvenientes
- El ejemplo anterior parece fácil porque es una sola tabla
  - ¿Qué pasaría si queremos hacer una query que haga un JOIN entre varias tablas?
  - Leer a mano el resultado de ese tipo de queries no suele ser una tarea muy agradable que digamos
- Suponer el caso del INSERT o el UPDATE
  - Habría que intercalar todos los valores necesarios en el string que define a la query 😞

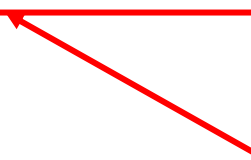
# ORM: Object Relational Mapping

Los ORM hacen un mapeo entre

Modelo de Objetos <-> Modelo Relacional

La idea es que podamos pensar en términos de objetos y preocuparnos lo menos posible por cómo se almacenan en la tabla

```
let ciudades : Ciudad[] = await this.ciudadRepository.find();
```



Crea un arreglo de objetos de la clase Ciudad con la respuesta de la BD

# Type ORM - Get All

- Usamos el controller.
- Reemplazando el mock por llamada a la BD.

```
import { Injectable } from '@nestjs/common';  
import { InjectRepository } from '@nestjs/typeorm';  
import { Repository } from 'typeorm';  
import { Ciudad } from './ciudad.entity';
```

```
@Injectable()  
export class CiudadService {  
  private ciudades : Ciudad[] = [];  
  
  constructor (@InjectRepository(Ciudad)  
    private readonly ciudadRepository : Repository<Ciudad>) {}  
  ...  
  public async getAll() : Promise<Ciudad[]> {  
    let ciudades: Ciudad[] = await this.ciudadRepository.find( );  
    return this.ciudades;  
  }  
  ...  
}
```

**ciudad.service.ts**

# Type ORM - Get By Id

- Usamos el controller con parámetros.
- Reemplazando el mock por llamada a la BD.

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { FindOneOptions, Repository } from 'typeorm';
import { Ciudad } from './ciudad.entity';
```

```
@Injectable()
export class CiudadService {
  private ciudades : Ciudad[] = [];

  constructor (@InjectRepository(Ciudad)
    private readonly ciudadRepository : Repository<Ciudad>) {}

  ...
  public async getById(id : number) : Promise<Ciudad> {
    const criterio : FindOneOptions = { where: { idCiudad: id } }
    let ciudad : Ciudad = await this.ciudadRepository.findOne( criterio );
    if (ciudad)
      return ciudad;
  }
  ...
}
```

**ciudad.service.ts**

**Crea un objeto de la clase Ciudad con la respuesta de la BD  
segun un criterio dado**

# Manejo de errores

- Retornando estados HTTP
  - Por defecto nest devuelve un error **500** cuando el servidor falla de alguna manera
  - Ahora lo que vamos a hacer es personalizar estos errores y devolverlos en forma HTTP status para un mejor control de nuestra aplicación

Podemos lanzar un error así:

```
throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
```

Para que nos devuelva una respuesta así:

```
{ "statusCode": 403, "message": "Forbidden" }
```

O podemos personalizar un error así:

```
throw new HttpException({status: HttpStatus.FORBIDDEN, error: 'This is a custom message'}, HttpStatus.FORBIDDEN);
```

Para obtener una respuesta así:

```
{ "status": 403, "error": "This is a custom message" }
```

# Type ORM - Control de Errores

```
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Ciudad } from './ciudad.entity';
```

ciudad.service.ts

```
@Injectable()
export class CiudadService {
  private ciudades : Ciudad[] = [];

  constructor (@InjectRepository(Ciudad)
    private readonly ciudadRepository : Repository<Ciudad>) {}

  ...

  public async getByld(id : number) : Promise<Ciudad> {
    try {
      const criterio : FindOneOptions = { where: { idCiudad: id } }
      let ciudad : Ciudad = await this.ciudadRepository.findOne( criterio );
      if (ciudad) return ciudad;
      throw new Exception( 'La ciudad no se encuentra' );
    } catch (error) {
      throw new HTTPException( { status : HttpStatus.NOT_FOUND,
        error : 'Error en la busqueda de ciudad '+id+ ' : '+error},
        HttpStatus.NOT_FOUND);
    }
  }
  ...
}
```

# Integracion

## Carrera Programador full-stack

*Ejercicios*