

**Autori:** Daniele Berardinell, Emanuele Bucciarelli

**Data:** 31/03/2025

# Analisi della realtà - Gestione scuola media

## Requisiti

Realizzare un'applicazione console in C# che interagisce con un API per ottenere e visualizzare dati relativi ai temi proposti.

Per prima cosa, creare il Modello E/R relativo al proprio argomento, poi creare il modello Logico di esso ed infine il Modello Fisico. Popolare il DB appena creato e creare le query assegnate.

Una volta sviluppato il Database, creare l'applicazione in C# che interagisce con un API, implementare le funzionalità CRUD per tutte le tabelle ipotizzate del DB e delle Query.

L'obiettivo è dunque creare un'applicazione console in C# che interagisca con le API REST, i passaggi per farlo sono:

1. Creare il modello E/R
2. Creare lo schema logico
3. Popolare il database e generare le query
4. Sviluppare l'applicazione
5. Implementare le funzionalità CRUD (Create, Read, Update e Delete)

## Contesto

Si considerino i seguenti fatti di interesse di una scuola media. Insegnanti: un insegnante è identificato dal codice fiscale; di ogni insegnante interessa il cognome, il nome, le materie d'insegnamento, le classi in cui le insegna (supponiamo che un insegnante possa insegnare materie diverse in classi diverse, ad es. Italiano in una classe e Storia e Geografia in un'altra classe). Studenti: uno studente è identificato da cognome, nome, di ogni studente interessa inoltre il luogo di nascita, la data di nascita, la classe che frequenta. Classi: una classe è identificata da un numero (1, 2 o 3) e dalla sezione; di ogni classe interessa inoltre il numero di studenti che la frequentano, gli insegnanti che vi insegnano, gli studenti che la frequentano

Il progetto si concentra sulla gestione dei dati di una scuola media, includendo informazioni su insegnanti, studenti e classi

## Entità considerate:

- Insegnante
- Studente
- Classe
- Insegnamento

## Relazione Ternaria "Insegnamento"

La tabella `Insegnamento` rappresenta una relazione ternaria tra `Insegnante`, `Classe` e `Materia`. Questa relazione è necessaria per modellare correttamente lo scenario scolastico, in cui:

- Un **insegnante** può insegnare più materie
- Un **insegnante** può insegnare a più classi
- Una **classe** può avere più insegnanti
- Ogni insegnamento è identificato dalla combinazione insegnante-classe-materia

La chiave primaria composta (`ID_Insegnante`, `ID_Classe`, `ID_Materia`) indica che un insegnante non possa essere registrato due volte per la stessa materia nella stessa classe. Questa struttura permette di tracciare con precisione quali insegnanti insegnano quali materie e a quali classi

## Struttura delle Tabelle

### Tabella Classe

Nome Variabile	Tipo Dato	NOT NULL
ID_Classe	INT	<input checked="" type="checkbox"/>
numero	TINYINT	<input checked="" type="checkbox"/>
sezione	VARCHAR(10)	<input checked="" type="checkbox"/>

### Tabella Insegnante

Nome Variabile	Tipo Dato	NOT NULL
ID_Insegnante	INT	<input checked="" type="checkbox"/>
cod_fiscale	VARCHAR(255)	<input checked="" type="checkbox"/>
nome	VARCHAR(255)	<input checked="" type="checkbox"/>

Nome Variabile	Tipo Dato	NOT NULL
cognome	VARCHAR(255)	✓
data_nascita	DATETIME	✓

## Tabella Materia

Nome Variabile	Tipo Dato	NOT NULL
ID_Materia	INT	✓
nome	VARCHAR(255)	✓

## Tabella Studente

Nome Variabile	Tipo Dato	NOT NULL
ID_Studente	INT	✓
nome	VARCHAR(255)	✓
cognome	VARCHAR(255)	✓
data_nascita	DATETIME	✓
luogo_nascita	VARCHAR(255)	✓
ID_Classe	INT	✓

## Tabella Insegnamento (Relazione Ternaria)

Nome Variabile	Tipo Dato	NOT NULL
ID_Insegnante	INT	✓
ID_Classe	INT	✓
ID_Materia	INT	✓

## Scelte progettuali

Per la tabella Insegnante, sebbene il codice fiscale ( `cod_fiscale` ) sia univoco, non è stato usato come chiave primaria perché:

- È una stringa alfanumerica, quindi meno efficiente da gestire come chiave primaria rispetto a un intero ( `INT` )

- Un ID numerico garantisce prestazioni migliori in join e ricerche
- Potrebbero esistere casi in cui l'insegnante non ha ancora un codice fiscale registrato, e un ID autoincrementato semplifica la gestione

#### Uso di `TINYINT` per il numero di classe:

- Il numero di classe può assumere solo valori limitati (tipicamente 1, 2, 3 per la scuola media)
- `TINYINT` è più efficiente rispetto a `INT` in termini di spazio occupato nel database
- Ottimizza le prestazioni riducendo il consumo di memoria

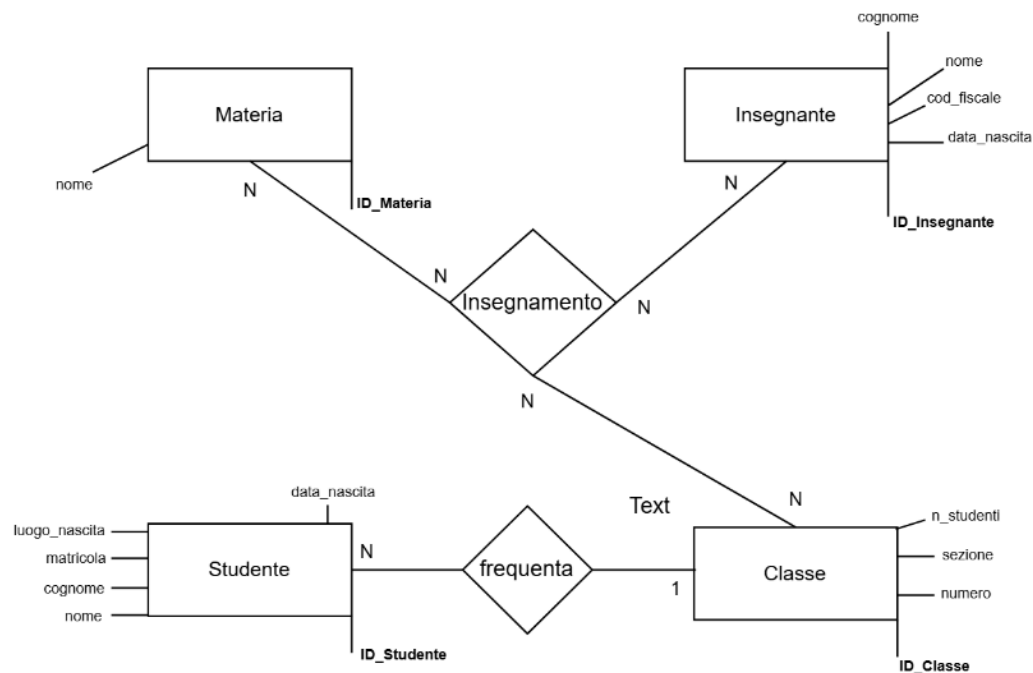
#### Utilizzo di `**DATETIME**` invece di `**DateOnly**`:

- `DateOnly` è una funzionalità recente di .NET e non è supportata direttamente da molte librerie ORM, tra cui Dapper
- Dapper gestisce meglio `DATETIME` poiché è un tipo di dato standard SQL Server
- Evita conversioni manuali tra `DateOnly` e `DateTime`, riducendo il rischio di errori

#### Handler

La conversione da `DateTime` a `DateOnly` avviene nel codice attraverso un handler, spiegato successivamente

## Schema E/R



## Relazioni

### Relazione: Insegnante - Insegnamento

- Un **insegnante** può insegnare più materie in più classi (Relazione 1:N)
- Una **materia** può essere insegnata da più insegnanti in più classi
- Una **classe** può avere più insegnanti per diverse materie
- **Tabella di riferimento:** Insegnamento (Relazione **Ternaria** N:N:N)

### Relazione: Studente - Classe

- Uno **studente** è assegnato a una sola classe (Relazione **N:1**)
- Una **classe** può avere più studenti iscritti
- **Tabella di riferimento:** Studente (ID\_Classe come FK)

### Relazione: Classe - Insegnante

- Una **classe** può avere più insegnanti per materie diverse (Relazione N:N attraverso Insegnamento )
- Un **insegnante** può insegnare a più classi
- **Tabella di riferimento:** Insegnamento

### Relazione: Classe - Materia

- Una **materia** può essere insegnata in più classi (Relazione N:N attraverso Insegnamento )
- Una **classe** può avere più materie assegnate
- **Tabella di riferimento:** Insegnamento

## Relazione: Insegnante - Materia

- Un **insegnante** può insegnare più materie (Relazione N:N attraverso Insegnamento )
- Una **materia** può essere insegnata da più insegnanti
- **Tabella di riferimento:** Insegnamento

## Schema logico

Studente(ID\_Studente(PK), data\_nascita, luogo\_nascita, cognome, nome, ID\_Classe(FK))

Classe(ID\_Classe(PK), numero, sezione)

Insegnante(ID\_Insegnante(PK), cod\_fiscale, nome, cognome, data\_nascita)

Materia(ID\_Materia(PK), nome)

Insegnamento(ID\_Insegnante(FK), ID\_Classe(FK), ID\_Materia(FK))

## Create table

Di seguito la creazione delle tabelle con SQL:

```
CREATE TABLE Classe
(
  ID_Classe INT PRIMARY KEY IDENTITY NOT NULL,
  numero TINYINT NOT NULL,
  sezione VARCHAR(10) NOT NULL
)

CREATE TABLE Insegnante
(
  ID_Insegnante INT PRIMARY KEY IDENTITY NOT NULL,
  cod_fiscale VARCHAR(255) NOT NULL,
  nome VARCHAR(255) NOT NULL,
  cognome VARCHAR(255) NOT NULL,
  data_nascita DATE NOT NULL
)

CREATE TABLE Materia
```

```

(
  ID_Materia INT PRIMARY KEY IDENTITY NOT NULL,
  nome VARCHAR(255) NOT NULL
)

CREATE TABLE Studente
(
  ID_Studente INT PRIMARY KEY IDENTITY NOT NULL,
  nome VARCHAR(255) NOT NULL,
  cognome VARCHAR(255) NOT NULL,
  data_nascita DATE NOT NULL,
  luogo_nascita VARCHAR(255) NOT NULL,
  ID_Classe INT NOT NULL FOREIGN KEY REFERENCES Classe(ID_Classe)
)

CREATE TABLE Insegnamento
(
  ID_Insegnante INT NOT NULL,
  ID_Classe INT NOT NULL,
  ID_Materia INT NOT NULL,
  PRIMARY KEY (ID_Insegnante, ID_Classe, ID_Materia),
  FOREIGN KEY (ID_Insegnante) REFERENCES Insegnante(ID_Insegnante),
  FOREIGN KEY (ID_Classe) REFERENCES Classe(ID_Classe),
  FOREIGN KEY (ID_Materia) REFERENCES Materia(ID_Materia)
)

```

## Popolamento

```

-- inserimento per le classi
INSERT INTO Classe (numero, sezione) VALUES (1, 'A');
INSERT INTO Classe (numero, sezione) VALUES (2, 'B');

-- inserimento per gli insegnanti
INSERT INTO Insegnante (cod_fiscale, nome, cognome, data_nascita) VALUES
('MRCMRC06M26I608Q', 'Marco', 'Mudric', '2006-08-26'),
('BRRDNL06E24I608Y', 'Daniele', 'Berardinelli', '2006-05-24');

-- inserimento di materie
INSERT INTO Materia (nome) VALUES ('Matematica');
INSERT INTO Materia (nome) VALUES ('Informatica');
INSERT INTO Materia (nome) VALUES ('Italiano');

-- inserimento di studenti
INSERT INTO Studente (nome, cognome, data_nascita, luogo_nascita, ID_Classe)
VALUES

```

```

('Giampaolo', 'Magi', '2006-09-15', 'Fano', 1),
('Claudio', 'Lenci', '2006-05-24', 'Senigallia', 1),
('Giacomo', 'Principi', '2006-11-05', 'Montesicuro', 2);

-- la relazione ternaria
INSERT INTO Insegnamento (ID_Insegnante, ID_Classe, ID_Materia) VALUES
(1, 1, 1), -- Marco Mudric insegna matematica alla classe 1A
(1, 1, 3), -- Marco Mudric insegna italiano alla classe 1A
(2, 2, 2), -- Daniele Berardinelli insegna informatica alla classe 2B
(2, 1, 2); -- Daniele Berardinelli insegna informatica anche alla classe 1A

```

## Query

### Query 1

Recuperare tutti gli studenti che frequentano una specifica classe, identificata dal numero e dalla sezione.

```

SELECT * FROM Studente
INNER JOIN Classe ON Studente.ID_Classe = Classe.ID_Classe
WHERE Classe.numero = X AND Classe.sezione = 'Y'; -- ovviamente X è il numero
della classe e 'Y' fa riferimento alla sezione, questa è una generalizzazione
che nel programma vengono inseriti dall'utente

```

### Spiegazione:

- Cerca nella tabella `Studente` e combina ( `INNER JOIN` ) con `Classe` tramite l'ID della classe
- Filtra ( `WHERE` ) per:
  - `numero` (es. 1 per prima, 2 per seconda...)
  - `sezione` (es. 'A', 'B'...)
- Restituisce tutti i campi ( `*` ) degli studenti corrispondenti

### Query 2

Elencare tutti gli insegnanti che insegnano in una determinata classe.

```

SELECT DISTINCT * FROM Insegnante
INNER JOIN Insegnamento ON Insegnante.ID_Insegnante =
Insegnamento.ID_Insegnante
WHERE Insegnamento.ID_Classe = (SELECT ID_Classe FROM Classe WHERE numero = x
and sezione = 'Y');

```



### Spiegazione:

1. Prima trova l'ID della classe con una sottoquery
2. Poi cerca nella tabella `Insegnamento` gli insegnanti associati a quell'ID
3. `DISTINCT` evita duplicati se un insegnante ha più materie nella stessa classe
4. Combina con i dati completi degli insegnanti

### Query 3

Trovare tutte le materie insegnate da un determinato insegnante e le classi in cui le insegna.

```
SELECT Materia.nome AS materia, Classe.numero, Classe.sezione
FROM Insegnamento
JOIN Materia ON Insegnamento.ID_Materia = Materia.ID_Materia
JOIN Classe ON Insegnamento.ID_Classe = Classe.ID_Classe
WHERE Insegnamento.ID_Insegnante = 1;
```

### Spiegazione:

- Parte dall'ID insegnante (es. 1)
- Combina tre tabelle per ottenere:
  - Nome materia (da `Materia`)
  - Numero e sezione classe (da `Classe`)
- Restituisce solo le colonne specificate

### Query 4

Individuare il numero totale di studenti per ciascuna classe, ordinando il risultato in ordine decrescente.

```
SELECT Classe.numero, Classe.sezione, COUNT(Studente.ID_Studente) AS
num_studenti FROM Classe
LEFT JOIN Studente ON Classe.ID_Classe = Studente.ID_Classe
GROUP BY Classe.ID_Classe, Classe.numero, Classe.sezione
ORDER BY num_studenti DESC;
```

### Spiegazione:

- `LEFT JOIN` include anche classi senza studenti
- `COUNT` calcola il numero di studenti per classe
- `GROUP BY` raggruppa per classe

- `ORDER BY DESC` ordina dalla classe più numerosa

## Query 5

Contare quanti insegnanti insegnano in più di una classe e restituire il numero totale di classi in cui ognuno insegna.

```
SELECT Insegnante.cod_fiscale, Insegnante.nome, Insegnante.cognome,  
COUNT(DISTINCT Insegnamento.ID_Classe) AS num_classi  
FROM Insegnante  
INNER JOIN Insegnamento ON Insegnante.ID_Insegnante =  
Insegnamento.ID_Insegnante  
GROUP BY Insegnante.ID_Insegnante, Insegnante.cod_fiscale, Insegnante.nome,  
Insegnante.cognome  
HAVING COUNT(DISTINCT Insegnamento.ID_Classe) > 1;
```

### Spiegazione:

- `COUNT(DISTINCT ...)` conta classi diverse (ignorando duplicati)
- `HAVING` filtra solo chi ha >1 classe
- Restituisce CF, nome completo e numero classi

## Documentazione del codice

### Architettura del Progetto

Il progetto consiste in 3 livelli:

#### 1. Models ( Progetto\_Scuola.Models )

Rappresentano le entità del dominio:

- `Classe.cs`
- `Studiante.cs`
- `Insegnante.cs`
- `Materia.cs`
- `Insegnamento.cs` (relazione ternaria)

`Classe.cs`:

```
public class Classe  
{  
    public int ID_Classe { get; set; } # Primary key della tabella classe
```

```

    public int Numero { get; set; } # numero che identifica la classe
    public string Sezione { get; set; } # sezione che identifica la classe
}

```

Insegnamento.cs:

```

namespace Progetto_Scuola.Models
{
    public class Insegnamento
    {
        public int ID_Insegnante { get; set; }
        public int ID_Classe { get; set; }
        public int ID_Materia { get; set; }
    }
}

```

La classe "Insegnamento" contiene la relazione ternaria, dunque le primary key delle 3 entità in gioco.

Insegnante.cs:

```

namespace Progetto_Scuola.Models
{
    public class Insegnante
    {
        public int ID_Insegnante { get; set; } # primary key di insegnante
        public string Cod_Fiscale { get; set; } # codice fiscale
        dell'insegnante
        public string Nome { get; set; }
        public string Cognome { get; set; }
        public DateTime Data_Nascita { get; set; } # settato a DateTime e poi
        gestito nell'handler
    }
}

```

Materia.cs:

```

namespace Progetto_Scuola.Models
{
    public class Materia
    {
        public int ID_Materia { get; set; } # primary key della tabella
        Materia
        public string Nome { get; set; } # nome della materia
    }
}

```

```
}  
}
```

Studente.cs:

```
namespace Progetto_Scuola.Models  
{  
    public class Studente  
    {  
        public int ID_Studente { get; set; } # primary key di Studente  
        public string Nome { get; set; }  
        public string Cognome { get; set; }  
        public DateTime Data_Nascita { get; set; } # modificato dall'handler  
        public string Luogo_Nascita { get; set; } # luogo di nascita dello  
        studente  
        public int ID_Classe { get; set; } # foreign key (1:N)  
    }  
}
```

## 2. Services (📁 Progetto\_Scuola.Services )

📁 Entità	📄 File Service	🔑 Funzionalità
<b>Classe</b>	ClasseService.cs	CRUD + Query 1 (studenti in classe) + Query 2 (insegnanti per classe) + Query 4 (numero studenti per classe)
<b>Studente</b>	StudenteService.cs	CRUD completo (inserimento, lettura, modifica, eliminazione)
<b>Insegnante</b>	InsegnanteService.cs	CRUD + Query 3 (materie e classi insegnate) + Query 5 (insegnanti in più classi)
<b>Materia</b>	MateriaService.cs	CRUD semplice su materia
<b>Insegnamento</b>	InsegnamentoService.cs	CRUD con chiave composita (no update)

### 🔗 Dettaglio ClasseService.cs

📄 Responsabilità: gestione CRUD della tabella `Classe` + query su studenti e insegnanti collegati

### ✅ Costruttore con Dependency Injection

```
private readonly SqlConnection db;

public ClasseService(SqlConnection _db)
{
    db = _db;
}
```

❗ Inietta la connessione SQL tramite il costruttore, in modo da poterla riutilizzare per ogni metodo (gestito dal `Program.cs` )

## GetAll()

```
GetAll()

public List<Classe> GetAll()
{
    return db.Query<Classe>("SELECT * FROM Classe").ToList();
}
```



## GetByID(int id)

```
public Classe GetByID(int id)
{
    return db.QuerySingleOrDefault<Classe>("SELECT * FROM Classe WHERE
ID_Classe = @id", new { id });
}
```



Ricerca una singola classe tramite ID.



Se non trova nulla, ritorna `null`



## Add(Classe c)


```
public void Add(Classe c)
{
    db.Execute("INSERT INTO Classe (numero, sezione) VALUES (@Numero,
@Sezione)", c);
}
```

✚ Inserisce una nuova classe nel database.  
I valori vengono presi direttamente dall'oggetto `Classe`

---

## Update(`Classe c`)


```
public void Update(Classe c)
{
    db.Execute("UPDATE Classe SET numero = @Numero, sezione = @Sezione WHERE ID_Classe = @ID_Classe", c);
}
```

 Aggiorna il numero e la sezione di una classe esistente identificata da `ID_Classe`

---

## Delete(`int id`)

```
public void Delete(int id)
{
    db.Execute("DELETE FROM Classe WHERE ID_Classe = @id", new { id });
}
```

 Rimuove una classe dal database in base all'ID

---

## Query 1 – `GetStudentiInClasse(int numero, string sezione)`

```
public List<Studente> GetStudentiInClasse(int numero, string sezione)
{
    return db.Query<Studente>(@"
        SELECT * FROM Studente
        INNER JOIN Classe ON Studente.ID_Classe = Classe.ID_Classe
        WHERE Classe.numero = @numero AND Classe.sezione = @sezione",
        new { numero, sezione }).ToList();
}
```

- 📌 Recupera tutti gli studenti appartenenti a una specifica classe usando `INNER JOIN`.
- ✅ Utile per query dinamiche, i parametri vengono passati dall'utente

## Query 2 – `GetInsegnantiInClasse(int numero, string sezione)`

```
public List<Insegnante> GetInsegnantiInClasse(int numero, string sezione)
{
    return db.Query<Insegnante>(@"
        SELECT DISTINCT i.* FROM Insegnante i
        INNER JOIN Insegnamento ins ON i.ID_Insegnante = ins.ID_Insegnante
        INNER JOIN Classe c ON ins.ID_Classe = c.ID_Classe
        WHERE c.numero = @numero AND c.sezione = @sezione",
        new { numero, sezione }).ToList();
}
```

📖 Usa due join:

- tra `Insegnante` e `Insegnamento`
  - tra `Insegnamento` e `Classe`
- `DISTINCT` serve per evitare insegnanti duplicati (es. stessi insegnanti con più materie)

## Query 4 – `GetNumeroStudentiPerClasse()`

```
public List<dynamic> GetNumeroStudentiPerClasse()
{
    return db.Query(@"
        SELECT c.numero, c.sezione, COUNT(s.ID_Studente) AS num_studenti
        FROM Classe c
        LEFT JOIN Studente s ON c.ID_Classe = s.ID_Classe
        GROUP BY c.numero, c.sezione
        ORDER BY num_studenti DESC").ToList();
}
```

- 📊 Aggrega i dati per contare quanti studenti sono presenti in ciascuna classe
- ⚠️ `LEFT JOIN` permette di includere anche le classi **senza studenti iscritti**



# Dettaglio InsegnanteService.cs

 Responsabilità: gestione insegnanti (CRUD) + **query avanzate su materie e classi**

---



## Costruttore e connessione DB

```
private readonly SqlConnection db;

public InsegnanteService(SqlConnection _db)
{
    db = _db;
}
```


**i** Riceve la connessione tramite dependency injection, garantendo separazione della logica e riutilizzabilità

---



## GetAll()

```
public List<Insegnante> GetAll()
{
    return db.Query<Insegnante>("SELECT * FROM Insegnante").ToList();
}
```


 Ritorna tutti gli insegnanti dal DB

---



## GetByID(int id)

```
public Insegnante GetByID(int id)
{
    return db.QuerySingleOrDefault<Insegnante>("SELECT * FROM Insegnante WHERE ID_Insegnante = @id", new { id });
}
```

 Cerca un insegnante tramite il suo ID


 Se non trovato, ritorna `null`



---

## Add(Insegnante i)

```
public void Add(Insegnante i)
{
    db.Execute("INSERT INTO Insegnante (cod_fiscale, nome, cognome,
data_nascita) VALUES (@Cod_Fiscale, @Nome, @Cognome, @Data_Nascita)", i);
}
```

 Inserisce un nuovo insegnante

---

## Update(Insegnante i)

```
public void Update(Insegnante i)
{
    db.Execute("UPDATE Insegnante SET cod_fiscale = @Cod_Fiscale, nome =
@Nome, cognome = @Cognome, data_nascita = @Data_Nascita WHERE ID_Insegnante =
@ID_Insegnante", i);
}
```

 Modifica i dati di un insegnante esistente

---

## Delete(int id)

```
public void Delete(int id)
{
    db.Execute("DELETE FROM Insegnante WHERE ID_Insegnante = @id", new { id
});
}
```

 Elimina un insegnante dal DB

---

## Query 3 – GetMaterieEClassi(int idInsegnante)

```

public List<dynamic> GetMaterieEClassi(int idInsegnante)
{
    return db.Query(@"
        SELECT m.nome AS Materia, c.numero AS NumeroClasse, c.sezione AS
Sezione
        FROM Insegnamento i
        JOIN Materia m ON i.ID_Materia = m.ID_Materia
        JOIN Classe c ON i.ID_Classe = c.ID_Classe
        WHERE i.ID_Insegnante = @id",
        new { id = idInsegnante }).ToList();
}

```

📌 Questa query serve per mostrare tutte le **materie insegnate** da un determinato insegnante e **in quali classi** le insegna

🔗 Utilizza due join: con `Materia` e con `Classe`

#### 📄 Esempio output:

```

[
  { "Materia": "Matematica", "NumeroClasse": 2, "Sezione": "B" },
  { "Materia": "Informatica", "NumeroClasse": 1, "Sezione": "A" }
]

```

## 📊 Query 5 – GetInsegnantiInPiuClassi()

```

public List<dynamic> GetInsegnantiInPiuClassi()
{
    return db.Query(@"
        SELECT ins.cod_fiscale, ins.nome, ins.cognome, COUNT(DISTINCT
i.ID_Classe) AS NumClassi
        FROM Insegnante ins
        JOIN Insegnamento i ON ins.ID_Insegnante = i.ID_Insegnante
        GROUP BY ins.cod_fiscale, ins.nome, ins.cognome
        HAVING COUNT(DISTINCT i.ID_Classe) > 1").ToList();
}

```

📊 Restituisce tutti gli insegnanti che insegnano in più di una classe


Usa `COUNT(DISTINCT ...)` + `HAVING` per il filtro

#### 📄 Esempio output:

```
[
  { "cod_fiscale": "BRRDNL06E24I608Y", "nome": "Daniele", "cognome":
    "Berardinelli", "NumClassi": 2 }
]
```



## Dettaglio `ClasseController.cs`

 Responsabilità: gestisce le operazioni API sulla tabella `Classe`, inclusa la logica per studenti/insegnanti collegati.



### Costruttore con injection

```
private readonly ClasseService service;


public ClasseController(SqlConnection db)
{
    service = new ClasseService(db);
}
```

**i** Crea il service `ClasseService` con una connessione SQL iniettata



### [GET] `GetAll()`


```
[HttpGet]
public ActionResult<List<Classe>> GetAll()
{
    return service.GetAll();
}
```

 Ritorna l'elenco di tutte le classi.



### [GET] `GetByID(int id)`

```
[HttpGet("{id}")]
public ActionResult<Classe> GetByID(int id)
{
    var classe = service.GetByID(id);
    return classe == null ? NotFound() : Ok(classe);
}
```

 Cerca una classe per ID. Se non trovata → 404


## **[POST]** Add()

```
[HttpPost]
public ActionResult Add([FromBody] Classe c)
{
    service.Add(c);
    return Ok();
}
```

 Aggiunge una nuova classe. Il JSON viene deserializzato automaticamente


## **[PUT]** Update()

```
[HttpPut]
public ActionResult Update([FromBody] Classe c)
{
    service.Update(c);
    return Ok();
}
```

 Modifica una classe esistente

## **[DELETE]** Delete(int id)

```
[HttpDelete("{id}")]
public ActionResult Delete(int id)
{
    service.Delete(id);
    return Ok();
}
```

 Elimina una classe

## **Query 2 – Insegnanti in una classe**


```
[HttpGet("insegnanti")]
public ActionResult<List<Insegnante>> GetInsegnantiInClasse(int numero, string
sezione)
{
```

```
return service.GetInsegnantiInClasse(numero, sezione);  
}
```


📌 Esempio: GET /Classe/insegnanti?numero=1&sezione=A  
Ritorna tutti gli insegnanti della classe specificata

## Query 4 – Studenti per classe (statistica)

```
[HttpGet("statistiche/studenti-per-classe")]  
public ActionResult GetNumeroStudentiPerClasse()  
{  
    return Ok(service.GetNumeroStudentiPerClasse());  
}
```

 Esempio: GET /Classe/statistiche/studenti-per-classe  
Ritorna un array con numero, sezione e conteggio degli studenti

## Dettaglio **InsegnanteController.cs**

 Responsabilità: espone le API REST per la gestione degli **insegnanti**, con operazioni **CRUD** e **query avanzate**.

## Costruttore con Dependency Injection

```
private readonly InsegnanteService service;
```

```
public InsegnanteController(SqlConnection db)  
{  
    service = new InsegnanteService(db);  
}
```

❗ L'oggetto `SqlConnection` è iniettato dal framework e passato al `Service`, che gestisce tutte le query con Dapper

## [GET] `GetAll()`

```
[HttpGet]
public ActionResult<List<Insegnante>> GetAll()
{
    return service.GetAll();
}
```

✅ Restituisce la lista completa degli insegnanti presenti nel database

## [GET] GetById(int id)

```
[HttpGet("{id}")]
public ActionResult<Insegnante> GetById(int id)
{
    var insegnante = service.GetById(id);
    return insegnante == null ? NotFound() : Ok(insegnante);
}
```

 Cerca un insegnante tramite il suo ID\_Insegnante

⚠️ Se non esiste, restituisce 404 Not Found

## [POST] Add()

```
[HttpPost]
public ActionResult Add([FromBody] Insegnante i)
{
    service.Add(i);
    return Ok();
}
```

✅ Inserisce un nuovo insegnante.

I dati devono essere passati in formato JSON nel body della richiesta


### Esempio JSON

```
{
  "Cod_Fiscale": "RSSMRA80A01H501Z",
  "Nome": "Maria",
  "Cognome": "Rossi",
}
```

```
"Data_Nascita": "1980-01-01"
}
```

## [PUT] Update()

```
[HttpPut]
public ActionResult Update([FromBody] Insegnante i)
{
    service.Update(i);
    return Ok();
}
...
```

- >  Modifica i dati di un insegnante già esistente
- > È necessario includere anche `ID\_Insegnante` nel JSON

---

##  [DELETE] `Delete(int id)`

```
```c#
[HttpDelete("{id}")]
public ActionResult Delete(int id)
{
    service.Delete(id);
    return Ok();
}
```


 Elimina un insegnante in base all'ID.


 Potrebbero esserci vincoli nel DB (es. relazioni in Insegnamento )

## Query 3 – Materie e Classi in cui insegna un insegnante

```
[HttpGet("materie-e-classi/{id}")]
public ActionResult GetMaterieEClassi(int id)
{
```

```
return Ok(service.GetMaterieEClassi(id));  
}
```

 Esempio chiamata: GET /Insegnante/materie-e-classi/1

 Restituisce tutte le **materie** e **classi** collegate a quell'insegnante tramite la tabella Insegnamento


 **Risultato atteso**

```
[  
  { "Materia": "Italiano", "NumeroClasse": 1, "Sezione": "A" },  
  { "Materia": "Storia", "NumeroClasse": 1, "Sezione": "A" }  
]
```

---

## Query 5 – Insegnanti che insegnano in più classi

```
[HttpGet("piu-classi")]  
public ActionResult GetInsegnantiInPiuClassi()  
{  
    return Ok(service.GetInsegnantiInPiuClassi());  
}
```

 Esegue una query aggregata che conta in quante classi insegna ciascun insegnante.

 Restituisce solo chi insegna in **più di una classe**.

 **Esempio output**


```
[  
  {  
    "cod_fiscale": "BRRDNL06E24I608Y",  
    "nome": "Daniele",  
    "cognome": "Berardinelli",  
    "NumClassi": 2  
  }  
]
```





# Program.cs – Configurazione dell'Applicazione ASP.NET Core

 **File:** Program.cs

 **Responsabilità:** avvio dell'applicazione, configurazione dei servizi (tra cui SQL, Swagger), middleware, mapping dei controller.

---



## Codice commentato

```
var builder = WebApplication.CreateBuilder(args);

// aggiunge il supporto per i controller (API REST)
builder.Services.AddControllers();

// swagger per la documentazione interattiva
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// configura la connessione al database SQL Server
builder.Services.AddTransient<SqlConnection>(_ =>
    new SqlConnection(builder.Configuration.GetConnectionString("Db")));
```

**i** `AddTransient<SqlConnection>` permette l'iniezione diretta della connessione nel costruttore dei controller.

La stringa "Db" si trova nel file `appsettings.json`

---

```
var app = builder.Build();

// abilita Swagger SOLO in modalità di sviluppo
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();

    // aggiunta foglio di stile custom per Swagger
    app.UseSwaggerUI(c =>
    {
        c.InjectStylesheet("/custom.css"); // file CSS per UI Swagger
        personalizzata
    })
}
```

```
});  
}
```

✅ Swagger viene caricato solo in ambiente di sviluppo ( Development ) per evitare esposizione in produzione

```
// middleware per routing e autorizzazioni  
app.UseAuthorization();  
  
// mapping dei controller alle route (es. /Classe, /Studente)  
app.MapControllers();  
  
// avvio dell'app  
app.Run();
```

## 📄 Esempio Connection String in appsettings.json

```
{  
  "ConnectionStrings": {  
    "Db":  
    "Server=localhost;Database=ScuolaMedia;Trusted_Connection=True;TrustServerCertificate=True;"  
  }  
}
```

## ✅ Funzionalità configurate in Program.cs

Sezione	Descrizione
AddControllers()	Abilita i controller API
AddSwaggerGen()	Abilita Swagger per testare le API
AddTransient()	Inietta connessione SQL per i servizi
MapControllers()	Mappa tutte le route verso i controller
UseSwaggerUI()	Interfaccia Swagger con stile personalizzato

Sezione	Descrizione
Run()	Avvia l'applicazione

---

## Stile Swagger

Il file `custom.css` migliora graficamente Swagger con:

- Colori personalizzati
- Hover animati
- Pulsanti colorati per i metodi HTTP ( GET , POST , PUT , DELETE )