

Prisma

Cosa è e a cosa serve:

Prisma è un **ORM***, ovvero un sistema di astrazione che permette di gestire DB.

Cap. 1 – Installazione e inizializzazione

Installazione:

- `npm init -y`
- `npm i --save-dev`
 - `prisma`
 - `typescript`
 - `ts-node`
 - `@types/node`
 - `nodemon`

Creazione di un file di configurazione per typescript

➤ **Tsconfig.json**

Con all'interno una configurazione proposta da Prisma

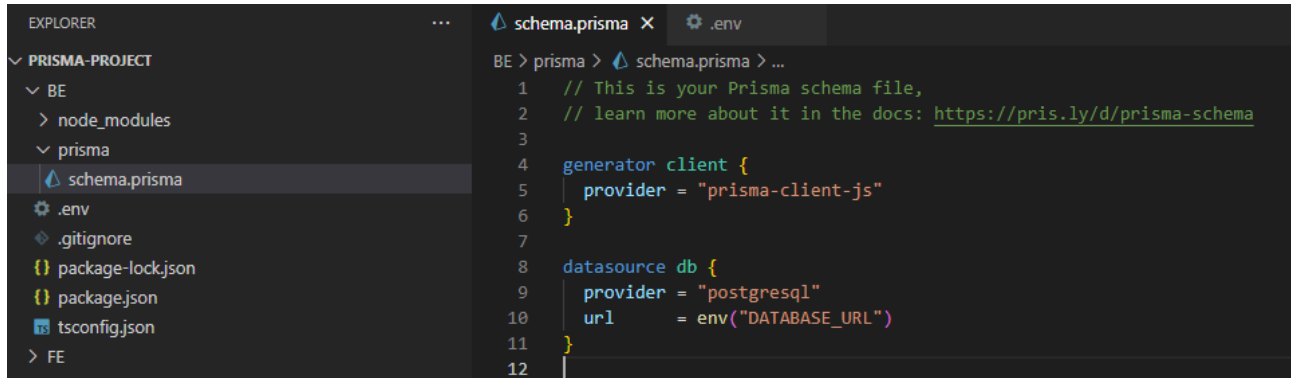
```
1  {
2    "compilerOptions": {
3      "sourceMap": true,
4      "outDir": "dist",
5      "strict": true,
6      "lib": ["esnext"],
7      "esModuleInterop": true
8    }
9  }
```

Inizializzazione ambiente:

- `npx prisma init --datasource-provider postgresql**`

** Cambia in base alla tipologia di DBMS utilizzato

Questo porterà alla creazione dello **schema prisma**:



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'prisma' directory containing 'schema.prisma'. The code editor shows the content of 'schema.prisma' with the following code:

```
1 // This is your Prisma schema file,  
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema  
3  
4 generator client {  
5   provider = "prisma-client-js"  
6 }  
7  
8 datasource db {  
9   provider = "postgresql"  
10  url      = env("DATABASE_URL")  
11 }  
12
```

- **Generator e datasource**

Il **generator** è cosa **genera il tuo codice**, ovvero colui che formatterà il codice (nel caso di uso di **GraphQL API** bisognerà utilizzare un generatore differente, un layer in più, ma nel 99% dei casi d'uso si utilizzerà questo di default).

Il **datasource** è, come dice lo stesso termine, la fonte dalla e verso la quale vengono letti/scritti i dati.

Nel file **.env** verrà definito il DB di riferimento

Es.

```
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/test-prisma"
```

```
postgresql
```

: Tipo di DB

```
postgres
```

: postgres

```
*****
```

: password

```
@localhost:5433
```

: porta di utilizzo

```
/test-prisma
```

: nome DB

* **Object-Relational Mapping** è una tecnica di programmazione che favorisce l'integrazione di sistemi software aderenti al paradigma della programmazione orientata agli oggetti con sistemi RDBMS.

Un prodotto ORM fornisce, mediante un'interfaccia orientata agli oggetti, tutti i servizi inerenti alla persistenza dei dati, astruendo al contempo le caratteristiche implementative dello specifico RDBMS utilizzato.

Cap. 2 – Creazione di un model

È possibile creare un model da sincronizzare con il DB, creando una tabella ex-novo, oppure sincronizzare il model con una tabella già esistente.

Ogni qual volta viene effettuata una modifica verrà effettuata una **migration**, che verrà tracciata in tabella.

Es. di model

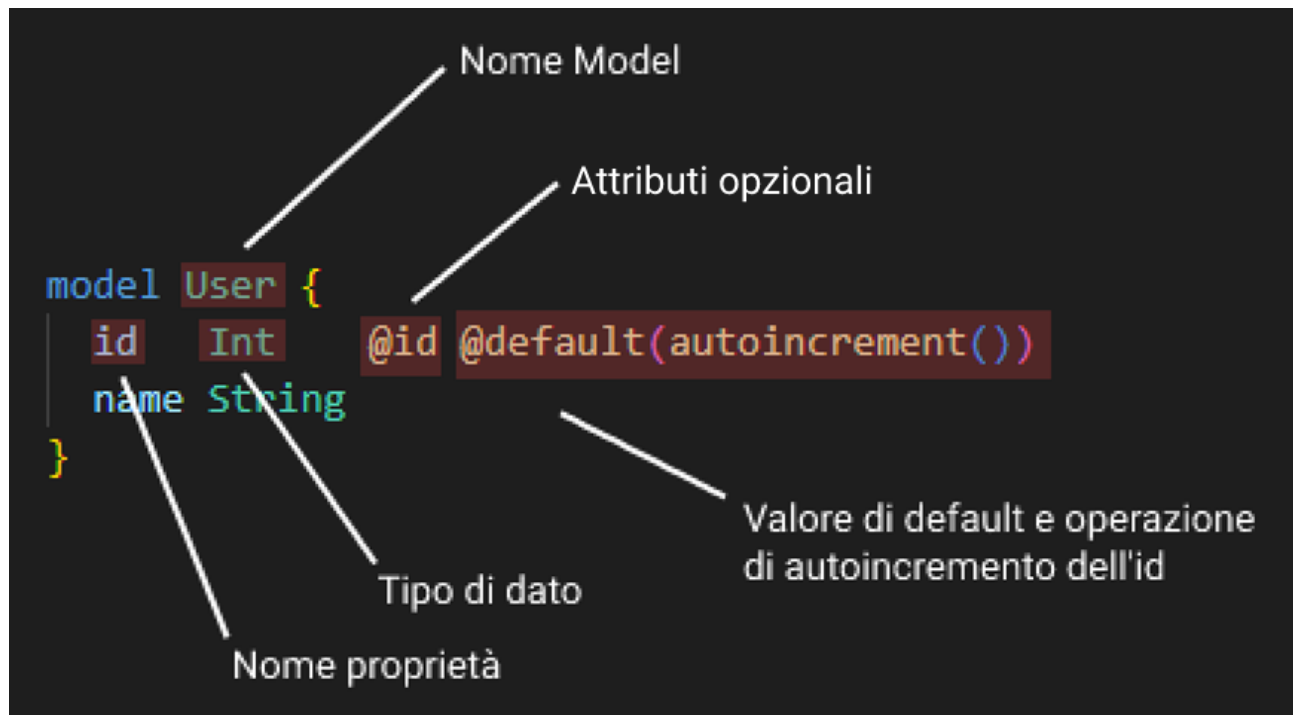


Figura 1 Model User con id e campo nome

Per effettuare la **sincronizzazione** tra model e DB bisognerà effettuare una migration.

```
npx prisma migrate dev --name init
```

...questo creerà il **prisma-client.js**, tutto il codice **generato**.

Esso sarà presente dentro i **node modules**, nella cartella **@prisma**.

Cap. 3 – Prisma Client

Per **gestire** il prisma-client (tutto il codice che. Compilato, permetterà la comunicazione con il DB), bisognerà installarlo adeguatamente, utilizzando:

```
npm i @prisma/client
```

Per poterlo rigenerare manualmente utilizzeremo il comando:

```
npx prisma generate
```

Adesso avremo la possibilità di, creando il file **script.ts**, di gestire manualmente il nostro **Prisma Client**

➤ **script.ts**

📄 script.ts

```
1 import { PrismaClient } from '@prisma/client'
2
3 const prisma = new PrismaClient()
4
5 async function main() {
6   // ... you will write your Prisma Client queries here
7 }
8
9 main()
10 .then(async () => {
11   await prisma.$disconnect()
12 })
13 .catch(async (e) => {
14   console.error(e)
15   await prisma.$disconnect()
16   process.exit(1)
17 })
```

Es. di creazione di un utente con un name e chiamata di visualizzazione di tutti gli altri.

```
5  async function main() {
6    // ... you will write your Prisma Client queries here
7
8    // Crea un utente con un nome
9    const user = await prisma.user.create({
10      data: {
11        name: "Giorgio"
12      }
13    })
14    console.log(user);
15
16    // Richiama tutti gli utenti
17    const users = await prisma.user.findMany()
18    console.log(users);
19  }
```

Flusso logico:

DataSource → generator (codice e transpilazione in un unico file) → a questo punto viene avviata una migration che rende effettive le modifiche al db.

- ➔ Da qui, il nostro generator che una libreria per noi, la script.ts, la quale ci permette di rapportarci con il nostro codice

Cap. 4 – Models

I **models** rappresentano le **tabelle** nel **DB**:

```
13  model User {
14    id      Int      @id @default(autoincrement())
15    name    String
16    isAdmin Boolean
17    preferences Json
18    blob    Unsupported("")
19    Post    Post[]
20  }
21
22  model Post {
23    id      Int      @id @default(autoincrement())
24    rating  Float
25    createdAt DateTime
26    updatedAt DateTime
27    author  User      @relation(fields: [userId], references: [id])
28    userId  Int
29  }
```

Figura 2 Es. di 2 model con una relazione

Ogni model può avere varie tipologie di campi, ognuno dei quali definito con un **tipo di dato** specifico.

- Ci sono casi particolari nei quali un campo può non essere necessario strettamente, in tali casi è giustapponibile il? al disopra del campo per definirne l'opinabilità.

```
isAdmin    Boolean?
```

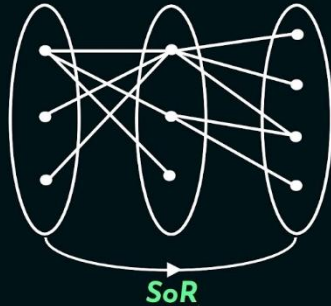
- È possibile definire anche tipologie di dato non supportate, ma tale evenienza si presenta in casi rari.

```
blob       Unsupported("")
```

Cap. 5 – Relazioni tra models

Composition of Relations

Let A , B , and C be three sets. Suppose, R is a relation from A to B , and S is a relation from B to C . The composite of R and S , denoted by SoR , is a binary relation from A to C consisting of ordered pairs (a, c) where $a \in A$ and $c \in C$. Also, $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S$.



Una relazione identifica un rapporto biunivoco tra 2 tabelle a seguito di un legame.

Esse possono essere di **3 Tipologie**:

1. OneToOne
2. OneToMany
3. ManyToMany

Nell'esempio qui presente abbiamo un legame che viene stipulato tra 2 models, ovvero L'utente e i suoi Posts.

Tale legame è definito tramite una @relation.

```
13  model User {
14    id          Int          @id @default(autoincrement())
15    name        String
16    isAdmin     Boolean
17    preferences Json
18    blob        Unsupported("")
19    writtenPosts Post[]      @relation("Written Posts")
20  }
21
22  model Post {
23    id          Int          @id @default(autoincrement())
24    rating       Float
25    createdAt   DateTime
26    updatedAt   DateTime
27    author      User         @relation("Written Posts", fields: [authorId], references: [id])
28    authorId    Int
29  }
30
```

Figura 3 Es. di relazione

"Written Posts" è un label che disambigua nel qual caso siano presenti ulteriori relazioni interne, nel caso in cui, ad esempio, siano presenti post preferiti e post scritti dall'utente stesso.

```

13 model User {
14   id          Int          @id @default(autoincrement())
15   name        String
16   isAdmin     Boolean
17   preferences  Json
18   blob        Unsupported("")
19   writtenPosts Post[]       @relation("Written Posts")
20   favoritesPosts Post[]     @relation("Favorites Posts")
21 }
22
23 model Post {
24   id          Int          @id @default(autoincrement())
25   rating       Float
26   createdAt    DateTime
27   updatedAt    DateTime
28   author       User        @relation("Written Posts", fields: [authorId], references: [id])
29   authorId     Int
30   favoritedBy  User?       @relation("Favorites Posts", fields: [favoritedById], references: [id])
31   favoritedById Int?
32 }

```

Figura 4 Es. di label di disambiguazione

Quest'ultimo è un esempio di relazione **1 – N**

Mentre, per quanto riguarda le **ManyToMany**:

Una categoria può avere tanti posts e ogni post può appartenere a tante categorie

```

    favoritedById Int?
    categories    Category[]
}

```

Figura 5 Post

```

35 model Category {
36   id    Int    @id @default(autoincrement())
37   posts Post[]
38 }

```

Figura 6 Categoria

Esempio di **OneToOne**:

```

20   favoritesPosts Post[]       @relation("Favorites Posts")
21   userPreferences UserPreferences[]
22 }
23
24 model UserPreferences {
25   id    Int    @id @default(autoincrement())
26   user  User    @relation(fields: [userId], references: [id])
27   userId Int
28 }

```

< User

< UserPref.

```

13  model User {
14      id          Int          @id @default(autoincrement())
15      email       String       @unique
16      name        String?
17      posts       Post[]
18      UserPreferences UserPreferences?
19  }
20
21  model UserPreferences {
22      id          Int @id @default(autoincrement())
23      user        User @relation(fields: [userId], references: [id])
24      userId      Int @unique
25  }

```

Gli **attributi**, come ad es. **unique**, **updatedAt**, **default(now())**, ecc... che permettono di definire delle proprietà per i singoli campi.

Possiamo avere 2 **tipologie di attributi**:

- Single attribute @

```

13  model User {
14      id          Int          @id @default(autoincrement())

```

- Block level attribute @@

```

13  model User {
14      id          Int          @id @default(autoincrement())
15      age         Int?
16      email       String       @unique
17      name        String?
18      posts       Post[]
19      UserPreferences UserPreferences?
20
21      @@unique([age, name])
22      @@index([email])
23  }

```

Figura 7 Utilizzo di un constraint unique su 2 campi combinati del model User



What are enums in TypeScript

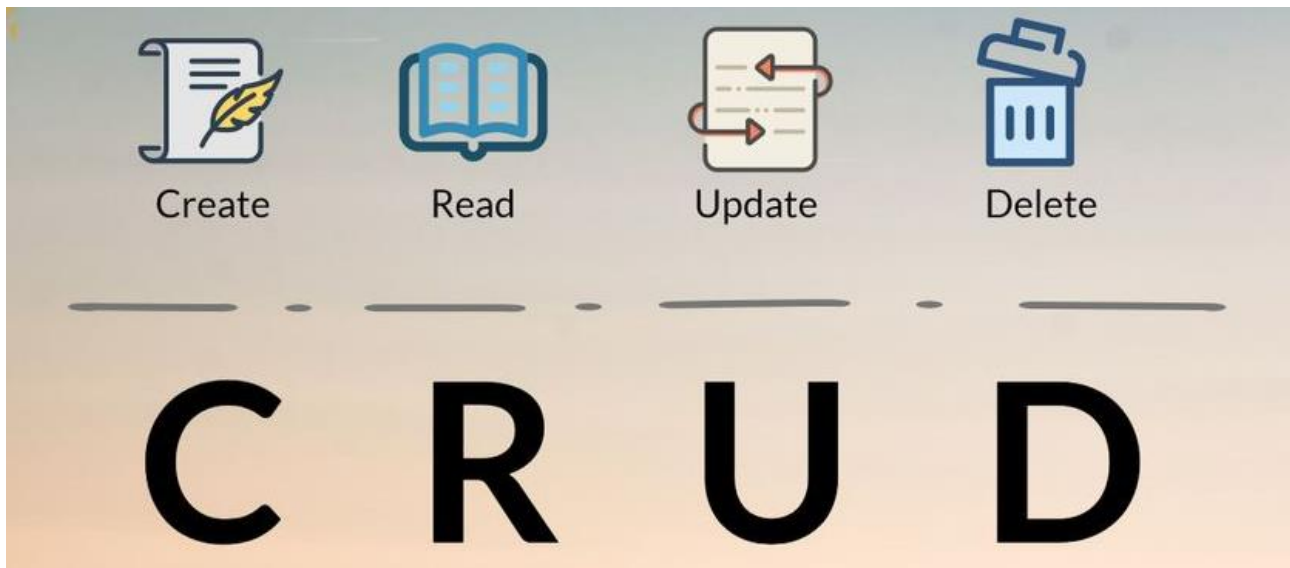
```
enum FavoriteColors {  
  GREEN = 'green',  
  BLUE = 'blue'  
}
```

Enumerazione di valori utilizzabile per la creazione di liste di valori utilizzabili come valori nei campi degli altri Models.

User ----->

Enum ----->

```
25 |  
26 |   role Role @default(BASIC)  
27 | }  
28 |  
29 | enum Role {  
30 |   BASIC  
31 |   ADMIN  
32 | }
```



Create:

- Creazione di un record in una tabella

...

```
56   const user = await prisma.user.create({  
57     data: {  
58       age: 30,  
59       email: "a@prism.net",  
60       name: "Bobo"  
61     }  
62   })
```

...

- Creazione di 2 record (User e UserPreferences) legati da una relazione

```

13 > model User { ...
25   }
26
27   model UserPreferences {
28     id          Int      @id @default(autoincrement())
29     user         User     @relation(fields: [userId], references: [id])
30     userId       Int      @unique
31     updatedEmail Boolean
32   }

```

Figura 8 Models

```

const user2 = await prisma.user.create({
  data: {
    age: 30,
    email: "c@prism.net",
    name: "Bobo2",
    UserPreferences: {
      create: {
        updatedEmail: true
      }
    }
  }
})

```

Metodo di creazione di un **utente** e di un altro record di un altro **model** **relazionato**.

Nota: è possibile mettere in chiaro i log delle **query**, dei **warn** o degli **error** che sono eseguiti in fase di esecuzione di **script.ts**.

Nota 2:

```

include: {
  UserPreferences: true
}

```

permette di richiamare anche i models con il quale il nostro model ha delle relazioni

```
const prisma = new PrismaClient({ log: ["query"] })
```

- Creazioni di record in Bulk

```

const user = await prisma.user.createMany({
  data: [
    {
      age: 30,
      email: "h@prism.net",
      name: "Bobo7",
    },
  ],
})

```

...



Read:

- 1) Filtri base
- 2) Filtri avanzati
- 3) Operatori logici
- 4) Filtri relazionali

Filtro base

- Lettura di un record in base ad un **parametro unico**

```
const user = await prisma.user.findUnique({  
  where: {  
    id: 1  
  }  
})
```

// Filtro semplice

```
  where: {  
    age_name: {  
      age: 32,  
      name: "Bobo3"  
    }  
  }
```

// Filtro per block attribute

- Lettura del primo record con quelle specifiche

```
const user = await prisma.user.findFirst({  
  where: {  
    age: 32  
  }  
})
```

- Lettura di tutti i record per quella tabella

```
const users = await prisma.user.findMany({
```

Filtro avanzato

- Distingue i risultati per l'attributo nome

```
distinct: ["name"],
```

- Prende i primi 2 risultati

```
take: 2,
```

- Salta il primo risultato

```
skip: 1
```

- Ordinamento

```
orderBy: {  
  age: "asc"  
}
```



- Operatori

```
where: {  
  name: {  
    equals: "Sally"  
  }  
},
```

// Uguale

```
where: {  
  name: {  
    not: "Sally"  
  }  
},
```

// Non uguale

```
where: {  
  name: {  
    in: ["Sally", "Alice"]  
  }  
},
```

// Uguale a un array

```
age: {  
  lt: 30  
}
```

```
//<
```

```
    age: {  
      gt: 30  
    }  
  }  
}
```

```
//>
```

```
  where: {  
    email: {  
      contains: "@prisma"  
    }  
  }  
}
```

// campo contiene una stringa

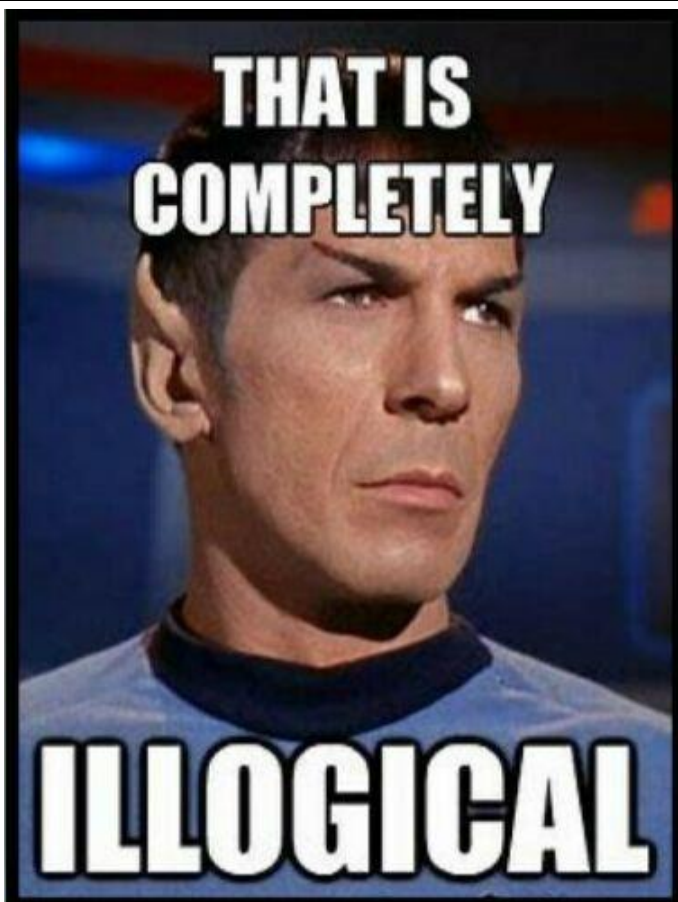
```
  email: {  
    endsWith: "@prisma"  
  }  
}
```

// Finisce con

```
  email: {  
    startsWith: "@prisma.io"  
  }  
}
```

// Inizia con

Operatori logici



▪ AND

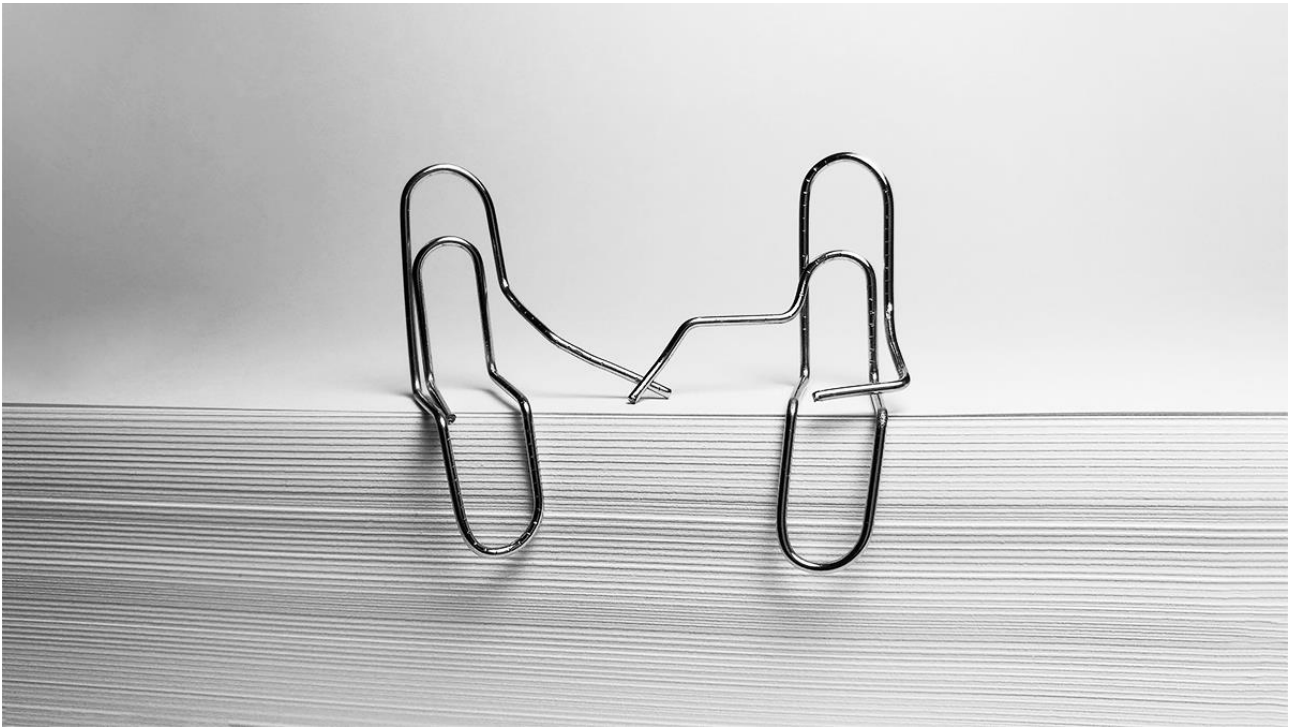
```
AND: [  
  { email: { endsWith: "@prisma.io" } },  
  { name: { contains: "Alice" } }  
]
```

▪ OR

```
OR: [  
  { email: { endsWith: "@prisma.io" } },  
  { name: { contains: "Alice" } }  
]
```

▪ NOT

```
NOT: [  
  { email: { endsWith: "@prisma.io" } },  
  { name: { contains: "Alice" } }  
]
```



Esempio base:

```
where: {
  UserPreferences: {
    updatedEmail: false
  }
}
```

Filter on "-to-many" relations

Prisma Client provides the `some`, `every`, and `none` options to filter records by the properties of related records on the "-to-many" side of the relation. For example, filtering users based on properties of their posts.

Figure 1 Parametri utilizzabili in relazioni "-to-many"

Every, some e none:

```
const users = await prisma.user.findMany({
  where: {
    posts: {
      some: {
        title: {
          equals: "Test per update"
        }
      }
    }
  }
})
```

// Filtro che permette di filtrare tutti gli utenti con post aventi titoli contenuti la stringa "Test per update"

Filter on "-to-one" relations

Prisma Client provides the `is` and `isNot` options to filter records by the properties of related records on the "-to-one" side of the relation. For example, filtering posts based on properties of their author.

For example, the following query returns `Post` records that meet the following criteria:

- Author's name is not Bob
- Author is older than 40

Figure 2 Parametri utilizzabili in relazioni "-to-one"

Is e IsNot:

```
const users = await prisma.user.findMany({
  where: {
    UserPreferences: {
      is: {
        updatedEmail: true
      }
    }
  }
})
```

// Filtro che prende il valore se il campo è vero



Update:

L'update utilizza la combinazione di **where** e **data**

```
const users = await prisma.user.update({
  where: {
    id: 15
  },
  data: {
    email: "zuppa@prisma.it"
  }
})
```

Tra gli **operatori utili** da utilizzare abbiamo:

- **increment**
- **decrement**
- **multiply**
- **divide**
- ecc...

Es.

...

```
data: {
  age: {
    increment: 1
  }
}
```

...

Connect e disconnect

```
const users = await prisma.user.update({
  where: {
    id: 21
  },
  data: {
    UserPreferences: {
      disconnect: true
    }
  }
})
```

// Permettono di scindere o suggellare un legame



Delete:

Eliminazione di un singolo elemento

```
const users = await prisma.post.delete({  
  where: {  
    id: 10  
  }  
})
```

Eliminazione di più elementi

```
const users = await prisma.post.deleteMany({  
  where: {  
    id: 10  
  }  
})
```

Flow di lavoro con Prisma in sintesi:

- 1) In `schema.prisma`
 - | Effettua modifiche
 - // es. Crea un Model
- 2) In `terminale`
 - | Sincronizza i models nel DB
 - // Avvia il comando `npx prisma migrate dev --name init`
- 3) (Opzionale in caso di mancato aggiornamento di Prisma Client)
 - Chiudi/Apri VSCode
 - Apri il `prisma-client.js` originale
 - Cancella file `script.ts` e ricrealo
- 4) In `terminale`
 - | Rigenera il Prisma Client così da rigenerare il cod. di modifica
 - // Avvia il comando `npx prisma generate`

