Data Management using

Prisma

Course Roadmap

Web Client

Request

Response

Web Server

Frontend development

- HTML for page content & structure
- CSS for styling
- JavaScript for interaction

Backend development

- Web API
- Web Pages
- Data Management

We are HERE

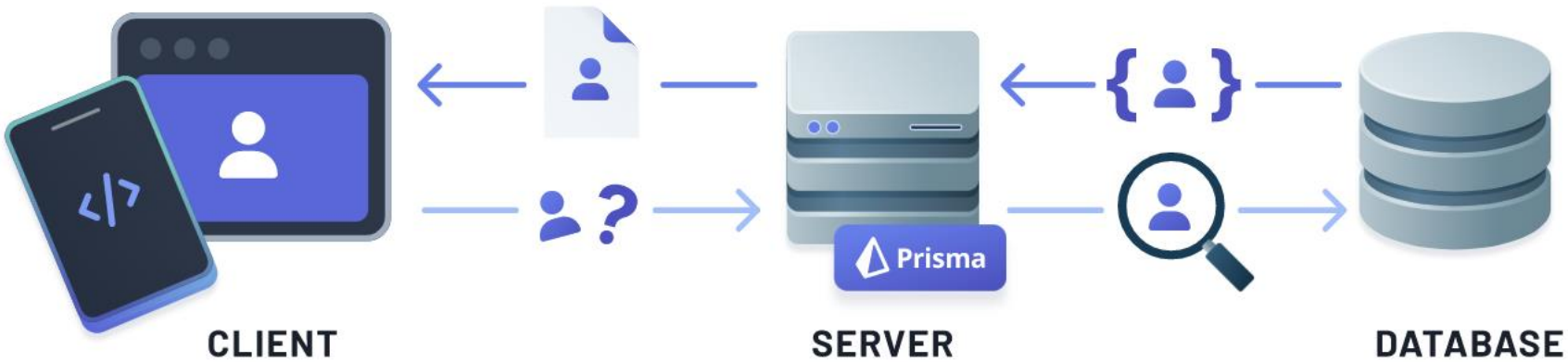
NEXT.JS

Prisma

Outline

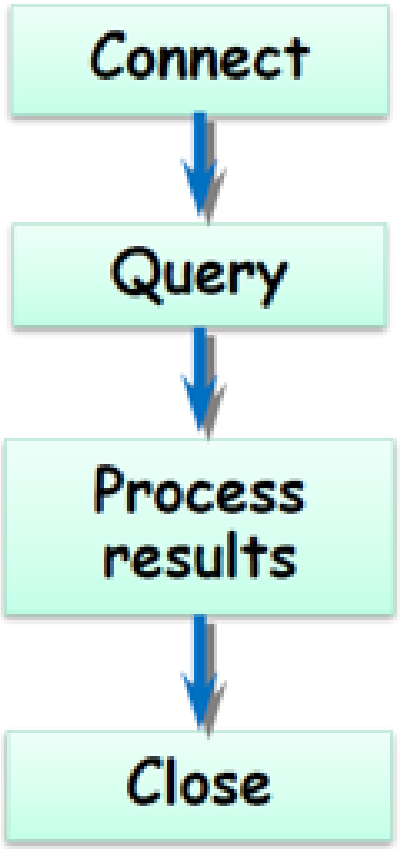
1. What is Prisma?
2. Data Model (Prisma Schema)
3. Migration (Apply changes to DB)
4. Queries (using Prisma Client)
5. Aggregation Queries

What is Prisma ?



Using Database without Prisma

- To use a database (DB) without Prisma you need to:
  - Connect to the DB server
  - Submit a SQL query statement
  - Get the results (in tabular format)
  - Convert the results to objects
  - Close the connection



Using Database without Prisma

- Structured Query Language (SQL)
  - Language used to define, alter and access the elements described above
- Creating data:

```
INSERT into PERSON (first_name, last_name)
  VALUES ('Ahmed', 'Sayed')
```

- Reading data:

```
SELECT first_name FROM person WHERE last_name = 'Sayed'
```

- Updating data:

```
UPDATE person SET first_name = 'Ali' where
  last_name = 'Sayed'
```

- Deleting data:

```
DELETE from person where last_name = 'Sayed'
```

What is Prisma?

- Prisma is a **server-side library** that simplifies read and write data to the database in an intuitive and efficient way
- Open-source **Object-Relational-Mapper (ORM)**, includes:
  - **Prisma Schema**: used to define the **data model** (entities and relations)
  - **Prisma Migrate**: apply schema changes to DB
  - **Prisma Client**: auto-generated to query data
  - **Prisma Studio**: GUI to view and edit data in your DB
- Why Prisma?
  - Facilitates defining the data model
  - Helps reducing the amount of code to read/write to a DB
  - Less or no SQL code to read/write to a DB
  - Abstract database-specific details => makes easier to change from one database to another

schema.prisma

- **Data Model** is defined in 1 file (**schema.prisma**)
  - Specifies the app entities and their relations
  - Syntax used is Prisma Schema Language (PSL)
- schema.prisma also specifies:
  - **Data source**: defines the data source details:
    - Database Provider (e.g., a PostgreSQL or SQLite)
    - Connection Url (e.g., postgresql://janedoe:mypassword@localhost:5432/mydb)
  - **Generator**: specifies what client should be generated based on the data model (e.g., Prisma Client)

Prisma DB providers

Reminder – Next.js getting started

- Create an empty folder (with no space in the name use **dash -** instead)
- Create next.js app (accept default for all questions)

```
npx create-next-app@latest .
```

```
√ Would you like to use TypeScript with this project? ... No / Yes
√ Would you like to use ESLint with this project? ... No / Yes
√ Would you like to use Tailwind CSS with this project? ... No / Yes
√ Would you like to use `src/` directory with this project? ... No / Yes
√ Use App Router (recommended)? ... No / Yes
√ Would you like to customize the default import alias? ... No / Yes
```

This creates a new **`Next.js`** project and downloads all the required packages

- Run the app in dev mode: **`npm run dev`**

Prisma – Getting started

- Install the Prisma packages using:

```
npm install prisma --save-dev
npm install @prisma/client
```

- Also install **Prisma VS Code extension**
- Set up Prisma with this command:

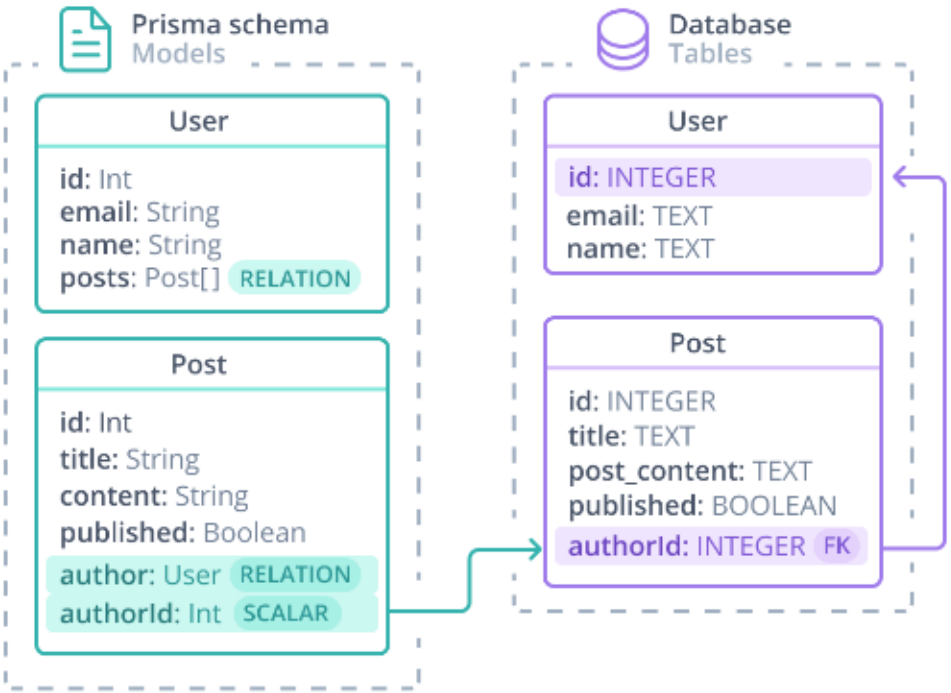
```
npx prisma init --datasource-provider sqlite
```

  - This creates a new **prisma** directory with **schema.prisma** file and configures SQLite as your database
  - You can define the data model inside **schema.prisma** file

Data Model

Data Model

- Data Model (aka. Schema) have two main purposes:
  - Describe app entities that map to tables in the underlying database: Data Model is used to create the database tables using **Prisma Migrate**
  - Serve as foundation to generate **Prisma Client** API
- A data model describes your app entities. For example:
  - In an ecommerce app you have models like Customer, Order, Item and Invoice
  - In a social media app you have models like User, Post, Photo and Message



Defining fields

- Each model entity defines fields
- Each field in the model has a type, e.g., **`id Int`**
  - A field type could be **scalar** type such as **`Int`**, **`String`**, **`Boolean`**, **`Float`**, **`DateTime`** or could be a **Relation** field to another Model
  - Optional type modifiers: **`[]`** makes a field a list

  `?` makes a field optional
- Fields may use **field attributes** to define:
  - Primary keys with the **`@id`** attribute.
    - Each model has a unique id that uniquely identifies each entity instance in the DB
  - Unique keys with the **@unique** attribute
  - Default values with the **@default** attribute

Identifier Generation

- Identifiers can be generated in the database by specifying `@id @default(...)` on the identifier
- The most common generation strategies include:
  - `@default(autoincrement())`: Id gets auto incremented by 1 by the DB
  - `id String @id @default(cuid())`: generates a globally unique identifier

Data Model Example

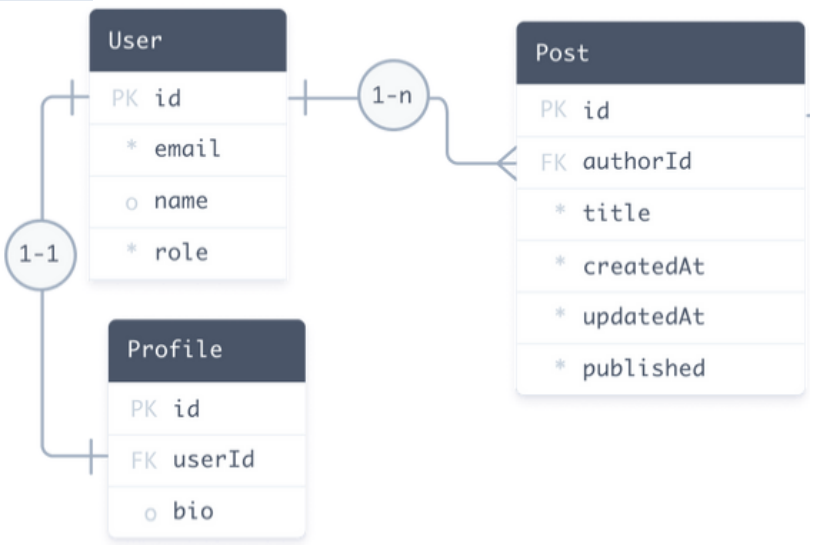
```
model User {
  id      Int      @id @default(autoincrement())
  email   String   @unique
  name    String?
  role    Role     @default(USER)
  posts   Post[]
  profile Profile?
}

model Profile {
  id     Int    @id @default(autoincrement())
  bio    String
  user   User   @relation(fields: [userId], references: [id])
  userId Int    @unique
}

model Post {
  id          Int        @id @default(autoincrement())
  createdAt   DateTime   @default(now())
  updatedAt   DateTime   @updatedAt
  title       String
  published   Boolean    @default(false)
  author      User       @relation(fields: [authorId], references: [id])
  authorId    Int
}

enum Role {
  USER
  ADMIN
}
```

DB Schema



Modeling relations

- A relation is a connection between two models. For example, there is a **one-to-many** relation between User and Post
  - At a Prisma level, a connection between two models is **always** represented by a **relation field** on **each side** of the relation.
- User / Post relation is made up of:
  - Two **relation** fields: **author** and **posts**. Relation fields define connections between models at the Prisma level and **do not exist in the database**.
    - These fields are used to generate the Prisma Client
  - The scalar **authorId** , which is referenced by the `@relation` attribute is the **foreign key** that connects **Post** and **User a**s defined by the attribute
    - This field **does exist in the database**

Defining One-to-Many Relationship

- The **Many side** (i.e., the entity having the foreign key) defines the mapping to the database using **`@relation`** to specify the foreign key column
- The **One** side of the relation must refer to the Many side by having a relation field
  - (e.g., **User** has **posts Post[]** relation field)

@@unique & @@id

Composite primary key

```
model User {
  firstName String
  lastName  String
  email     String  @unique
  isAdmin   Boolean @default(false)

  @@id([firstName, lastName])
}
```

Composite Unique key

```
model User {
  id        Int     @id @default(autoincrement())
  firstName String
  lastName  String
  email     String  @unique
  isAdmin   Boolean @default(false)

  @@unique([firstName, lastName])
}
```

@map

- By default, model field names are the same as the DB table column names
- **@map** attribute can be used for mapping between model fields and table columns
  - e.g., the **content** field maps to the **post_content** database column

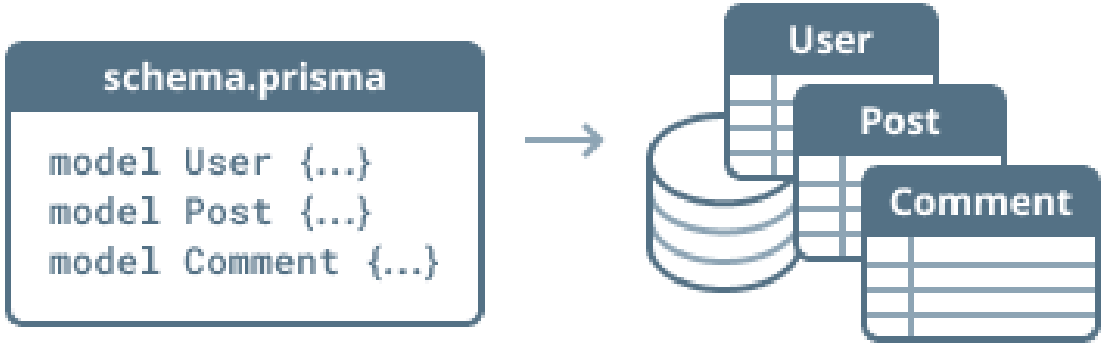
```
model Post {
  id        Int     @id @default(autoincrement())
  title     String
  content   String? @map("post_content")
  published Boolean @default(false)
  author    User?   @relation(fields: [authorId], references: [id])
  authorId  Int?
}
```

@@map

- By default, model name is the same as the DB table name
- **@@map** can be used to map the model's name to a different table name
  - E.g., Comment model can be mapped to the comments table in the underlying database

```
model Comment {
   // Fields
   @@map("comments")
}
```

Migration
(apply changes to DB)



Migration

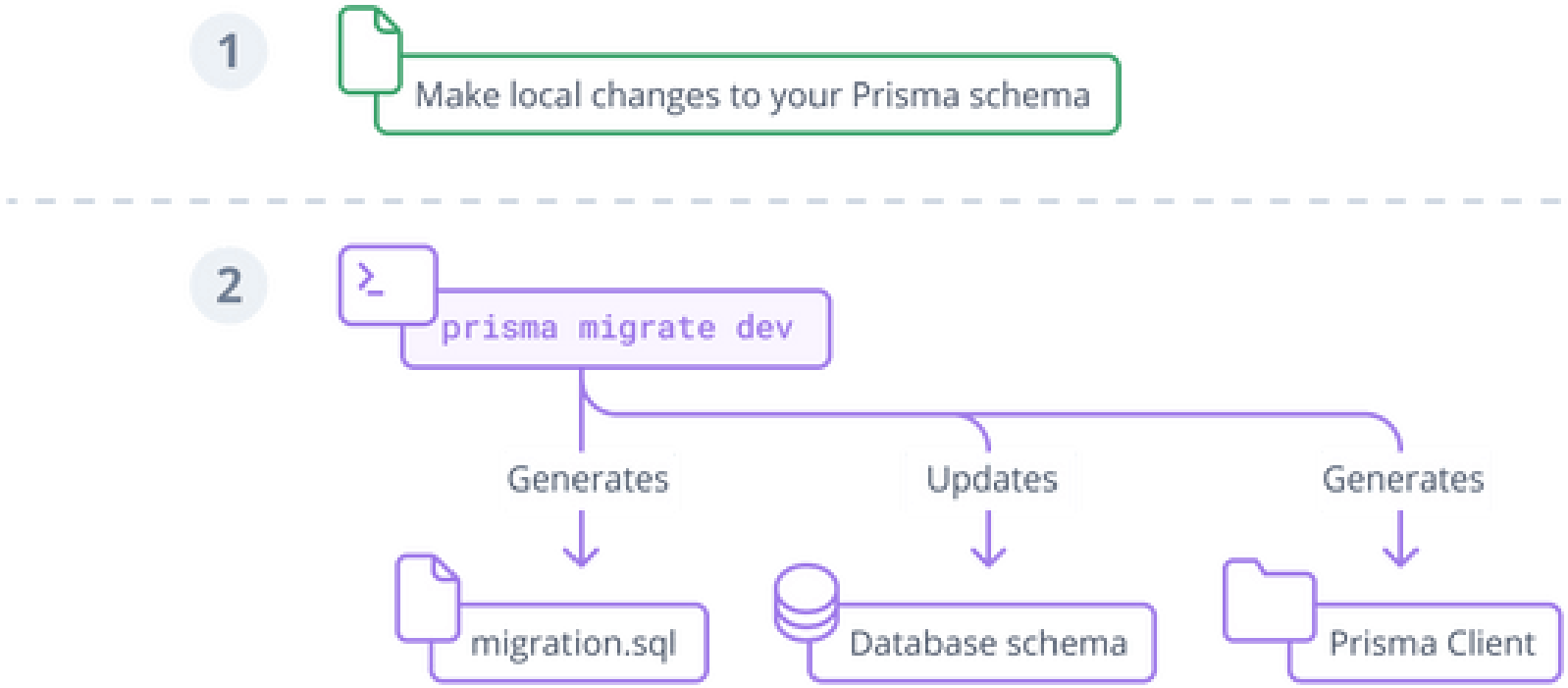
- Prisma Migrate auto-generates SQL migration file from the Prisma schema to apply the changes to the database:
  - Keep the database schema in sync with Prisma schema (while keeping existing data in your database)

Prisma migrate

`npx prisma migrate dev --name init`

- This command did 4 things:
  - It creates a new SQL migration file under `prisma/migrations` directory
  - It runs the SQL migration file against the local development database
  - It generates Prisma Client
  - It runs database initialization code in **seed.js** (if any)
- If the database does not exist, then I will create it
  - E.g., if the SQLite database file didn't exist, the command also created it inside the prisma directory with the name **`dev.db`** as defined via the environment variable in the .env file

Prisma migrate workflow



Note that **`prisma db push`** command allows syncing the Prisma schema and database schema without persisting a migration under `/prisma/migrations`

Queries
(using Prisma Client)

SQL\Prisma	Single	Multiple
Insert	create	createMany
Update	update	updateMany
Delete	delete	deleteMany
Select	findUnique/ findFirst	findMany
Insert/Update	upsert	-

Prisma Client

- Run `npx prisma migrate`

(or `npx prisma generate`)

To generate a Prisma Client that is tailored to data models defined in **schema.prisma**

It offers auto-completion to help write the queries to read/write to DB

```
import { PrismaClient } from '@prisma/client'

const prisma = new PrismaClient()

const newAuthor = await prisma.author.create({
data: {
    firstName: 'John',
    lastName: 'Doe',
},
})

const authors = await prisma.author.findMany()
```

Role of Prisma Client

- Prisma client lets you access the database without writing SQL. Instead, you write code in JavaScript to read/write to the database, and the Prisma Client will **translate** it into SQL queries

```
prisma.post.findMany({
    where: {
        published: true
    },
    orderBy: {
        createdAt: "desc"
    }
});
```

→

```
SELECT * FROM post
WHERE published = true
ORDER BY updatedAt DESC
```

DB Operations

Prisma client offers the following operations for each model:

- `create/createMany`
- `update/updateMany`
- `upsert (create or update)`
- `delete/deleteMany`
- `findUnique/findMany/findFirst`
- `aggregate/groupBy`

Example Query

Query

```
// Creating a new record
await prisma.user.create({
  firstName: "Alice",
  email: "alice@prisma.io"
})
```

Table

id	firstName	email
1	Bobby	bobby@tables.io
2	Nilufar	nilu@email.com
3	Jürgen	jums@dums.edu
4	Alice	alice@prisma.io

```
const user = await prisma.user.findUnique({
  where: {
    email: 'alice@prisma.io',
  },
})
```

- All queries return plain old JavaScript objects

Get record by ID or unique identifier

- Queries return a single record using **`findUnique`** by **`Id`** or by a column marked as **unique**

```
// By unique identifier
const user = await prisma.user.findUnique({
  where: {
    email: 'elsa@prisma.io',
  },
})

// By ID
const user = await prisma.user.findUnique({
  where: {
    id: 99,
  },
})
```

Fetching relations

- By default, Prisma will return all the scalar fields of a model
- Fetch relations with Prisma Client is done with the **include** option. For example, to fetch a user and their posts would be done as follows:

```
const user = await prisma.user.findUnique({
  where: {
    email: 'alice@prisma.io',
  },
  include: {
    posts: true,
  },
})
```

Get a filtered list of records

- The following query returns users with an email that ends with *prisma.io* and have at least one (some) published post

```
const users = await prisma.user.findMany({
  where: {
    email: {
      endsWith: "prisma.io"
    },
    posts: {
      some: {
        published: true
      }
    }
  },
}
```

Select a subset of fields

- The following query uses **select** to return the email and name fields

```
const user = await prisma.user.findUnique({
  where: {
    email: 'emma@prisma.io',
  },
  select: {
    email: true,
    name: true,
  },
})
```

```
{ email: 'emma@prisma.io', name: "Emma" }
```

Select a subset of related record fields

- The following query uses a nested select to return: the user's email & the likes field of each post

```
const user = await prisma.user.findUnique({
  where: {
    email: 'emma@prisma.io',
  },
  select: {
    email: true,
    posts: {
      select: {
        likes: true,
      },
    },
  },
})
```

```
{ email: 'emma@prisma.io', posts: [ { likes: 0 }, { likes: 0 } ] }
```

Update a single record

- The following query uses **update** to find and update a single User record by email:

```
const updatedUser = await prisma.user.update({
  where: {
    email: 'viola@prisma.io',
  },
  data: {
    name: 'Viola the Magnificent',
  },
})
```

Update multiple records

- The following query uses **`updateMany`** to update all User records that contain prisma.io

```
const updatedCount = await prisma.user.updateMany({
  where: {
    email: {
      contains: 'prisma.io',
    },
  },
  data: {
    role: 'ADMIN',
  },
})
```

upsert

- Query uses **`upsert`** to update a user record with a specific email address, or create that user record if it does not exist

```
const upsertUser = await prisma.user.upsert({
  where: {
    email: 'fatima@prisma.io',
  },
  update: {
    name: 'Fatima the Magnificent',
  },
  create: {
    email: 'fatima@prisma.io',
    name: 'Fatima the Magnificent',
  },
})
```

number operations

- Use number operations to update a number field based on its current value using **increment**, **decrement**, **multiply** and **divide**
- The following query increments the views and likes fields by 1

```
const updatePosts = await prisma.post.updateMany({
  data: {
    views: {
      increment: 1,
    },
    likes: {
      increment: 1,
    },
  },
})
```

delete and deleteMany

- Query uses delete to **`delete`** a single user

```
const deleteUser = await prisma.user.delete({
  where: {
    email: 'bert@prisma.io',
  },
})
```

- Query uses **`deleteMany`** to delete all users where email contains *prisma.io*

```
const deleteUsers = await prisma.user.deleteMany({
  where: {
    email: {
      contains: 'prisma.io',
    },
  },
})
```

onDelete: Cascade

- Attempting to delete a user with one or more posts result in an error, as every post requires an author
- Adding `onDelete: Cascade` to the author field on the Post model means that deleting the User record will also delete all related Post records

```
model Post {
  id        Int      @id @default(autoincrement())
  title     String
   …
  authorId  Int // Foreign key (FK) associted to PK of User
  author    User     @relation(fields: [authorId],
                 references: [id], onDelete: Cascade)
}
```

prisma.$queryRaw

- Prisma Client has methods to send raw SQL queries

  $queryRaw returns query results of a SELECT statement

  $executeRaw returns a count of affected rows after an UPDATE or DELETE

- Examples

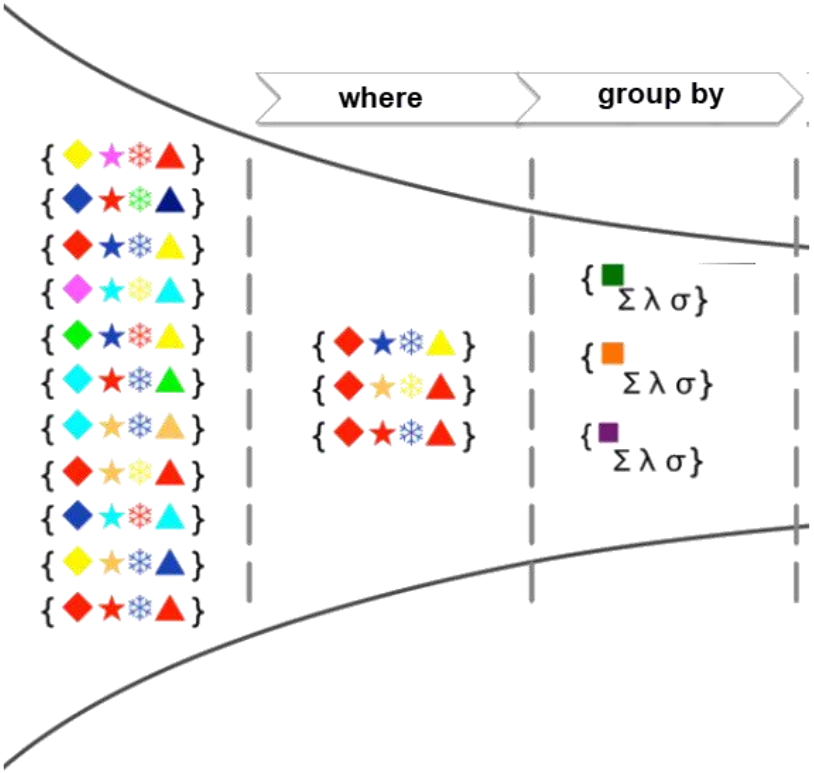
```
const users = await prisma.$queryRaw`SELECT * FROM User`
```

=> Returns a list of users

```
const email = "alice@prisma.io"
const result =
  await prisma.$queryRaw`SELECT * FROM User WHERE email = ${email}`
```

=> Returns the user having the email *alice@prisma.io*

Aggregation Queries



Aggregation Queries

- Summarize data typically for reports
- Prisma Client allows to **aggregate** (avg, count, sum) on the number fields (Int and Float) of a model
  - E.g., query returns the average age and count of users

```
const aggregations = await prisma.user.aggregate({
  _avg: {
    age: true,
  },
  _count: {
    age: true,
  },
})
```

```
{
  _avg: {
    age: 32
  },
  _count: {
    id: 9
  }
}
```

Aggregate with Filtering and Ordering

You can aggregate after filtering and ordering e.g., return the average age of 10 youngest *prisma.io* users:

- Where email contains *prisma.io*
- Ordered by age ascending
- Limited to the 10 users

```
const aggregations = await prisma.user.aggregate({
  _avg: {
    age: true,
  },
  where: {
    email: {
      contains: 'prisma.io',
    },
  },
  orderBy: {
    age: 'asc',
  },
  take: 10,
})

console.log('Average age:' + aggregations._avg.age)
```

Group By

- This groups all users by the **country** field and returns the total number of profile views for each country

```
const groupUsers = await prisma.user.groupBy({
  by: ['country'],
  _sum: {
    profileViews: true,
  },
})
```

```
[
  { country: 'Germany', _sum: { profileViews: 126 } },
  { country: 'Sweden', _sum: { profileViews: 0 } },
]
```

Filtering before and after grouping

- Use **where** to filter all records before grouping. E.g., only includes users where the email address contains prisma.io
- Use **having** to filter groups by an aggregate value such as the sum or average of a field

e.g., only return groups where the average profileViews is greater than 100

```
const groupUsers = await prisma.user.groupBy({
  by: ['country'],
  where: {
    email: {
      contains: 'prisma.io',
    },
  },
  _sum: {
    profileViews: true,
  },
  having: {
    profileViews: {
      _avg: {
        gt: 100,
      },
    },
  },
})
```

Prisma Studio

- GUI to view, explore and edit the data in the DB
  - Browse across tables, filter, paginate, traverse relations and edit data

```
npx prisma studio
```

DB Seeding

- Allows initialing the database with
  - data that is required for the app to start (e.g., adding user types)
  - basic data for testing and using the app in a development environment
- Add DB initialization code to **`seed.js`** file
- Add this to package.json:

```
"prisma": {
    "seed": "node prisma/seed.js"
}
```

- Run it using: **`npx prisma db seed`**

Resources

- Prisma Documentation

https://www.prisma.io/docs/getting-started/quickstart

- Prisma Playground

https://playground.prisma.io/examples/

- Prisma Examples

https://github.com/prisma/prisma-examples

- Aggregation Queries

https://www.prisma.io/docs/concepts/components/prisma-client/aggregation-grouping-summarizing