Prisma

Índice

Introducción a Prisma

Prisma es una herramienta de ORM (Object-Relational Mapping) que ayuda a los desarrolladores a trabajar con bases de datos de una manera más sencilla y eficiente. Con Prisma, los desarrolladores pueden definir modelos de datos y trabajar con ellos como si fueran objetos, sin tener que escribir consultas complejas en SQL. Además, Prisma facilita la implementación de características avanzadas como relaciones, transacciones y migraciones.

Instalación

Para instalar Prisma, es necesario tener Node.js y npm instalados en el sistema. Luego, se puede instalar Prisma a través de npm con el siguiente comando:

```
npm install prisma --save
```

Creación de proyecto

Para crear un nuevo proyecto con Prisma, se puede utilizar el siguiente comando:

```
npx prisma init
```

Esto creará un nuevo directorio llamado `prisma` en el proyecto, que contendrá los archivos de configuración de Prisma y un archivo `schema.prisma` donde vamos a definir nuestro modelo, este es un archivo `schema.prisma` convencional:

```
datasource db {
    provider =  "postgresql"
    url =  "postgresql://user:password@localhost:5432/mydb?schema=public"
}

generator client {
    provider =  "prisma-client-js"
}

model User  {
    id Int @id @default(autoincrement())
    name String
    email String @unique
    password String
}
```

Si no entendés nada de lo que acabas de leer, no te preocupes, es solo para que tengas una idea.

Modelo de datos

Para crear un modelo de datos con Prisma, se utiliza su lenguaje de definición de esquemas (SDL). Dentro del modelo identificamos tres elementos principales, el `datasource`, `generator` y `model`.

Datasource

El `datasource` es una de las cosas más hermosas de esta herramienta, porque es la parte que se encarga de conectarse con nuestra base de datos local con dos simples parámetros, un `provider` que define la base de datos que usaremos, como PostgreSQL, MySQL, etc. Y una `url` que es la dirección para conectarse a esta base de datos en cuestión. Este es un ejemplo de un `datasource` general:

```
datasource db {
    provider =  "postgresql"
    url =  "postgresql://user:password@localhost:5432/mydb?schema=public"
}
```

NOTA: Para evitar problemas de seguridad, se recomienda fuertemente usar un archivo de variables de entorno (También llamado ENV) para guardar la `url` de tu base de datos.

Generator

El `generator` es una sección que permite definir cómo Prisma debe generar el código necesario para interactuar con la base de datos según el modelo de datos definido en el esquema. Un ejemplo de `generator` seria el siguiente:

```
generator client {
    provider =  "prisma-client-js"
}
```

Model

Vamos a comprender al `model` como el equivalente a las tablas y columnas de una base de datos convencional, acá tenemos un ejemplo extraído del punto anterior.

```
model User {
    id Int @id @default(autoincrement())
    name String
    email String @unique
    password String
}
```

En este modelo, se definen los campos `id`, `name`, `email` y `password`. Esto, nuevamente, es el equivalente de una base de datos a una tabla llamada `User`, con una clave primaria `id` y otras tres columnas o campos que definen a nuestro usuario.

Recomendamos ver la documentación original de Prisma para saber más.

Relaciones

Una de las características más importantes de Prisma es su capacidad para trabajar con relaciones entre diferentes modelos de datos. A continuación, se detallan los tipos de relaciones que pueden ser definidas en el archivo `schema.prisma`, y algunas de las propiedades que pueden ser utilizadas para personalizar su comportamiento.

Uno a Muchos

Una relación de uno a muchos se define cuando un modelo tiene una relación con uno o más modelos secundarios. Por ejemplo, un usuario en una aplicación de una lista de tareas, puede tener muchas anotadas, por lo que para un solo usuario, hay muchos registros de tareas pertenecientes a ese usuario. Esto en Prisma se adapta de la siguiente forma:

```
model User {
    id Int @id @default(autoincrement())
    name String
    email String @unique
    password String
    todos Todo[]
}

model Todo {
    id Int @id @default(autoincrement())
    name String
    description String
    isFinished Bool
    authorId Int
    author User @relation(fields: [authorId], references: [id])
}
```

Como ven, se crea un modelo nuevo, en este caso llamado `Todo` para almacenar todos los registros de `todos` con sus campos correspondientes, en términos de bases de datos, seria como crear una nueva tabla y agregar algunas columnas. Por otro lado, esto no crea en sí la relación, sino que lo hacen las nuevas líneas que vemos en el ejemplo. El `todo Todos[]` declara que existe la posibilidad de que en registros del modelo `Todo` se pueda hacer referencia a un registro de un usuario. A sí mismo, del otro lado se hace algo similar, ya que la propiedad `author` y `authorId` hacen que cada registro, o cada `todo` necesite tener el `ID` del autor, esto es gracias al `authorId` que es usado como clave foránea al `id` del modelo `User`.

Muchos a Muchos

Una relación de muchos a muchos se define cuando dos o más modelos tienen una relación con múltiples instancias de otros modelos. Por ejemplo, muchos usuarios pueden pertenecer a muchos grupos.

```
model User {
    id Int @id @default(autoincrement())
    name String
    email String @unique
    groups Group[] @relation("UserToGroup")
}
```

```
model Group {
    id Int @id @default(autoincrement())
    name String
    users User[] @relation("UserToGroup")
}
```

En este ejemplo, la relación entre `User` y `Group` se define en ambos modelos usando la propiedad `@relation`. El argumento `"UserToGroup"` se utiliza para indicar que ambas tablas están relacionadas. Prisma creará automáticamente una tabla de unión para almacenar las relaciones entre los modelos.

Enum

Los modelos enum se utilizan para definir un conjunto finito de valores que pueden ser asignados a una columna. Por ejemplo, un modelo `Todo` puede tener una columna `status` que solo puede ser uno de los valores `"done"`, `"in progress"` o `"todo"`.

```
model Todo {
  id     Int     @id @default(autoincrement())
  title  String
  status TodoStatus
}

enum TodoStatus {
  DONE
  IN_PROGRESS
  TODO
}
```

En este ejemplo, se define el modelo enum `TodoStatus`, que se utiliza como tipo para la columna `status` en el modelo `Todo`. Los valores de `TodoStatus` se utilizan para definir los valores posibles que pueden ser asignados a la columna. Recomendamos ver la documentación original de Prisma para saber más.

Propiedades

Existen algunas propiedades adicionales que pueden ser utilizadas para personalizar el comportamiento de las relaciones. Estas son principalmente dos, `onDelete` y `onUpdate`. Estas propiedades se usan para definir qué hacer cuando se borra o modifica un registro que tiene relaciones con otros registros. Estas propiedades tienen distintos valores posibles a adoptar para así comportarse de una u otra forma, estos son cinco; `Cascade`, `Restrict`, `NoAction`, `SetNull`, `SetDefault`.

onDelete

Esta propiedad define cómo se manejará la eliminación de registros en el modelo principal cuando hay registros relacionados en el modelo secundario. Se comporta diferente según que valor tenga, estos son los distintos comportamientos:

- `Cascade`: Cuando se elimina un registro en el modelo principal, también se eliminan automáticamente todos los registros relacionados en el modelo secundario.
- `Restrict`: Evita que se elimine un registro en el modelo principal si hay registros relacionados en el modelo secundario.
- `SetNull`: Establece los valores de la clave foránea (como el `authorId` en el ejemplo de la relación uno a muchos) en nulo en los registros relacionados en el modelo secundario cuando se elimina un registro en el modelo principal.
- `NoAction`: No se realiza ninguna acción en los registros relacionados en el modelo secundario cuando se elimina un registro en el modelo principal.
- `SetDefault`: Establece los valores de la clave foránea en su valor predeterminado en los registros relacionados en el modelo secundario cuando se elimina un registro en el modelo principal.

onUpdate

Esta propiedad define cómo se manejará la actualización de registros en el modelo principal cuando hay registros relacionados en el modelo secundaria. Este es el comportamiento según los posibles valores:

- `Cascade`: Actualiza automáticamente los registros relacionados en el modelo secundario con los nuevos valores cuando se actualiza un registro en el modelo principal.

`Restrict`: No permite actualizar un registro el modelo tabla principal si hay registros relacionados en el modelo secundario.
`SetNull`: Establece los valores de la clave foránea en `NULL` en los registros relacionados en el modelo secundario cuando se actualiza un registro el modelo tabla principal.
`NoAction`: No realiza ninguna acción en los registros relacionados en el modelo secundario cuando se actualiza un registro el modelo tabla principal.
`SetDefault`: Establece los valores de la clave foránea en su valor predeterminado en los registros relacionados en el modelo secundario cuando se actualiza un registro el modelo tabla principal.

Es importante mencionar que estos valores únicamente se aplican a las operaciones de actualización en el modelo principal, es decir, no tienen efecto sobre las operaciones de eliminación de registros. Para esto último se debe usar la propiedad `onDelete`.

Es posible que algunos valores estén o no disponibles según la base de datos que uses, te invito a ver la documentación.

Métodos

Una vez que se ha definido el modelo de datos y se ha configurado la conexión a la base de datos, toca cubrir la parte divertida de Prisma, sus métodos, estos son usados para reemplazar las tan conocidas consultas SQL, porque no, acá no usamos SQL nunca, así que cada vez que queramos hacer un movimiento en la base de datos, utilizaremos estos métodos de a continuación. Siempre que se tengan que usar los métodos, hay que crear un `PrismaClient` en el archivo de JavaScript introduciendo el siguiente bloque de código:

```
const { PrismaClient } = require('@prisma/client')
const prisma = new  PrismaClient()
```

Además, correr el siguiente comando:

```
npx prisma generate
```

Vale aclarar que todos los métodos se usan siguiendo la misma sintaxis de `prisma.[NombreDelModelo].[NombreDelMetodo]`. También que los archivos JSON tanto para introducirlos como parámetros en ciertos métodos como para recibir las respuestas de los mismos.

findMany

El método `findMany()` se utiliza para recuperar varios registros de una tabla en la base de datos. Por ejemplo, para recuperar todos los usuarios de la tabla `User`, se puede utilizar el siguiente código:

```
const users = await prisma.user.findMany();
```

findOne

El método `findOne()` se utiliza para recuperar un registro de una tabla en la base de datos. Por ejemplo, para recuperar un usuario de la tabla `User`, se puede utilizar el siguiente código:

```
const user = await prisma.user.findOne();
```

NOTA: El `findOne()` siempre devuelve el primero de todos los registros que reciba.

create

El método `create()` se utiliza para crear un nuevo registro en una tabla en la base de datos. Por ejemplo, para crear un nuevo usuario en la tabla `User`, se puede utilizar el siguiente código:

```
const newUser = await prisma.user.create({
  data: {
    name: "Marcelo Guiterrez",
    email: "MarceG@gmail.com",
    password: "contraseña"
  }
});
```

NOTA: En este caso, el parámetro `data` está en formato JSON, este formato lo vemos repetidas no solo en Prisma, sino que en JavaScript en general.

En el caso de que se quiera crear mas de un mismo registro en la misma tabla, se puede pasar a `data` una lista con un objeto distinto por registro. Por ejemplo:

```
const newUsers = await prisma.user.createMany({
  data: [
    { name: "Marcelo Guiterrez", email: "MarceG@gmail.com", password: "contraseña" },
    { name: "Gonzalo Vangue", email: "GonzaV@gmail.com", password: "incorrecta" },
    { name: "Juan Perez", email: "JuaniPerez@gmail.com", password: "123456" }
  ]
});
```

Es importante aclarar que la idea de pasar una lista con muchos registros puede ser usada en cualquier contexto, no solo en el de crear registros, sino también en el de actualizar o eliminar registros. Ademas de no solo en el de crear en la misma tabla, sino también en el de crear registros en tablas relacionadas, como en el caso de una aplicacion de mensajes, donde mas de una persona estaria relacionada, por ejemplo, con un solo mensaje.

update

El método `update()` se utiliza para actualizar un registro en una tabla en la base de datos. Por ejemplo, para actualizar el nombre del usuario con `id` igual a 1 en la tabla `User`, se puede utilizar el siguiente código:

```
const updatedUser = await prisma.user.update({
  where: {
    id: 1
  },
  data: {
    name: "Marcela Gutierrez"
  }
});
```

delete

El método `delete()` se utiliza para eliminar un registro de una tabla en la base de datos. Por ejemplo, para eliminar el usuario con `id` igual a 1 de la tabla `User`, se puede utilizar el siguiente código:

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

Recomendamos ver la documentación original de Prisma para saber más.

Filtrado y Ordenado

Como ya vimos en ejemplos anteriores, dentro del parámetro que le pasamos a cualquier método de Prisma, podemos pasarle otras propiedades que ayudan a poder refinar nuestras consultas y obtener solo la información que necesitamos. Veremos cómo utilizar funciones como `select`, `orderBy`, `contains`, `startWith`, `endWith` y más para filtrar y ordenar datos de manera efectiva. Además, veremos cómo utilizar la función `connect` para obtener información de modelos relacionados en nuestras consultas.

Select

Permite seleccionar qué campos deseas obtener de una entidad. Por ejemplo, si solo quieres obtener el nombre y el correo electrónico de los usuarios, puedes usar `select` para obtener solo esos campos.

```
const users = await prisma.user.findMany({
  select: {
    name: true,
    email: true
  }
})
```

Take

Permite limitar la cantidad de registros que devuelve el método `findMany()`. Por ejemplo, si deseas seleccionar los diez primeros usuarios de la base de datos.

```
const users = await prisma.user.findMany({
    take: 10
})
```

Include

Permite incluir campos de una entidad relacionada en la consulta. Por ejemplo, si deseas obtener todos los posts de un usuario y también incluir los comentarios de cada post, puedes usar `include` para obtener toda la información.

```
const user = await prisma.user.findUnique({
  where: {
    id: 1
  },
  include: {
    posts: {
      include: {
        comments: true
      }
    }
  }
})
```

Where

Permite filtrar los registros en función de los valores de sus campos. Por ejemplo, si solo quieres obtener los usuarios que tengan el nombre "Juan".

```
const users = await prisma.user.findMany({
  where: {
    name: "Juan"
  }
})
```

Contains

Permite buscar registros que contengan una cadena determinada. Por ejemplo, si deseas buscar todos los usuarios que tengan la palabra "programador" en su descripción.

```
const users = await prisma.user.findMany({
  where: {
    description: {
      contains: "programador"
    }
  }
})
```

startsWith y endsWith

Permiten buscar registros que comiencen o terminen con una cadena determinada. Por ejemplo, si deseas buscar todos los usuarios que tengan un apellido que empiece por "a".

```
const users = await prisma.user.findMany({
  where: {
    surname: {
      startsWith: "a"
    }
  }
})
```

orderBy

Permite ordenar los resultados por uno o más campos. Por ejemplo, si deseas obtener todos los usuarios ordenados por su nombre en orden alfabético.

```
const users = await prisma.user.findMany({
  orderBy: {
    name: "asc"
  }
})
```

Connect

Permite conectar dos entidades relacionadas en una sola consulta. Por ejemplo, si deseas obtener todos los comentarios que ha hecho un usuario en una publicación específica.

```
const comments = await prisma.comment.findMany({
  where: {
    post: {
      id: 1
    },
    author: {
      id: 2
    }
  }
})
```

Estas son solo algunas de las opciones de filtrado y ordenamiento disponibles en Prisma. Hay muchas más opciones y combinaciones posibles que puedes explorar en la documentación oficial de Prisma. También hay que tener en cuenta que estas opciones pueden encadenarse, permitiéndole al desarrollador, por ejemplo, seleccionar todos los nombres comenzados por la letra "p", que contengan la letra "h" y finalmente ordenarlos en orden alfabético. Hay una cantidad gigante de combinaciones posibles, pensado para poder facilitarle al máximo al desarrollador la tarea de interactuar con la base de datos.

Comandos

Acá vamos a ver algunos de los comandos más útiles en Prisma, que son esenciales para trabajar con bases de datos y realizar cambios en el esquema de la base de datos. También cómo es posible importar y exportar archivos SQL, lo que permite una mayor flexibilidad en la manipulación de los datos.

Migrate Dev

El comando `npx prisma migrate dev` se utiliza para ejecutar migraciones en una base de datos. Las migraciones son una forma de aplicar los cambios que hicimos en el modelo del `schema.prisma` a la base de datos local, además de tener un registro de cambios. La opción `dev` se utiliza para ejecutar migraciones en un entorno de desarrollo.

Migrate Deploy

El comando `npx prisma migrate deploy` despliega tu esquema de Prisma a la base de datos especificada en el archivo de configuración. Es una buena práctica ejecutar este comando después de hacer cambios en tu esquema de Prisma.

Prisma Studio

El comando `npx prisma studio` comando se utiliza para abrir Prisma Studio, una interfaz gráfica de usuario para explorar los datos almacenados en la base de datos. Prisma Studio es una herramienta muy útil para visualizar y manipular los datos en la base de datos.

Prisma Format

El comando `npx prisma format` formatea automáticamente tu archivo `schema.prisma` para que sea legible y esté bien estructurado. Generalmente después de cualquier cambio en el archivo `schema.prisma` se hace un `npx prisma format`.

DB Pull

El comando `npx prisma db pull` importa la estructura de tu base de datos actual en tu archivo `schema.prisma`. Es útil si necesitas actualizar tu archivo de esquema a partir de los cambios en la base de datos.

Estos son algunos de todos los posibles comandos que hay disponibles gracias a Prisma, recomendamos acceder a la documentación original para saber más, y en cualquier caso o problema particular, buscar en internet.