2. Explore Prisma Client

Goal

The goal of this lesson is get comfortable with the Prisma C
available database queries you can send with it. You'll learn
(like nested writes), filtering and pagination. Along the way,
introduce a second model with a *relation* to the `User` mode

Setup

You can continue working in the same `prisma-workshop` p
`script.ts` file contains a `main` function that is invoked ea

Hints

▼ ***Type yourself*, don't copy and paste**

To learn and really *understand* what you are doing for ea
paste the solution but type out the solution yourself (ev

▼ **Autocompletion**

Prisma Client provides a number of queries that you car
You can learn about these queries in the documentation
using *autocompletion*.

To invoke the autocompletion, you can open `src/index`
`main` function (you can delete the comment `// ... y`
`go here` that's currently there):

```
import { PrismaClient } from '@prisma/client
ent() async function main() { const result =
on will show up if you type this } main() .c
.finally(async () => await prisma.$disconnec
```

▼ Expand for a screenshot of the autocompletion

Once you typed the line `const result = await prism`
be shown that lets you select the options for composing
want to query or using another top-level function like `$`
Autocompletion is available for the *entire* query, includin
to provide!

▼ **Prisma Studio**

Prisma Studio is a GUI for your database that you can us
You can start Prisma Studio by running the following co

```
npx prisma studio
```

Tasks

At the end of each task, you can run the script using the foll

```
npm run dev
```

Task 1: Write a query to return *all* `User` records

To warm yourself up a bit, go and write a query to return *all*
Print the result to the console using `console.log`.

▼ Solution

```
import { PrismaClient } from '@prisma/client
ent() async function main() { const result =
console.log(result) } main() .catch((e) => c
c () => await prisma.$disconnect())
```

Task 2: Write a query to create a new `User` reco

In this task you'll create another `User` record. In your Prism
for `email` but *not* for `name`:

- `email`: `"alice@prisma.io"`

Can you find the query that lets you do that?

▼ Solution

```
import { PrismaClient } from '@prisma/client
ent() async function main() { const result =
ta: { email: "alice@prisma.io" } }) console.
) => console.error(e)) .finally(async () =>
```

Task 3: Write a query to update an existing `User`

In this task, you will update the `User` record you just create

- `name`: `"Alice"`

How can you update an existing database record with Prism

▼ Solution

```
import { PrismaClient } from '@prisma/client
PrismaClient() async function main() { const
prisma.user.update({ where: { email: "alice@
"Alice" } }) console.log(result) } main() .c
.finally(async () => await prisma.$disconnec
```

Task 4: Add a `Post` table to your database

To explore more interesting Prisma Client queries, let's expa
model and configure a *relation* between the existing and the

The new `Post` model should be shaped as follows:

- `id`: an auto-incrementing integer to uniquely identify e
- `title`: the title of a post; this field should be *required* i
- `content`: the content/body of the post; this field shoul
- `published`: indicates whether a post is published or nc
database; by default any post that is created should *not*

- `author` and `authorId`: configures a *relation* from a pc considered the author of the post; the relation should b necessarily need an author in the database; note that *al* meaning you'll need to add the second side of the relati model as well

▶ Solution

Once you have adjusted the Prisma schema and your two m apply the changes against your database:

```
npx prisma migrate dev --name add-post
```

Task 5: Write a query to create a new `Post` rec

In this task, you'll create a first `Post` record with the title `"`

▼ Solution

```
import { PrismaClient } from '@prisma/client
ent() async function main() { const result =
ta: { title: "Hello World" } }) console.log(
console.error(e)) .finally(async () => await
```

Task 6: Write a query to connect `User` and `Pos`

You now have several `User` records and exactly one `Post` connected via the `authorId` foreign key column in the data

When using Prisma Client, you don't need to manually set fo relations using Prisma Client's type-safe API. Can you figure record and ***connect*** it to an existing `User` record via the `e`

Use the editor's autocompletion to find out about the query

▼ Solution

```
import { PrismaClient } from "@prisma/client
ient(); async function main() { const result
where: { id: 1 }, data: { author: { connect:
}, }, }); console.log(result); } main() .cat
inally(async () => await prisma.$disconnect(
```

Task 7: Write a query to retrieve a single `User` r

In task 1, you learned how to fetch a list of records from the
retrieve a single `User` record with a Prisma Client query by

▼ Solution

```
import { PrismaClient } from "@prisma/client
ient(); async function main() { const result
e({ where: { email: "alice@prisma.io" } }) c
atch((e) => console.error(e)) .finally(async
t());
```

Note that you can use any *unique* field of a Prisma mode
argument, so in this case you could identify a `User` rec

Task 8: Write a query that selects only a subset

For this task, you can reuse the same `findMany` query for u
However, this time your goal is to only select a subset of the
specifically all returned objects should only contain the `id`

▼ Solution

```
import { PrismaClient } from "@prisma/client
PrismaClient(); async function main() { cons
prisma.user.findMany({ select: { id: true, n
console.log(result) } main() .catch((e) => c
.finally(async () => await prisma.$disconnec
```

Task 9: Write a nested query to *include* a relatio

You'll now start exploring more relation queries of Prisma Cl where you *include* a relation, concretely: Take your query fr the `Post` table in the result.

▼ Solution

```
import { PrismaClient } from "@prisma/client
ient(); async function main() { const result
e({ where: { email: "alice@prisma.io" }, inc
sole.dir(result, { depth: null })} main() .c
.finally(async () => await prisma.$disconnec
```

Notice that the the `result` of your query is fully typed! by Prisma Client, here's what it looks like:

```
const result: (User & { posts: Post[]; }) |
nd `User` types look as follows: type Post =
ntent: string | null published: boolean auth
r = { id: number name: string | null email:
```

Task 10: Write a nested write query to create a r `Post` record

In this task, you'll create a new `User` along with a new `Pos` (nested write) query. You can again use the autocompletion the documentation here.

▼ Solution

```
import { PrismaClient } from "@prisma/client
ient(); async function main() { const result
data: { name: "Nikolas", email: "burk@prisma
: "A practical introduction to Prisma" }, },
main() .catch((e) => console.error(e)) .fina
disconnect());
```

Task 11: Write a query that filters for users whos

For this task, you can again reuse the same `findMany` quer
Only that this time, you don't want to return *all* `User` record
which starts with the letter `"A"`. Can you find the right ope
condition?

▼ Solution

```
import { PrismaClient } from "@prisma/client
PrismaClient(); async function main() { cons
prisma.user.findMany({ where: { name: { star
console.log(result); } main() .catch((e) =>
.finally(async () => await prisma.$disconnec
```

Task 12: Write a pagination query

Prisma Client provides several ways to paginate over a list o
from before to return only the *third* and *fourth* `User` record

▼ Solution

```
import { PrismaClient } from "@prisma/client
PrismaClient(); async function main() { cons
prisma.user.findMany({ skip: 2, take: 2, });
.catch((e) => console.error(e)) .finally(asy
prisma.$disconnect());
```

Next steps

With theses tasks, you only scratched the surface of what's
Feel free to explore more queries and try out some of the or
features.