CMPS 350 · Web Development Fundamentals

Tutorial 07 · Prisma

This tutorial is based on the Prisma Get Started / Quickstart guide: `https://www.prisma.io/docs/getting-started/quickstart`.

1. Create a new Next 13.3 application using: `npx create-next-app@latest --experimental-app .`
2. Install the Prisma extension for Visual Studio Code from the marketplace.
3. Install the Prisma package: `npm install prisma --save-dev` and initialize it using SQLite as data source provider: `npx primsa init --datasource-provider sqlite`.
4. Update the environment variable `DATABASE_URL` in `.env` with the desired location of the database file, for example, `DATABASE_URL="file:data/dev.db"`.
5. Update the schema, `prisma/schema.json`, with data models for a one-to-many relationship between users and their posts. Each `User` has an `id`, a unique `email`, a `name`, and a list of `posts`. Each `Post` has an `id`, a `title`, a text `content`, a `published` status, and a corresponding `author`.

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

6. Create the supporting SQLite database: `npx prisma migrate dev --name init`. This will generate and execute a SQL migration against the database[1] to define its tables and relationships.[2]
7. Create a repository module, `utilities/repository.js`, to manage access to the data.
8. Install the client library: `npm install @prisma/client` and use it to access the data in the repository.

```
import { PrismaClient } from "@prisma/client";
const prisma = new PrismaClient();
```

The client is custom-tailored to the data model and must be generated using: `npx prisma generate` every time the model/schema is updated.

[1] `prisma db push` can be used instead if migrations are not needed.
[2] `prisma db pull` can be used instead with a preexisting database.

9. Use Prisma Studio: `npx prisma studio` to view the database tables and edit/filter their records if needed.

10. Create a `disconnect` method that closes a client connection that will be used after having executed a query:

```
async function disconnect() {
  try {
    await prisma.$disconnect();
  } catch (e) {
    console.error(e);
    await prisma.$disconnect();
    process.exit(1);
  }
}
```

11. Create an `seed` method that populates the database with sample data. Use `prisma.user.create` and `prisma.post.create`. It should call `disconnect` after executing the queries and before returning the results. The sample data can be procedurally generated using Faker (https://fakerjs.dev):

```
import { faker } from "@faker-js/faker";

export async function seed() {
  Array(Math.floor(Math.random() * 60))
    .fill()
    .forEach(
      async () =>
        await prisma.user.create({
          data: {
            email: faker.internet.email(),
            name: faker.name.fullName(),
            posts: {
              create: Array(Math.floor(Math.random() * 12))
                .fill()
                .map(() => ({
                  title: faker.commerce.productName(),
                  content: faker.lorem.text(),
                  published: Math.random() > 0.5,
                })),
            },
          },
        })
    );
  await disconnect();
}
```

12. Create a `readUsers` method that returns all users along with their posts. Use `prisma.user.findMany` and include the posts:

```
export async function readUsers() {
  const users = await prisma.user.findMany({
    include: {
      posts: true,
    },
  });
  await disconnect();
  return users;
}
```

13. Import and use the repository in `api/users/route.js` to implement a `GET` method that returns all users along with their posts using `readUsers`.

    ```
    import * as repo from "@/utilities/repository.js";

    export async function GET(request) {
      return Response.json(await repo.readUsers());
    }
    ```

14. Import and use the repository in `api/seed/route.js` to implement a `POST` method that seeds the data store using `seed`.
15. Test the routes using Postman.
16. In a long-running application, `prisma.$disconnect` should not be explicitly called. One solution is to not call `disconnect` and instead create a client module, `client.js`, then import it to cache the `PrismaClient` instance and reuse it across the application:

    ```
    import { PrismaClient } from "@prisma/client";
    let prisma = new PrismaClient();
    export default prisma;
    ```