CMPS 350 · Web Development Fundamentals

Tutorial 08 · Prisma[1]

1. Create a Next 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. The environment variable `DATABASE_URL` in `.env` can be updated 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[2,3] to define its tables and relationships.
7. Create a repository module, `utilities/repository.js`, to manage access to the data.
8. Install[4] 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 regenerated using: `npx prisma generate` when the schema is updated.

[1] This tutorial is based on the Prisma Get Started / Quickstart.
[2] `prisma db push` can be used instead if migrations are not needed.
[3] `prisma db pull` can be used instead with a preexisting database.
[4] `prisma migrate` already performs this step.

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, which 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 a `readUsers` method that returns all users along with their posts. It should call `disconnect` after executing the queries and before returning the results. 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;
}
```

12. 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());
}
```

13. Test the route using Postman.
14. Create a `prisma/seed.js`[5] script that populates the database with sample data using `prisma.user.create`. The sample data can be procedurally generated using Faker (https://fakerjs.dev):

```
const { PrismaClient } = require("@prisma/client");
const { faker } = require("@faker-js/faker");
const prisma = new PrismaClient();

const seed = async () => {
  try {
    Array(Math.floor(Math.random() * 60))
      .fill()
```

[5] Note that the CommonJS syntax must be used for imports and exports.

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

  seed();
```

15. Add the following lines at the top level of `package.json` to automatically seed the database in `npx prisma migrate dev` and `npx prisma migrate reset`. The script can also be invoked manually using `npx prisma db seed`.

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

16. In a long-running application, `prisma.$disconnect` should not be explicitly called. One approach is to not call `disconnect` and instead create a client module, `utilities/client.js`, then import it to cache the `PrismaClient` instance and reuse it across the application:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
export default prisma;
```