Seminar 07 - Working with the database

Agenda

- Database - best practices
- Using Prisma as the ORM tool to work with databases
- Repository pattern

Database - best practices

- Write ERD for the database
- Model the tables according to the ERD
- Deleting data: records should have a visibility attribute (f.e. `deletedAt`) - deleting tables can cause issues in large, tightly coupled schemas
- Separate tables for addresses, prices and data that can change over time
- Storing multimedia in the database - a BAD idea (when talking about relational DBs) - databases are often cached in-memory
- Primary keys should always be either UUIDs or integers with autoincrement function, try to avoid composite keys
- Joining many-to-many relations done via join tables

Prisma - Install

- Add TypeScript to the project in the same fashion as in the previous seminar
- Add Prisma to the project

```
# Prisma is a developer dependency, only the client is used at runtime!
npm i -D prisma
```

If your code does not compile, extend your `tsconfig.json` file (however, there should not be any issues)

```
  "compilerOptions": {
    "sourceMap": true,
    "outDir": "dist",
    "strict": true,
    "lib": ["esnext"],
    "esModuleInterop": true
  }
```

Prisma - Schema & Migrations

This command will bootstrap the Prisma in the project:

```
npx prisma init
```

Created files: `prisma/schema.prisma` and `.env` file with the database connection string.

Schema contains our table definitions.

```prisma
model Artist {
  id              Int      @default(autoincrement()) @id
  name            String   @db.VarChar(255)
  verified        Boolean  @default(false)
  profilePicture  String?
  coverPicture    String?
  description     String
  albums          Album[]
}

model Album {
  id              Int      @default(autoincrement()) @id
  artist          Artist   @relation(fields: [artistId], references: [id])
  artistId        Int
  name            String   @db.VarChar(255)
  releaseDate     DateTime
  description     String
  coverPicture    String?
}
```

Schema example

Note: for the rest of the slides, we're referencing this schema!

Connecting to the database - SQLite

- As we moved the Docker lecture to the end of the semester, we will be using SQLite as our database provider of choice (both during interations, and on seminars)
- We can connect Prisma to `sqlite` database - which is a database provider available (at least on the linux machines) on school computers
- Create a file `database.db` in the `prisma` folder
- Modify the portion of the `prisma.schema` file:

Prisma will stop looking for the `.env` file and connect to the `database.db` file via SQLite. Now you can follow along with the seminar!

Connecting to the database - Postgres (optional)

If you have your own computer and already know Docker/podman you can run a Postgres database in a container. We advise you to create a `docker-compose` file which will set the database up and add some other container to look into the database, such as `adminer`.

The connection string is stored in the `.env` file - Prisma uses it to create a connection to the DB:

```
# for postgres database
DATABASE_URL="postgresql://johndoe:randompassword@localhost:5432/mydb?schema=public"
```

- **NEVER** commit these files - they should never be tracked by the versioning software

Prisma - Schema & Migrations

After writing the schema, we need to generate a migration.

- Migration is a file with SQL definitions, which defines the database tables.
- Every schema change must be reflected by running another migration (which will update the DB) and re-compiling the Prisma client.

```
npx prisma migrate dev --name init
```

This command will **also generate a new client** with type definitions for us

Adding Prisma to the code

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

const prisma = new PrismaClient()

const main = async () => {
  // ... you will write your Prisma Client queries here
}

main()
  .catch(e => {
    throw e
  })
  .finally(async () => {
    await prisma.$disconnect()
  })
```

Repository pattern

- Separates the database logic from the rest of the application
- Creates an API to work with your database
  - The API stays the same, even if the underlying implementation is completely rewritten
  - Makes working with the db in your application (REST API, GraphQL app, ...) as simple, as calling a function (with correct parameters) and `await`ing the result

Read more here.

Usage (you will see more of repository pattern in the optional demo at the end of this presentation):

```
import userRepository from './repositories/user';

// reading all albums in the database
const result = await albumRepository.read.all();
```

CRUD operations in Prisma

Prisma allows several different CRUD (create, read, update, delete) operations:

- `findMany`, `findFirst`, `findUnique`: all read data - obvious from names
- `create`, `createMany`: Creates a record / creates many records in a batch query
- `update`, `updateMany`: Updates a single record / updates many records in a batch query
- `upsert`: Create OR update a record (updates an existing record, or creates it if it does not exist)
- `delete`, `deleteMany`: Deletes a single record / deletes many records in a batch query

Example:

```
// find all users
const artists = await prisma.artist.findMany();
```

Prisma queries

Prisma query is comprised of some parts:

- `where` field: specifies the conditions which we want to run the query with
- `select` field: which data we want to retrieve from the database (if not included, the whole model/record gets retrieved)
- `data` field: specifies what data we want to update / create
- `include` : joining data from relations in the response - does not work with `select` on the same level, `select` can also join the related records if we want to only retrieve some parts of the model/record!
- `orderBy` : ordering of the data - we want to let the db do the ordering whenever possible
- `take` : number of records to retrieve, can be used only in conjunction with `orderBy` to ensure deterministic behavior
- `skip` : enables pagination

And many more, see the whole client documentation for the detailed

Prisma query example

```
// find all albums where their description contains the word 'rap'
const albums = await prisma.album.findMany({
  where: {
    description: {
      contains: 'rap'
    },
  },
});
```

Prisma transactions

- Encapsulate a code that needs to either succeed as a whole or fail as a whole
- Either sequential or interactive
- On error, the transaction rolls back - as if it was never executed

Interactive transactions

- Should perform only the necessary operations
- Use them together with Isolation levels to avoid race conditions within transactions
- Use them with caution!

Read the whole transactions documentation for more details.

Prisma interactive transaction example

```
const result = await prisma.$transaction(async (transaction) => {
  // use "transaction" parameter of this async function instead of regular "prisma" calls
  const albums = transaction.album.findFirst({
    // whatever query here
  });

  if (albums) {
    // we can now write some logic within the transaction, whatever the condition
    // or intended reason for this custom logic is
  }

  return transaction.artist.update({
    // perform some operation that is dependent on the previous query
    // and previous logic within the transaction
  });
});
```

Many-to-many relationships: implicit & explicit

- Prisma can handle basic many to many relation by defining lists of items in both affected Prisma models in the schema
- In case you need to store more information than just the many to many relation, you need to create an explicit many to many relation by defining a **join table** with all necessary properties.
- We recommend using implicit relationships only if you don't wish to extend them in the future.

Exceptions from Prisma

As with everything, Prisma calls can also fail due to multiple reasons:

- Failed constraints during the query execution
- Conflicting query creation (using `select` together with `include` on the same level)
- Unable to connect to the database (for various reasons)
- Database does not have correct models (connection successful, but migrations have not been executed yet)

Always use Prisma queries within a try-catch block:

```
try {
  const something = await prisma. // write some prisma query(/ies) or transaction(s)
} catch (e) {
  // handle error
}
```

`Result` type

- In functional programming, `Result` types indicate the status of some operation which can fail
- They are null-safe, always returning some value

In TypeScript, we can use the `@badrap/result` npm package, which brings the `Result` type into TypeScript. An example (taken from the npm package page):

```
import { Result } from '@badrap/result';
const res = Math.random() < 0.5 ? Result.ok(1) : Result.err(new Error("oh no"));

if (res.isErr) {
  // TypeScript now knows that res is a Result.Err, and we can access res.error
  res.error; // Error("oh no")
}

if (res.isOk) {
  // TypeScript now knows that res is a Result.Ok, and we can access res.value
  res.value; // 1
}
```

For more information about results, you can read this wikipedia page.

Demo - complete tasks in the Prisma playground

Open Prisma playground and level up your Prisma knowledge!

Optional demo: repository pattern

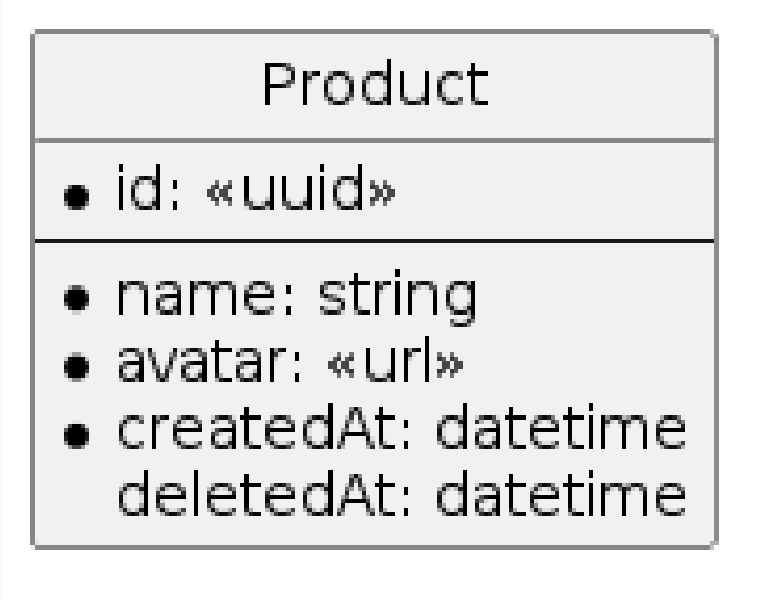
Create a repository pattern that allows working with the data defined by the provided ERD (next slide, and also in the template)

If you wish, you can create everything from scratch (the preferred way, to really learn how to create such project). The steps with the asterisk (*) have already been done by us in the template to speed the process up (the template uses SQLite as the database provider).

- *Create a TypeScript project in Node.js and add Prisma
- *Create a Prisma schema from the ERD, run migrations
- Create a repository pattern:
  - *Define repositories
  - *Define all possible operations (CRUD, in case needed define additional) over the database
  - *Write type definitions for input / output data from the repository functions
  - Write Prisma queries
  - Use the repository in some example code
    - Add script to `package.json` to execute the script

Optional demo: ERD

- A "complex" ERD for the database:



Hands on: Iteration 05 - Prisma & Repository pattern

You can find the assignment in GitLab issues.

Let's take a look together.

Before you start:

- Please check whether your tutor has already accepted your MR
- If they have, make sure you have merged your solution from the previous week

*Note: if your tutor has **not** seen your MR, it's completely ok. You do **not** need to have the previous iteration merged to be able to work on a new one - **iterations are independent**. However, if you **do** have an accepted MR that still has not been merged, make sure to merge it first.*