Week 08: Prisma queries & transactions, repository pattern

Agenda

- Recapitulation from the lectures
- Repository pattern
- Creating a simple repository

Let's remember what we learned during the last two lectures

CRUD operations in Prisma

Prisma allows several different CRUD (create, read, update, delete) operations. As in the previous :

- `findMany`, `findFirst`, `findUnique`: all read data (and behave differently) - 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

Let's make an example with a music streaming platform (*side note*: this will be a part of the third iteration's assignment)!

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

Prisma queries

Prisma query is comprised of some parts:

- `where` field: specifies the conditions under 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` : the 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 explanation

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
- When an error is encountered, 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 documentation about transactions 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 = await 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 await 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 relationships 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)
- The database does not have correct models (connection successful, but migrations have not been executed yet)

Always write Prisma queries within a try-catch block:

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

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
  - This separates the need to rewrite the whole app when there are database changes (migrating to a different DBMS, rewriting queries for efficiency, etc.) and isolates the implementation into its own "subpart"
  - Makes working with the database in your application (REST API, GraphQL app, gRPC microservice, etc.) as simple, as calling an `async` function (with correct parameters) and `await`ing the result

Read more here.

```
import { albumRepository } from './repository.ts';

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

Repository pattern

- As always, we don't want the repository to throw an exception
- We'll focus on error handling with the help of the `Result` pattern you've already seen several times and should be familiar with by now
- Look in the documentation of the `@badrap/result`

Let's code!

As always, the assignment zip can be found in the interactive syllabus