Backend Read

Lernziele

- [ ] Wissen über ORM und (`mongoose` als) ODM
- [ ] Verstehen, wie man ein `mongoose`-Schema schreibt
- [ ] Wissen, wie man eine Anwendung mit einer (lokalen) Datenbank über `mongoose` verbindet
- [ ] Wissen, wie man Daten mit einem `mongoose`-Modell liest

Was und warum mongoose

Um von deiner App auf eine MongoDB zuzugreifen, benötigen wir eine JavaScript-API. Diese API wird manchmal als Datenbank-Treiber bezeichnet (stell es dir wie deinen Druckertreiber vor).

Wir werden eine Bibliothek namens `mongoose` verwenden. Das ist ein ODM (Object Document Mapper).

Unterschied zwischen ORM und ODM

ORM (*Object Relation Mapping*):

- Technik zur Durchführung von CRUD-Operationen hauptsächlich für relationale Datenbanken (MySQL, PostgreSQL usw.),
- verwendet ein *objektorientiertes Paradigma*,
- wie eine Excel-Tabelle mit Zeilen und Spalten => man kann kein Feld zu einem Eintrag hinzufügen, das nicht für alle existiert,
- wird auf ein einziges Objekt für alle Einträge abgebildet.

ODM (*Object Document Mapping*):

- wie ORM, aber für nicht-relationale Datenbanken (MongoDB),
- verwendet ein *dokumentenorientiertes Paradigma*.

Gründe, `mongoose` als ORM zu verwenden

- Es hilft beim Erstellen eines **Schemas** und bei Abfragen der Datenbank (es ist auch unser DB-Treiber).
- Es muss auf dem Server laufen, da der Datenbankzugriff im Browser nicht sicher ist.
    - Denke daran: Wir haben bereits einen Server (= Next.js API-Routen).

DB-Verbindung

Um Daten aus einer Datenbank zu lesen und sie in unserer App zu verwenden, benötigen wir zwei Dinge:

- eine (lokale) Datenbank mit Dokumenten (z. B. über Witze),
- eine Verbindung zwischen dieser Datenbank und der Next.js-App mit `mongoose`.

Um die Verbindung herzustellen, folge diesen Schritten:

1. Installiere `mongoose` mit `npm install mongoose`.
2. Erstelle eine `.env.local`-Datei im Stammverzeichnis deines Projekts mit folgendem Inhalt:
   `MONGODB_URI=mongodb+srv://<username>:<password>@cluster0.<dein-cluster-id>.mongodb.net/jokes-database?retryWrites=true&w=majority`
   - Dateien, die `.env` heißen, enthalten Umgebungsvariablen: geheime Informationen wie Benutzernamen und Passwörter, die **du nicht mit anderen teilen solltest**.
   - Diese Dateien sollten von git in der `.gitignore`-Datei ignoriert werden.
   - Beachte den Aufbau des Inhalts: Die Variable heißt `MONGODB_URI` und hat den Wert `mongodb+srv://<username>:<password>@cluster0.<dein-cluster-id>.mongodb.net/jokes-database?retryWrites=true&w=majority`.
   - `jokes-database` ist der Name deiner Datenbank: dieser Wert kann variieren.
3. Erstelle eine `db/connect.js`-Datei und kopiere den Inhalt aus dem Next.js mongoose-Beispiel.
   - Beachte, dass diese Datei die `MONGODB_URI` verwendet, die wir gerade in `.env.local` eingerichtet haben, um eine Verbindung herzustellen.

Schema und Modelle

Wir müssen ein Schema erstellen, das den Datentyp der Dokumente in einer Sammlung beschreibt.

Wir verwenden dieses Schema, um ein Modell zu erstellen, mit dem wir mit der Datenbank interagieren können.

Beachte den Unterschied zwischen *Schema* und *Modell*:

- das *Schema* beschreibt die Struktur eines Dokuments,
- das *Modell* gibt uns eine Programmierschnittstelle zur Interaktion mit der Datenbank (wie das Suchen in der Datenbank, Aktualisieren usw.).

Ein Schema schreiben

Wir schreiben ein Schema in der entsprechenden Datei im `db/models`-Ordner, wie folgt:

- Beim Erstellen eines `new Schema` übergeben wir ein Objekt mit den Schlüssel-Wert-Paaren, die unsere Dokumente haben sollen, wie `joke`, das ein `String` und `required` ist.
- Wir müssen die `id` nicht definieren, da `mongoose` automatisch eine erstellt.
- Exportiere das Schema, um es in unserer Anwendung verfügbar zu machen.

```
// db/models/Joke.js
import mongoose from "mongoose";

const { Schema } = mongoose;

const jokeSchema = new Schema({
  joke: { type: String, required: true },
});

const Joke = mongoose.models.Joke || mongoose.model("Joke", jokeSchema);

export default Joke;
```

Weitere Hinweise:

- Der Name der Sammlung, auf der das Modell arbeitet, wird aus dem Modellnamen generiert, in diesem Fall "Joke" => "jokes".
- Du kannst die Methode `mongoose.model` mit einem dritten Argument aufrufen, das den Namen der Sammlung enthält.
- Wir müssen prüfen, ob das Modell mit dem Namen "Joke" bereits kompiliert wurde, und falls ja, das bereits kompilierte Modell verwenden. Deshalb verwenden wir den logischen ODER (`||`)-Operator.

Verwendung des Modells: Abfragen der DB (.find, .findById)

In unserer Next.js API-Route können wir jetzt einen Request-Handler schreiben, der

- sich mit der Datenbank über dbConnect() verbindet,
- das Modell verwendet, um ein Dokument zu suchen,
- und die Daten zurückgibt.

```
// api/jokes/index.js
import dbConnect from "../../../db/connect";
import Joke from "../../../db/models/Joke";

export default async function handler(request, response) {
  await dbConnect();

  if (request.method === "GET") {
    const jokes = await Joke.find();
    return response.status(200).json(jokes);
  } else {
    return response.status(405).json({ message: "Method not allowed" });
  }
}
```

`mongoose` kommt mit einer `.findById()`-Methode, die du in einer dynamischen Route verwenden kannst:

```
// api/jokes/[id].js
import dbConnect from "../../../db/connect";
import Joke from "../../../db/models/Joke";

export default async function handler(request, response) {
  await dbConnect();
  const { id } = request.query;

  if (request.method === "GET") {
    const joke = await Joke.findById(id);

    if (!joke) {
      return response.status(404).json({ status: "Not Found" });
    }
```

```
    response.status(200).json(joke);
  }
}
```

Beachte, dass MongoDB eine `_id` anstelle von `id` zurückgibt, sodass du möglicherweise dein Frontend anpassen musst, um die richtigen Informationen abzurufen.

> 📙 Du findest eine Referenz zu allen Methoden eines Modells in der mongoose-Dokumentation.

Verknüpfte Sammlungen mit `.populate()` abrufen

Stell dir vor, deine MongoDB hat zwei Sammlungen: `jokes` und `comments` zu diesen Witzen. Sie sind über die `commentIds` verknüpft.

Wenn du die `jokes` liest, möchtest du auch die Kommentare erhalten. Das kannst du ganz einfach erreichen, indem du beide Schemata verknüpfst und beim Abfragen der Datenbank einfach `.populate()` mit Method Chaining hinzufügst. Zuerst verknüpfst du die Schemata für `Joke` und `Comment`:

```
// verknüpfe die Schemata
const jokeSchema = new Schema({
  joke: { type: String, required: true },
  comments: { type: [Schema.Types.ObjectId], ref: "Comment" },
});

const commentSchema = new Schema({
  comment: { type: String, required: true },
  author: { type: String, required: true },
});

const Joke = mongoose.models.Joke || mongoose.model("Joke", jokeSchema);
const Comment =
  mongoose.models.Comment || mongoose.model("Comment", commentSchema);
```

Zweitens, wenn du aus der Datenbank liest, fülle die Kommentare auf:

```
const joke = await Joke.findById(id).populate("comments");
```

> 📙 Lies mehr über populate in der mongoose-Dokumentation.

Resources

- ORM vs. ODM