Databas-schema

- All information om hur vår databas ser ut finns i ./prisma/schema.prisma
  - De poster som heter "model" motsvarar "tables" i SQL
  - modellens innehåll motsvarar:

```
  model ExampleTable {

  id      Int         @id
  ^        ^           ^
  |        |           |
column  datatyp  restrictions

  }
```

- För att skapa relationer använder vi namnet på den model vi vill koppla till som datatyp:

```
model Movie {
  id              Int     @id @default(autoincrement())
  name            String
  duration        Int
  actor           Actor[] <- Många-till-många relation. En film kan ha flera skådespelare, en skådespelare kan vara med i flera filmer
  filmedInCountry Country @relation(fields: [countryId], references: [id]) < - En-till-många relation. En film filmades i ett land.
  countryId       Int
}

model Actor {
  id    Int     @id @default(autoincrement())
  name  String
  age   Int
  movie Movie[] <- Många-till-många relation. En film kan ha flera skådespelare, en skådespelare kan vara med i flera filmer
}

model Country {
  id    Int     @id @default(autoincrement())
  name  String
  Movie Movie[] < - Många filmer kan ha filmats i ett land
}
```

- Prisma kan skapa en-till-många relationer åt oss med hjälp av `npx prisma format`, här är ett exempel på detta:

```
Före:
model Post {
  id    Int     @id @default(autoincrement())
  user User
}
model User {
  id    Int     @id @default(autoincrement())
}
```

```
Efter:
model Post {
  id      Int   @id @default(autoincrement())
  user    User2 @relation(fields: [userId], references: [id])
  userId  Int
}

model User {
  id   Int    @id @default(autoincrement())
  Post Post[]
}
```

- Många-till-många relationer skapas automatiskt av prisma utan att vi behöver köra något kommando om båda modellerna innehåller en array av den kopplade modellen:

```
model Post {
  id   Int   @id @default(autoincrement())
  tags Tag[]
}

model Tag {
  id    Int    @id @default(autoincrement())
  posts Post[]
}
```

- Om ett värde är optional använder vi ? efter datatypen, vill vi att det ska ha ett default-värde skriver vi @default i den tredje kolumnen:

```
model Optional {
  id      Int       @id @default(autoincrement())
  content String?
  default String    @default("Default-värde")
}
```

tRPC-struktur

- tRPC är ramverket som låter oss kommunicera med Prisma via våra endpoints.
- Vi skapar våra endpoints och definierar hur de fungerar i ./src/server/api/routers
- För att de ska vara tillgängliga i vår frontend måste de också läggas in i .src/server/api/root.ts:

```ts
import { exampleRouter } from "~/server/api/routers/example";
import { createTRPCRouter } from "~/server/api/trpc";
import { exerciseRouter } from "./routers/exercise";
import { movieRouter } from "./routers/movie";

/**
 * This is the primary router for your server.
 *
 * All routers added in /api/routers should be manually added here.
 */
export const appRouter = createTRPCRouter({

 // Vad vår endpoint heter när vi anropar den i frontend, bör vara samma som model i Prisma
    |
    v
  example: exampleRouter,
             ^
             |
    // Definitionen av endpointen i /routers

  exercise: exerciseRouter,
  movie: movieRouter,
});

// export type definition of API
export type AppRouter = typeof appRouter;
```

- Våra endpoints i /routers ser ut såhär:

```
import { z } from "zod";

import { createTRPCRouter, publicProcedure } from "~/server/api/trpc";

// Namnet på vår router, det bör vara modelnamnet+Router
                    |
                    v
export const exerciseRouter = createTRPCRouter({
                                      ^
                                      |
    // Inbyggd funktion i tRPC som ser till att allt hänger ihop

// Endpointens namn
  |
  |         // En query hämtar information från databasen
  |                       |
  v                       v
getAll: publicProcedure.query(({ ctx }) => {
              ^                   ^
              |                   |
              |    // Ett objekt som kommer från tRPC, används för att prata med Prisma
              |
// Innebär att endpointen kan anropas av vem som helst (mer om detta senare i kursen)
    return ctx.db.exercise.findMany();
  }),

// Fortsättning på nästa sida
```

```
createExercise: publicProcedure
// Berättar att endpointen tar emot data
       |
       |        // zod-definitionen för hur input får se ut
       |                        |
       |   --------------------^---------------------------------
       v   v                                                    v
    .input(z.object({ name: z.string(), duration: z.number() }))
    .mutation(({ input, ctx }) => {
        ^          ^
        |          |
        |     // Det validerade input-objektet
        |
 // En mutation muterar/förändrar innehållet i databasen
      return ctx.db.exercise.create({
                       ^        ^
                       |        |
// Vilken tabell vi vill jobba med samt vad vi vill göra med den

// Data att skicka in i databasen
         |
         v
        data: {
          name: input.name,
          duration: input.duration,
        },
      });
    }),
});
```

- När vi har skapat definitionerna i ./src/server/api kan vi använda dem i vår frontend.

```
import { api } from "~/utils/api";
         ^
         |
  // Inbyggd funktion som ger oss tillgång till backenden

           // Modellens namn från root.ts och endpointens namn från exercise.ts
                           |        |
                           v        v
const allExercises = api.exercise.getAll.useQuery().data;
                                          ^         ^
                                          |         |
                                          |   // Resultatet av anropet
                                          |
                           // Inbyggd funktion på alla querys i tRPC

// Mutations anropas inte direkt, men måste skapas för att kunna användas när vi har informationen som behövs
                                                    |
                                                    v
const createExercise = api.exercise.createExercise.useMutation();

const onSubmit = async () => {
    // Funktionen från raden ovanför, att det är en asynkron mutation samt datat vi skickar in
               |          |                    |
               v          v                    v
  await createExercise.mutateAsync({ name: "name", duration: 10 });
};
```