

Filstruktur i t3:

```
.
├─ prisma
│   └─ schema.prisma // Definitionen för vår databasmodell
├─ src
│   ├── env.mjs // Definition för hur våra env-variabler ser ut, säkerställer också att vi inte råkar exponera fel variabler i frontend
│   ├── pages // Vår frontend-applikation, alla mappar i /pages blir till sökvägar i webbläsaren
│   │   ├── _app.tsx
│   │   ├── api // Inbyggd funktionalitet för att kommunicera med vårt api, vi navigerar inte hit manuellt
│   │   ├── example
│   │   │   ├── index.tsx // Där vi hamnar om vi går till localhost:3000/example
│   │   │   └─ test.tsx // Där vi hamnar om vi går till localhost:3000/example/test
│   │   └─ index.tsx // Startsidan för appen
│   ├── server // Vår backend
│   │   ├── auth.ts // Information om autentiseringslösningar
│   │   ├── db.ts // Prisma //> tRPC-konfiguration
│   │   └─ api // Mappen som innehåller definitionen för våra endpoints
│   │       ├── routers // Mappen som innehåller endpoints för våra olika modeller i databasen
│   │       │   └─ example.ts // Definitionen för vilka funktioner som ska kunna göras på example-endpointen,
│   │       │           // ska ha samma namn som modellen i schema.prisma men med liten bokstav
│   │       ├── trpc.ts // Förkonfigurerade lösningar för tRPC
│   │       └─ root.ts // Huvudkällan för information om våra endpoints, alla filer i /routers ska importeras här och läggas till inuti approuter-objektet
│   └─ utils
│       └─ api.ts // Fil som innehåller kopplingen mellan frontend och tRPC, importeras i alla react-komponenter som behöver komma åt data från databasen
├─ .env // Variabler som behövs för att t.ex. koppla till databasen eller autentisera mot olika inloggningstjänster
├─ package.json
└─ tsconfig.json
```

Prisma-kommandon

```
npx prisma db push // Skapar en databas utifrån hur vår schema.prisma-fil ser ut
```

```
npx prisma format // Skapar en-till-många relationer åt oss och formatterar vår schema.prisma-fil
```

```
npx prisma generate // Genererar nya typescript-typer, görs automatiskt i samband med db push
```

```
npx prisma studio // Ger oss ett webbgränssnitt där vi kan utforska datan i vår databas
```

OBS!

Om typescript ger konstiga fel efter en db push, pröva att starta om VSCode och se om det löser problemet.

Man kan också använda kortkommandot `ctrl-shift-P` och söka efter "Reload Window"

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  datatype attributes  
  
}
```

Prisma attributes

<https://www.prisma.io/docs/reference/api-reference/prisma-schema-reference#attributes>

- 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      <- Se förklaring kring hur en-till-många relationer hänger ihop om två sidor
}

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    User   @relation(fields: [userId], references: [id])
  userId  Int
}

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

- En-till-många relationer hänger ihop sådär:

```
model Post {  
  id      Int    @id @default(autoincrement())  
  Berättar vilken kolumn i den Post-modellen som ska användas för att referera till User-modellen  
      |  
      v  
  user    User @relation(fields: [userId], references: [id])  
      ^               ^  
      |               |  
      Berättar att det är en relation      Berättar vilken kolumn i User-modellen som ska användas som koppling  
  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`:

root.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
  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
.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 queries 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 });  
};
```

Flöde från Prisma till frontend

- Skapa en ny model i schema.prisma
- Skapa en router för modellen i server/api/routers
 - Lägg till de funktioner du vill kunna göra med modellen, t.ex. findAll, create osv.
- Importera och lägg till den i approutern i server/api/root.ts
- Skapa en mapp med samma namn som modellen och inuti den en index.tsx i pages-mappen
- Importera { api } från utils/api och använd det för att göra dina anrop i din komponent