

## CONTENTS



Voraussetzungen

Schritt 1 — Erstellen Ihres TypeScript-Projekts

Schritt 2 — Einrichten von Prisma mit PostgreSQL

Schritt 3 — Definieren des Datenmodells und Erstellen von Datenbanktabellen

Schritt 4 — Erkunden von Prisma Client-Abfragen in einem einfachen Skript

Schritt 5 — Implementieren Ihrer ersten REST-API-Route

Schritt 6 — Implementieren der verbleibenden REST-API-Routen

Zusammenfassung



# Erstellen einer REST-API mit Prisma und PostgreSQL

Published on September 12, 2020

PostgreSQL

API

TypeScript

Node.js

Databases

Docker



nikolasburk and Kathryn Hancox

Deutsch



Der Autor hat den [Diversity in Tech Fund](#) dazu ausgewählt, eine Spende im Rahmen des Programms [Write for DONations](#) zu erhalten.

## Einführung

[Prisma](#) ist ein Open-Source-basiertes Datenbank-Toolkit. Es besteht aus drei Haupttools:

- **Prisma Client:** Ein automatisch generierter und typensicherer Query Builder für Node.js und TypeScript.
- **Prisma Migrate:** Ein deklaratives Modellierungs- und Migrationssystem für Daten.
- **Prisma Studio:** Eine GUI zum Anzeigen und Bearbeiten von Daten in Ihrer Datenbank.

Diese Tools dienen dazu, die Produktivität von Anwendungsentwicklern in ihren Datenbank-Workflows zu steigern. Einer der größten Vorteile von Prisma ist die Ebene der Abstraktion, die möglich ist: Anstatt komplexe SQL-Abfragen oder Schemamigrationen zu erstellen, können Anwendungsentwickler bei der Arbeit mit ihrer Datenbank unter Verwendung von Prisma Daten intuitiver verwalten.

In diesem Tutorial erstellen Sie eine REST-API für eine kleine Blogging-Anwendung in [TypeScript](#) mithilfe von Prisma und eine [PostgreSQL](#)-Datenbank. Sie werden Ihre PostgreSQL-Datenbank mit [Docker](#) lokal einrichten und die REST-API mit [Express](#) implementieren. Am Ende des Tutorials verfügen Sie über einen Webserver, der auf Ihrem Rechner lokal ausgeführt wird und auf verschiedene HTTP-Anfragen reagieren sowie Daten in der Datenbank lesen und schreiben kann.

# Voraussetzungen

Dieses Tutorial setzt Folgendes voraus:

- [Node.js v10](#) oder höher, auf Ihrem Rechner installiert. Sie können zur Einrichtung einen der Leitfäden zum [Installieren von Node.js und Erstellen einer lokalen Entwicklungsumgebung](#) verwenden, die für Ihr Betriebssystem gelten.
- [Docker](#), auf Ihrem Rechner installiert (zum Ausführen der PostgreSQL-Datenbank). Sie können die Installation unter macOS und Windows über die [Docker-Website](#) vornehmen oder [Installieren und Verwenden von Docker](#) für Linux-Distributionen folgen.

Grundlegende Vertrautheit mit TypeScript und REST-APIs ist hilfreich, für dieses Tutorial jedoch nicht erforderlich.

## Schritt 1 – Erstellen Ihres TypeScript-Projekts

In diesem Schritt werden Sie mit `npm` ein einfaches TypeScript-Projekt einrichten. Dieses Projekt wird als Grundlage für die REST-API dienen, die Sie im Laufe dieses Tutorials erstellen werden.

Erstellen Sie zunächst ein neues Verzeichnis für Ihr Projekt:

Copy

```
$ mkdir my-blog
```

Navigieren Sie als Nächstes in das Verzeichnis und initialisieren Sie ein leeres `npm`-Projekt. Beachten Sie, dass die Option `-y` hier bedeutet, dass Sie die interaktiven Eingabeaufforderungen des Befehls überspringen. Um die Eingabeaufforderungen zu durchlaufen, entfernen Sie `-y` aus dem Befehl:

Copy

```
$ cd my-blog  
$ npm init -y
```

Weitere Details zu diesen Eingabeaufforderungen finden Sie in Schritt 1 unter [Verwenden von Node.js-Modulen mit npm und package.json](#).

Sie erhalten eine Ausgabe, die der folgenden ähnelt und die Standardantworten umfasst:

Output  
Wrote to ../../my-blog/package.json:

Products	>
Solutions	>

Developers	>
Partners	>
Pricing	

[Log in](#) [Sign up](#)[Blog](#)[Docs](#)[Get Support](#)[Contact Sales](#)[Tutorials](#)[Questions](#)[Product Docs](#)[Cloud Chats](#)[Q Search Community](#)

Die letzte Aufgabe besteht darin, eine `tsconfig.json`-Datei hinzuzufügen, um sicherzustellen, dass TypeScript für die von Ihnen erstellte Anwendung richtig konfiguriert ist.

Führen Sie den folgenden Befehl aus, um die Datei zu erstellen:

[Copy](#)

```
$ nano tsconfig.json
```

Fügen Sie in der Datei den folgenden JSON-Code hinzu:

my-blog/tsconfig.json

[Copy](#)

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

Speichern und schließen Sie die Datei.

Dies ist eine Standard- und Minimalkonfiguration für ein TypeScript-Projekt. Wenn Sie mehr über die einzelnen Eigenschaften der Konfigurationsdatei erfahren möchten, können Sie die [TypeScript-Dokumentation](#) konsultieren.

Sie haben Ihr einfaches TypeScript-Projekt mit `npm` eingerichtet. Als Nächstes werden Sie Ihre PostgreSQL-Datenbank mit Docker einrichten und Prisma damit verbinden.

## Schritt 2 – Einrichten von Prisma mit PostgreSQL

In diesem Schritt installieren Sie die [Prisma-CLI](#), erstellen Ihre erste [Prisma-Schemadatei](#), richten PostgreSQL mit Docker ein und verbinden Prisma damit. Das Prisma-Schema ist die wichtigste Konfigurationsdatei für Ihr Prisma-Setup und enthält das Datenbankschema.

Installieren Sie zunächst die Prisma-CLI mit dem folgenden Befehl:

Copy

```
$ npm install @prisma/cli --save-dev
```

Als bewährte Praxis wird empfohlen, [die Prisma-CLI in Ihrem Projekt lokal zu installieren](#) (und nicht im Rahmen einer globalen Installation). Dadurch lassen sich Versionskonflikte vermeiden, falls Sie mehr als ein Prisma-Projekt auf Ihrem Rechner verwenden.

Als Nächstes richten Sie mit Docker Ihre PostgreSQL-Datenbank ein. Erstellen Sie mit dem folgenden Befehl eine neue Docker Compose-Datei:

Copy

```
$ nano docker-compose.yml
```

Fügen Sie der neu erstellten Datei den folgenden Code hinzu:

my-blog/docker-compose.yml

Copy

```
version: '3.8'
services:
  postgres:
    image: postgres:10.3
    restart: always
    environment:
      - POSTGRES_USER= sammy
      - POSTGRES_PASSWORD= your_password
    volumes:
      - postgres:/var/lib/postgresql/data
    ports:
      - '5432:5432'
```

```
volumes:
  postgres:
```

Diese Docker Compose-Datei konfiguriert eine PostgreSQL-Datenbank, auf die über Port 5432 des Docker-Containers zugegriffen werden kann. Beachten Sie außerdem, dass die Anmeldedaten für die Datenbank aktuell `sammy` (Benutzer) und `your_password` (Passwort) lauten. Sie können diese Anmeldedaten in Ihren bevorzugten Benutzer und Ihr bevorzugtes Passwort ändern. Speichern und schließen Sie die Datei.

Fahren Sie nun fort und starten Sie den PostgreSQL-Datenbankserver mit dem folgenden Befehl:

Copy

```
$ docker-compose up -d
```

Die Ausgabe dieses Befehls wird in etwa wie folgt aussehen:

#### Output

```
Pulling postgres (postgres:10.3)...
10.3: Pulling from library/postgres
f2aa67a397c4: Pull complete
6de83ca23e55: Pull complete
...
Status: Downloaded newer image for postgres:10.3
Creating my-blog_postgres_1 ... done
```

Sie können mit folgendem Befehl überprüfen, ob der Datenbankserver ausgeführt wird:

Copy

```
$ docker ps
```

Dadurch erhalten Sie eine Aufgabe, die in etwa wie folgt aussieht:

#### Output

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
8547f8e007ba	postgres:10.3	"docker-entrypoint.s..."	3 seconds ago	Up 2 seconds

Nachdem der Datenbankserver ausgeführt wird, können Sie nun Ihr Prisma-Setup erstellen. Führen Sie den folgenden Befehl über die Prisma-CLI aus:

Copy

```
$ npx prisma init
```

Dadurch erhalten Sie folgende Ausgabe:

#### Output

```
✓ Your Prisma schema was created at prisma/schema.prisma.  
You can now open it in your favorite editor.
```

Beachten Sie, dass Sie als bewährte Praxis allen Aufrufen der Prisma-CLI `npx` voranstellen sollten. Dadurch wird sichergestellt, dass Sie Ihre lokale Installation verwenden.

Nachdem Sie den Befehl ausgeführt haben, erstellt die Prisma-CLI in Ihrem Projekt einen neuen Ordner namens `prisma`. Er enthält die folgenden zwei Dateien:

- `schema.prisma`: Die Hauptkonfigurationsdatei für Ihr Prisma-Projekt (schließt Ihr Datenmodell mit ein).
- `.env`: Eine [dotenv](#)-Datei zum Definieren Ihrer Datenbankverbindungs-URL.

Um sicherzustellen, dass Prisma den Speicherort Ihrer Datenbank kennt, öffnen Sie die Datei `.env` und passen Sie die Umgebungsvariable `DATABASE_URL` an.

Öffnen Sie zunächst die `.env`-Datei:

Copy

```
$ nano prisma/.env
```

Jetzt können Sie die Umgebungsvariable wie folgt setzen:

```
my-blog/prisma/.env
```

```
DATABASE_URL="postgresql:// sammy : your_password @localhost:5432/my-blog?schema=public"
```

Ändern Sie die Anmeldedaten für die Datenbank unbedingt auf jene, die Sie in der Docker Compose-Datei angegeben haben. Um mehr über das Format der Verbindungs-URL zu erfahren, besuchen Sie die [Prisma-Dokumentation](#).

Wenn Sie damit fertig sind, speichern und schließen Sie die Datei.

In diesem Schritt haben Sie Ihre PostgreSQL-Datenbank mit Docker eingerichtet, die Prisma-CLI installiert und Prisma über eine Umgebungsvariable mit der Datenbank verbunden. Im nächsten Abschnitt definieren Sie Ihr Datenmodell und erstellen Ihre Datenbanktabellen.

## Schritt 3 – Definieren des Datenmodells und Erstellen von Datenbanktabellen

In diesem Schritt definieren Sie Ihr [Datenmodell](#) in der Prisma-Schemadatei. Dieses Datenmodell wird dann mit [Prisma Migrate](#) der Datenbank zugeordnet; dadurch werden die SQL-Anweisungen

generiert und gesendet, um die Tabellen zu erstellen, die Ihrem Datenmodell entsprechen. Da Sie eine Blogging-Anwendung erstellen, werden die wichtigsten Entitäten der Anwendung Benutzer und Beiträge sein.

Prisma verwendet seine eigene [Datenmodellierungssprache](#) zum Definieren der Form Ihrer Anwendungsdaten.

Öffnen Sie zunächst Ihre `schema.prisma`-Datei mit dem folgenden Befehl:

Copy

```
$ nano prisma/schema.prisma
```

Fügen Sie nun folgende Modelldefinitionen hinzu. Sie können die Modelle am Ende der Datei platzieren, unmittelbar nach dem `generator client`-Block:

my-blog/prisma/schema.prisma

```
...
model User {
  id    Int    @default(autoincrement()) @id
  email String @unique
  name  String?
  posts Post[]
}

model Post {
  id        Int    @default(autoincrement()) @id
  title     String
  content   String?
  published Boolean @default(false)
  author    User?  @relation(fields: [authorId], references: [id])
  authorId  Int?
}
```

Speichern und schließen Sie die Datei.

Sie definieren zwei [Modelle](#) namens `User` und `Post`. Jedes von ihnen verfügt über eine Reihe von [Feldern](#), die die Eigenschaften des Modells darstellen. Die Modelle werden Datenbanktabellen zugeordnet; die Felder stellen die einzelnen Spalten dar.

Beachten Sie außerdem, dass es eine one-to-many-[Beziehung](#) zwischen den beiden Modellen gibt, die von den Beziehungsfeldern `posts` und `author` in `User` und `Post` angegeben werden. Das bedeutet, dass ein Benutzer mit verschiedenen Beiträgen verknüpft sein kann.

Nach Implementierung dieser Modelle können Sie nun unter Verwendung von Prisma Migrate die entsprechenden Tabellen in der Datenbank erstellen. Führen Sie in Ihrem Terminal folgenden Befehl aus:

Copy



```
$ npx prisma migrate save --experimental --create-db --name "init"
```

Dieser Befehl erstellt in Ihrem Dateisystem eine neue Migration. Hier finden Sie einen kurzen Überblick über die drei Optionen, die dem Befehl bereitgestellt werden:

- `--experimental`: Erforderlich, da Prisma Migrate derzeit in einem *experimentellen* Zustand ist.
- `--create-db`: Ermöglicht Prisma Migrate die Erstellung der Datenbank mit dem Namen `my-blog`, die in der Verbindungs-URL angegeben ist.
- `--name "init"`: Gibt den Namen der Migration an (wird zum Benennen des Migrationsordners verwendet, der in Ihrem Dateisystem erstellt wird).

Die Ausgabe dieses Befehls wird in etwa wie folgt aussehen:

#### Output

New datamodel:

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id      Int      @default(autoincrement()) @id
  email   String   @unique
  name    String?
  posts   Post[]
}

model Post {
  id          Int      @default(autoincrement()) @id
  title       String
  content     String?
  published   Boolean @default(false)
  author     User?    @relation(fields: [authorId], references: [id])
  authorId   Int?
}
```

Prisma Migrate just created your migration 20200811140708-init in

```
migrations/
├─ 20200811140708-init/
│   └─ steps.json
│   └─ schema.prisma
│   └─ README.md
```

Sie können die Migrationsdateien erkunden, die im Verzeichnis `prisma/migrations` erstellt wurden.

Um die Migration für Ihre Datenbank auszuführen und die Tabellen für Ihre Prisma-Modelle zu erstellen, führen Sie in Ihrem Terminal folgenden Befehl aus:

Copy

```
$ npx prisma migrate up --experimental
```

Sie erhalten die folgende Ausgabe:

#### Output

```
. . .
Checking the datasource for potential data loss...

Database Changes:

Migration          Database actions          Status
20200811140708-init 2 CreateTable statements.  Done 🚀

You can get the detailed db changes with prisma migrate up --experimental --verbose
Or read about them here:
  ./migrations/20200811140708-init/README.md

🚀 Done with 1 migration in 206ms.
```

Prisma Migrate generiert nun die SQL-Anweisungen, die für die Migration erforderlich sind, und sendet sie an die Datenbank. Im Folgenden sehen Sie die SQL-Anweisungen, die die Tabellen erstellt haben:

Copy

```
CREATE TABLE "public"."User" (
  "id" SERIAL,
  "email" text NOT NULL ,
  "name" text ,
  PRIMARY KEY ("id")
)

CREATE TABLE "public"."Post" (
  "id" SERIAL,
  "title" text NOT NULL ,
  "content" text ,
  "published" boolean NOT NULL DEFAULT false,
  "authorId" integer ,
  PRIMARY KEY ("id")
)

CREATE UNIQUE INDEX "User.email" ON "public"."User"("email")
```

```
ALTER TABLE "public"."Post" ADD FOREIGN KEY ("authorId") REFERENCES "public"."User"("id") ON DEL
```

In diesem Schritt haben Sie mit Prisma Migrate Ihr Datenmodell im Prisma-Schema definiert und die jeweiligen Datenbanktabellen erstellt. Im nächsten Schritt installieren Sie Prisma Client in Ihrem Projekt, damit Sie die Datenbank abfragen können.

## Schritt 4 – Erkunden von Prisma Client-Abfragen in einem einfachen Skript

Prisma Client ist ein automatisch generierter und typensicherer Query Builder, mit dem Sie aus einer Node.js- oder TypeScript-Anwendung Daten in einer Datenbank programmatisch lesen und schreiben können. Sie werden ihn für Datenbankzugriff in Ihren REST-API-Routen verwenden, indem Sie traditionelle ORMs, einfache SQL-Abfragen, benutzerdefinierte Datenzugriffsebenen oder andere Methoden zur Kommunikation mit einer Datenbank ersetzen.

In diesem Schritt installieren Sie Prisma Client und lernen die Abfragen kennen, die Sie damit senden können. Bevor Sie in den nächsten Schritten die Routen für Ihre REST-API implementieren, werden Sie zunächst einige der Prisma Client-Abfragen in einem einfachen ausführbaren Skript erkunden.

Zuerst installieren Sie Prisma Client in Ihrem Projekt, indem Sie Ihr Terminal öffnen und das Prisma Client `npm`-Paket installieren:

Copy

```
$ npm install @prisma/client
```

Erstellen Sie als Nächstes ein neues Verzeichnis namens `src`, das Ihre Quelldateien enthalten wird:

Copy

```
$ mkdir src
```

Erstellen Sie nun in dem neuen Verzeichnis eine TypeScript-Datei:

Copy

```
$ nano src/index.ts
```

Alle Prisma Client-Abfragen geben **promises** (Zusagen) zurück, auf die Sie in Ihrem Code warten können. Dazu müssen Sie die Abfragen in einer `async`-Funktion senden.

Fügen Sie den folgenden Codebaustein mit einer `async`-Funktion hinzu, die in Ihrem Skript ausgeführt wird:

my-blog/src/index.ts

Copy

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

const prisma = new PrismaClient()

async function main() {
  // ... your Prisma Client queries will go here
}

main()
  .catch((e) => console.error(e))
  .finally(async () => await prisma.disconnect())
```

Hier finden Sie einen kurzen Überblick über den Codebaustein:

1. Sie importieren den `PrismaClient`-Konstruktor aus dem zuvor installierten `@prisma/client` npm-Paket.
2. Sie instanziierten `PrismaClient`, indem Sie den Konstrukteur aufrufen, und erhalten eine Instanz namens `prisma`.
3. Sie definieren eine `async`-Funktion namens `main`, wo Sie als Nächstes Ihre Prisma Client-Abfragen hinzufügen werden.
4. Sie rufen die `main`-Funktion auf, fangen dabei alle möglichen Ausnahmen ab und stellen sicher, dass Prisma Client alle offenen Datenbankverbindungen schließt, indem Sie `prisma.disconnect()` aufrufen.

Nach Implementierung der `main`-Funktion können Sie mit dem Hinzufügen von Prisma Client-Abfragen in das Skript beginnen. Passen Sie `index.ts` wie folgt an:

my-blog/src/index.ts

Copy

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

const prisma = new PrismaClient()

async function main() {
  const newUser = await prisma.user.create({
    data: {
      name: 'Alice',
      email: 'alice@prisma.io',
      posts: {
        create: {
          title: 'Hello World',
        },
      },
    },
  })
  console.log('Created new user: ', newUser)
```

```
const allUsers = await prisma.user.findMany({
  include: { posts: true },
})
console.log('All users: ')
console.dir(allUsers, { depth: null })
}

main()
  .catch((e) => console.error(e))
  .finally(async () => await prisma.disconnect())
```

In diesem Code verwenden Sie zwei Prisma Client-Abfragen:

- **create**: Erstellt einen neuen user -Eintrag. Beachten Sie, dass Sie in Wahrheit ein *nested write* verwenden, was bedeutet, dass Sie in derselben Abfrage sowohl einen user - als auch einen post -Eintrag erstellen.
- **findMany**: Liest alle vorhandenen user -Einträge aus der Datenbank. Sie geben die **include** - Option an, wodurch zusätzlich auch die entsprechenden post -Einträge für die einzelnen user -Einträge geladen werden.

Führen Sie das Skript nun mit dem folgenden Befehl aus:

Copy

```
$ npx ts-node src/index.ts
```

Sie erhalten in Ihrem Terminal folgende Ausgabe:

#### Output

```
Created new user: { id: 1, email: 'alice@prisma.io', name: 'Alice' }
[
  {
    id: 1,
    email: 'alice@prisma.io',
    name: 'Alice',
    posts: [
      {
        id: 1,
        title: 'Hello World',
        content: null,
        published: false,
        authorId: 1
      }
    ]
  }
]
```

**Anmerkung:** Wenn Sie eine Datenbank-GUI verwenden, können Sie überprüfen, ob die Daten erstellt wurden, indem Sie sich die Tabellen `User` und `Post` ansehen. Alternativ können Sie die Daten in Prisma Studio erkunden, indem Sie `npx prisma studio --experimental` ausführen.

Sie haben Prisma Client nun verwendet, um Daten in Ihrer Datenbank zu lesen und zu schreiben. In den verbleibenden Schritten wenden Sie dieses neue Wissen an, um die Routen für eine beispielhafte REST-API zu implementieren.

## Schritt 5 – Implementieren Ihrer ersten REST-API-Route

In diesem Schritt installieren Sie [Express](#) in Ihrer Anwendung. Express ist ein beliebtes Webframework für Node.js, das Sie zur Implementierung der REST-API-Routen in diesem Projekt verwenden werden. Die erste Route, die Sie implementieren werden, erlaubt es, mithilfe einer GET - Anfrage alle Benutzer von der API abzurufen. Die Benutzerdaten werden mit Prisma Client aus der Datenbank abgerufen.

Installieren Sie dann Express mit dem folgenden Befehl:

Copy

```
$ npm install express
```

Da Sie TypeScript verwenden, sollten Sie die jeweiligen Typen auch als Entwicklungsabhängigkeiten installieren. Führen Sie dazu folgenden Befehl aus:

Copy

```
$ npm install @types/express --save-dev
```

Nach Implementierung der Abhängigkeiten können Sie Ihre Express-Anwendung einrichten.

Öffnen Sie dazu erneut Ihre zentrale Quelldatei:

Copy

```
$ nano src/index.ts
```

Löschen Sie nun den ganzen Code in `index.ts` und ersetzen Sie ihn durch Folgendes, um Ihre REST-API zu starten:

my-blog/src/index.ts

Copy

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

const prisma = new PrismaClient()
const app = express()

app.use(express.json())
```

```
// ... your REST API routes will go here

app.listen(3000, () => {
  console.log('REST API server ready at: http://localhost:3000'),
})
```

Hier finden Sie eine kurze Aufschlüsselung des Codes:

1. Sie importieren `PrismaClient` und `express` aus den jeweiligen `npm`-Paketen.
2. Sie instanziierten `PrismaClient`, indem Sie den Konstruktor aufrufen, und erhalten eine Instanz namens `prisma`.
3. Sie erstellen Ihre Express-Anwendung, indem Sie `express()` aufrufen.
4. Sie fügen die Middleware `express.json()` hinzu, um sicherzustellen, dass Express JSON-Daten ordnungsgemäß verarbeiten kann.
5. Sie starten den Server unter Port `3000`.

Jetzt können Sie Ihre erste Route implementieren. Fügen Sie zwischen den Aufrufen für `app.use` und `app.listen` folgenden Code hinzu:

my-blog/src/index.ts

Copy

```
...
app.use(express.json())

app.get('/users', async (req, res) => {
  const users = await prisma.user.findMany()
  res.json(users)
})

app.listen(3000, () => {
  console.log('REST API server ready at: http://localhost:3000'),
})
```

Speichern und schließen Sie anschließend Ihre Datei. Starten Sie dann mit dem folgenden Befehl Ihren lokalen Webserver:

Copy

```
$ npx ts-node src/index.ts
```

Sie erhalten die folgende Ausgabe:

Output

```
REST API server ready at: http://localhost:3000
```

Um auf die Route `/users` zuzugreifen, können Sie Ihren Browser auf `http://localhost:3000/users` oder einen anderen HTTP-Client verweisen.

In diesem Tutorial testen Sie alle REST-API-Routen mit `curl` (einem Terminal-basierten HTTP-Client).

**Anmerkung:** Wenn Sie lieber einen GUI-basierten HTTP-Client verwenden, können Sie Alternativen wie [Postwoman](#) oder den [Advanced REST Client](#) verwenden.

Öffnen Sie zum Testen Ihrer Route ein neues Terminalfenster oder eine Registerkarte (damit Ihr lokaler Webserver weiter ausgeführt werden kann) und führen Sie folgenden Befehl aus:

Copy

```
$ curl http://localhost:3000/users
```

Sie erhalten die im vorherigen Schritt erstellten `user`-Daten:

Output

```
[{"id":1,"email":"alice@prisma.io","name":"Alice"}]
```

Beachten Sie, dass das `posts`-Array diesmal nicht enthalten ist. Das liegt daran, dass Sie die Option `include` nicht an den `findMany`-Aufruf in der Implementierung der Route `/users` übergeben.

Sie haben unter `/users` Ihre erste REST-API-Route implementiert. Im nächsten Schritt implementieren Sie die verbleibenden REST-API-Routen, um Ihrer API weitere Funktionen hinzuzufügen.

## Schritt 6 – Implementieren der verbleibenden REST-API-Routen

In diesem Schritt implementieren Sie die verbleibenden REST-API-Routen für Ihre Blogging-Anwendung. Am Ende wird Ihr Webserver verschiedene `GET`-, `POST`-, `PUT`- und `DELETE`-Anfragen bereitstellen.

Hier finden Sie einen Überblick über die verschiedenen Routen, die Sie implementieren werden:

HTTP-Methode	Route	Beschreibung
GET	<code>/feed</code>	Ruft alle <i>veröffentlichten</i> Beiträge ab.
GET	<code>/post/:id</code>	Ruft einen einzelnen Beitrag anhand seiner ID ab.
POST	<code>/user</code>	Erstellt einen neuen Benutzer.



HTTP-Methode	Route	Beschreibung
POST	/post	Erstellt einen neuen Beitrag (als <i>Entwurf</i> ).
PUT	/post/publish/:id	Setzt das <code>published</code> -Feld eines Beitrags auf <code>true</code> .
DELETE	post/:id	Löscht einen Beitrag anhand seiner ID.

Fahren Sie fort und implementieren Sie zunächst die verbleibenden `GET`-Routen.

Öffnen Sie die Datei `index.ts` mit dem folgenden Befehl:

Copy

```
$ nano src/index.ts
```

Fügen Sie dann im Anschluss an die Implementierung der Route `/users` folgenden Code hinzu:

my-blog/src/index.ts

Copy

```
. . .

app.get('/feed', async (req, res) => {
  const posts = await prisma.post.findMany({
    where: { published: true },
    include: { author: true }
  })
  res.json(posts)
})

app.get(`/post/:id`, async (req, res) => {
  const { id } = req.params
  const post = await prisma.post.findOne({
    where: { id: Number(id) },
  })
  res.json(post)
})

app.listen(3000, () => {
  console.log('REST API server ready at: http://localhost:3000'),
})
```

Speichern und schließen Sie Ihre Datei.

Dieser Code implementiert die API-Routen für zwei `GET`-Anfragen:

- `/feed`: Gibt eine Liste mit veröffentlichten Beiträgen zurück.
- `/post/:id`: Gibt einen einzelnen Beitrag anhand seiner ID zurück.

Prisma Client wird in beiden Implementierungen verwendet. In der Implementierung der Route `/feed` filtert die Abfrage, die Sie mit Prisma Client senden, nach allen `post`-Einträgen, bei denen die Spalte `published` den Wert `true` enthält. Außerdem nutzt die Prisma Client-Abfrage `include`, um die entsprechenden `author`-Informationen für die einzelnen Beiträge abzurufen. In der Implementierung der Route `/post/:id` übergeben Sie die ID, die aus dem URL-Pfad abgerufen wird, um einen bestimmten `post`-Eintrag aus der Datenbank zu lesen.

Sie können den Server anhalten, indem Sie auf Ihrer Tastatur `strg+c` drücken. Starten Sie dann den Server neu:

Copy

```
$ npx ts-node src/index.ts
```

Um die Route `/feed` zu testen, können Sie folgenden `curl`-Befehl verwenden:

Copy

```
$ curl http://localhost:3000/feed
```

Da noch keine Beiträge veröffentlicht wurden, ist die Antwort ein leeres Array:

Output

```
[]
```

Um die Route `/post/:id` zu testen, können Sie folgenden `curl`-Befehl verwenden:

Copy

```
$ curl http://localhost:3000/post/1
```

Dadurch wird der von Ihnen ursprünglich erstellte Beitrag zurückgegeben:

Output

```
{"id":1,"title":"Hello World","content":null,"published":false,"authorId":1}
```

Implementieren Sie als Nächstes die beiden `post`-Routen. Fügen Sie nach den Implementierungen der drei `GET`-Routen folgenden Code zu `index.ts` hinzu:

my-blog/src/index.ts

Copy

```

. . .

app.post(`/user`, async (req, res) => {
  const result = await prisma.user.create({
    data: { ...req.body },
  })
  res.json(result)
})

app.post(`/post`, async (req, res) => {
  const { title, content, authorEmail } = req.body
  const result = await prisma.post.create({
    data: {
      title,
      content,
      published: false,
      author: { connect: { email: authorEmail } },
    },
  })
  res.json(result)
})

app.listen(3000, () =>
  console.log('REST API server ready at: http://localhost:3000'),
)

```

Wenn Sie damit fertig sind, speichern und schließen Sie die Datei.

Dieser Code implementiert die API-Routen für zwei `POST`-Anfragen:

- `/user`: Erstellt in der Datenbank einen neuen Benutzer.
- `/post`: Erstellt in der Datenbank einen neuen Beitrag.

Wie zuvor wird in beiden Implementierungen Prisma Client verwendet. In der Implementierung der Route `/user` übergeben Sie die Werte aus dem Haupttext der HTTP-Anfrage an die Prisma Client-Abfrage `create`.

Die Route `/post` ist etwas aufwendiger: Hier können Sie die Werte aus dem Haupttext der HTTP-Anfrage nicht direkt übergeben; stattdessen müssen Sie sie zunächst manuell extrahieren, um sie dann an die Prisma Client-Abfrage zu übergeben. Der Grund dafür besteht darin, dass die Struktur von JSON im Haupttext der Anfrage nicht mit der Struktur übereinstimmt, die Prisma Client erwartet. Daher müssen Sie die erwartete Struktur manuell einrichten.

Sie können die neuen Routen testen, indem Sie den Server mit `strg+C` anhalten. Starten Sie dann den Server neu:

Copy

```
$ npx ts-node src/index.ts
```

Um über die Route `/user` einen neuen Benutzer zu erstellen, können Sie mit `curl` folgende `POST`-Anfrage senden:

Copy

```
$ curl -X POST -H "Content-Type: application/json" -d '{"name":"Bob", "email":"bob@prisma.io"}
```

Dadurch wird in der Datenbank ein neuer Benutzer erstellt und folgende Ausgabe ausgedruckt:

Output

```
{"id":2,"email":"bob@prisma.io","name":"Bob"}
```

Um über die Route `/post` einen neuen Beitrag zu erstellen, können Sie mit `curl` folgende `POST`-Anfrage senden:

Copy

```
$ curl -X POST -H "Content-Type: application/json" -d '{"title":"I am Bob", "authorEmail":"bob
```

Dadurch wird in der Datenbank ein neuer Beitrag erstellt und unter Verwendung der E-Mail-Adresse `bob@prisma.io` mit dem Benutzer verbunden. Sie erhalten die folgende Ausgabe:

Output

```
{"id":2,"title":"I am Bob","content":null,"published":false,"authorId":2}
```

Schließlich können Sie die Routen `PUT` und `DELETE` implementieren.

Öffnen Sie die Datei `index.ts` mit dem folgenden Befehl:

Copy

```
$ nano src/index.ts
```

Fügen Sie dann nach der Implementierung der beiden `POST`-Routen den hervorgehobenen Code hinzu:

my-blog/src/index.ts

Copy

```
...  
  
app.put('/post/publish/:id', async (req, res) => {  
  const { id } = req.params
```

```

    const post = await prisma.post.update({
      where: { id: Number(id) },
      data: { published: true },
    })
    res.json(post)
  })

  app.delete(`/post/:id`, async (req, res) => {
    const { id } = req.params
    const post = await prisma.post.delete({
      where: { id: Number(id) },
    })
    res.json(post)
  })

  app.listen(3000, () =>
    console.log('REST API server ready at: http://localhost:3000'),
  )

```

Speichern und schließen Sie Ihre Datei.

Dieser Code implementiert die API-Routen für eine PUT - sowie eine DELETE -Anfrage.

- `/post/publish/:id` (PUT): Veröffentlicht einen Beitrag anhand seiner ID.
- `/post/:id` (DELETE) Löscht einen Beitrag anhand seiner ID.

Erneut wird Prisma Client in beiden Implementierungen verwendet. In der Implementierung der Route `/post/publish/:id` wird die ID des zu veröffentlichenden Beitrags von der URL abgerufen und an die `update`-Abfrage von Prisma Client übergeben. Die Implementierung der Route `/post/:id` zum Löschen eines Beitrags in der Datenbank ruft außerdem die Beitrags-ID aus der URL ab und übergibt sie an die `delete`-Abfrage von Prisma Client.

Halten Sie den Server mit `strg+c` auf Ihrer Tastatur erneut an. Starten Sie dann den Server neu:

Copy

```
$ npx ts-node src/index.ts
```

Sie können die Route PUT mit dem folgenden `curl`-Befehl testen:

Copy

```
$ curl -X PUT http://localhost:3000/post/publish/2
```

Dadurch wird der Beitrag mit einem ID-Wert von 2 veröffentlicht. Wenn Sie die Anfrage `/feed` neu senden, wird dieser Beitrag nun in die Antwort aufgenommen.

Abschließend können Sie die Route DELETE mit dem folgenden `curl`-Befehl testen:

```
$ curl -X DELETE http://localhost:3000/post/1
```

Dadurch wird der Beitrag mit dem ID-Wert von 1 gelöscht. Um zu überprüfen, ob der Beitrag mit dieser ID gelöscht wurde, können Sie erneut eine GET-Anfrage an die Route `/post/1` senden.

In diesem Schritt haben Sie die verbleibenden REST-API-Routen für Ihre Blogging-Anwendung implementiert. Die API reagiert nun auf verschiedene GET -, POST -, PUT - und DELETE -Anfragen und implementiert Funktionen zum Lesen und Schreiben von Daten in der Datenbank.

## Zusammenfassung

In diesem Artikel haben Sie eine REST-API mit einer Reihe von verschiedenen Routen erstellt, um Benutzer- und Beitragsdaten für eine beispielhafte Blogging-Anwendung zu erstellen, zu lesen, zu aktualisieren und zu löschen. Innerhalb der API-Routen haben Sie den Prisma Client zum Senden der entsprechenden Abfragen an Ihre Datenbank verwendet.

Als Nächstes können Sie weitere API-Routen implementieren oder Ihr Datenbankschema mithilfe von Prisma Migrate erweitern. Konsultieren Sie die [Prisma-Dokumentation](#), um mehr über verschiedene Aspekte von Prisma zu erfahren und einige sofort einsatzbereite Beispielprojekte zu erkunden (im Repository `prisma-examples`). Verwenden Sie dazu Tools wie [GraphQL](#) oder [gRPC APIs](#).

Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases.

[Learn more about our products →](#)

## About the authors



[nikolasburk](#) Author



[Kathryn Hancox](#) Editor

Still looking for an answer?

Ask a question

Search for more help

Was this helpful?

Yes

No



## Comments

### Leave a comment

**B** *I* U H<sub>1</sub> H<sub>2</sub> H<sub>3</sub> “,”



Leave a comment...

This textbox defaults to using **Markdown** to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment



This work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.

### Try DigitalOcean for free

Click below to sign up and get **\$200 of credit** to try our products over 60 days!

Sign up

### Popular Topics

AI/ML

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

[All tutorials →](#)

[Talk to an expert →](#)



## Become a contributor for community

Get paid to write technical tutorials and select a tech-focused charity to receive a matching donation.

[Sign Up →](#)





## DigitalOcean Documentation

Full documentation for every DigitalOcean product.

[Learn more →](#)



## Resources for startups and SMBs

The Wave has everything you need to know about building a business, from raising funding to marketing your product.

**Learn more** →

## Get our newsletter

Stay up to date by signing up for DigitalOcean's Infrastructure as a Newsletter.

Email address

Submit

New accounts only. By submitting your email you agree to our [Privacy Policy](#)

## The developer cloud

Scale up as you grow — whether you're running one virtual machine or ten thousand.

**View all products**



# Get started for free

Sign up and get \$200 in credit for your first 60 days  
with DigitalOcean.\*

[Get started](#)



\*This promotional offer applies to new accounts only.

<b>Company</b>	▼
<hr/>	
<b>Products</b>	▼
<hr/>	
<b>Resources</b>	▼
<hr/>	
<b>Solutions</b>	▼
<hr/>	
<b>Contact</b>	▼

