UniSenac
Centro Universitário RS

Senac
Fecomércio
Sesc

Desenvolvimento de
Serviços e APIs

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas
Prof. Edécio Fernando Iepsen

03 INTRODUÇÃO

04 JUSTIFICATIVA

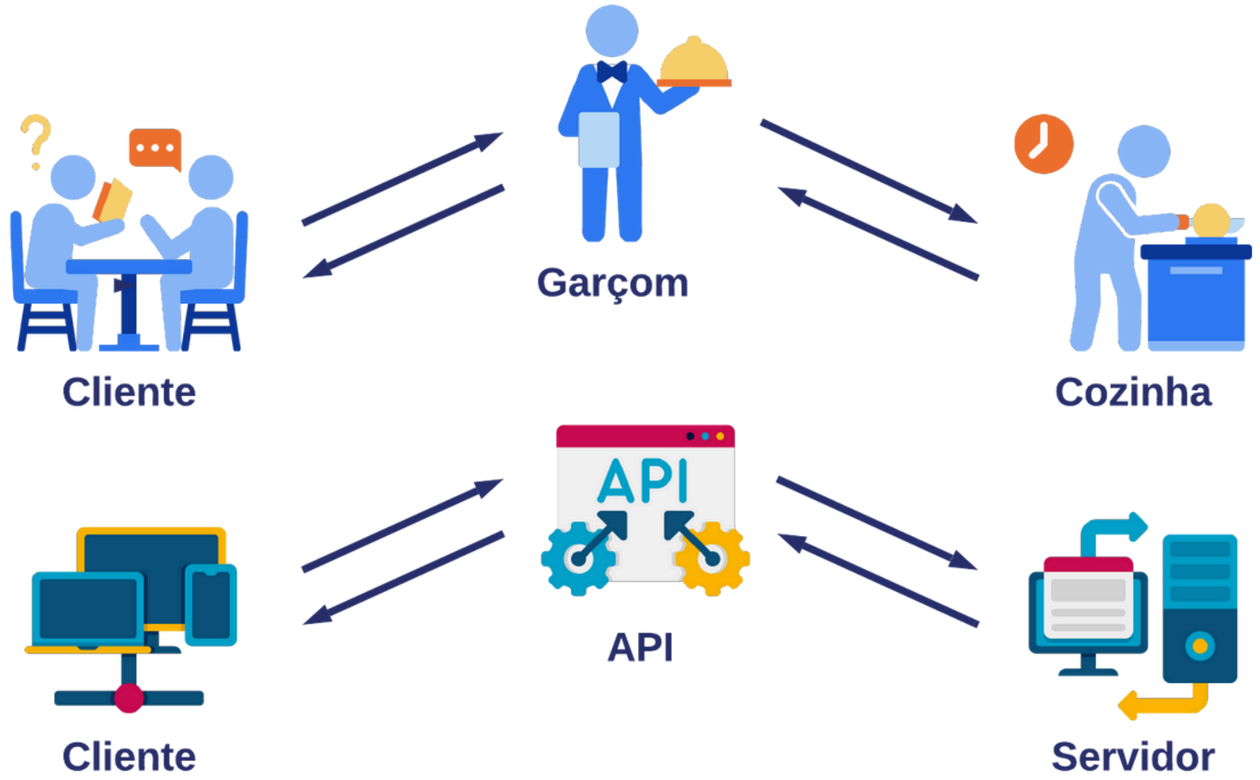
05 OBJETIVOS

06 REVISÃO TEÓRICA

07 METODOLOGIA

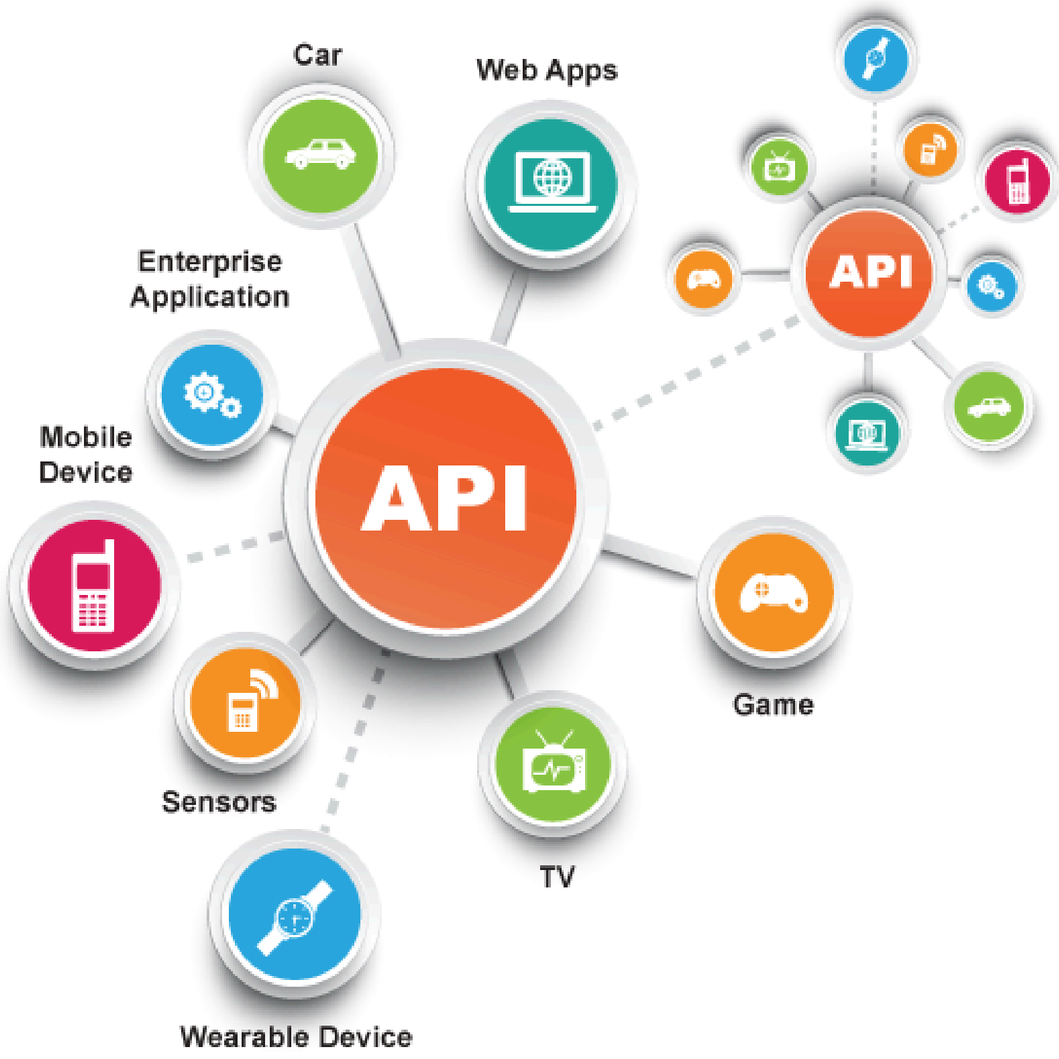
08 RESULTADOS

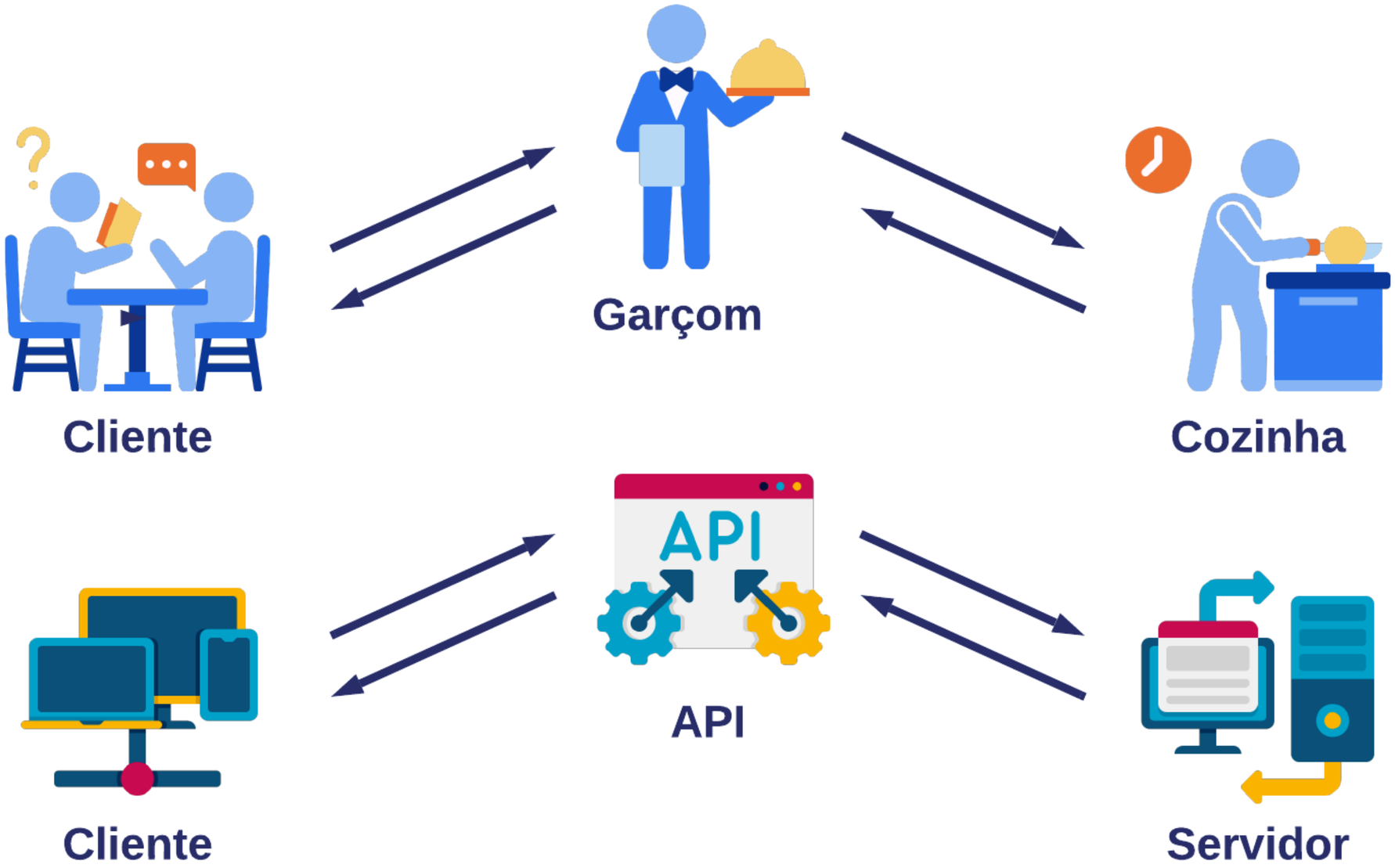
09 CONCLUSÃO



Revisando...

Uma API (Application Programming Interface) é um conjunto de regras e definições que permite a comunicação entre sistemas, facilitando a troca de dados e funcionalidades de forma padronizada.





https://analisemacro.com.br/data-science/o-que-sao-apis-e-como-utiliza-las-no-python/

O cliente faz o pedido ao garçom e este repassa à cozinha. A cozinha prepara o pedido para o garçom retirar e entregar ao cliente. APIs funcionam basicamente da mesma forma.

Cliente = Cliente (i.e., usuário solicitante, navegador, app);

Garçom = API (i.e., Application Programming Interface, comunicação entre cliente e servidor);

Cozinha = Servidor (i.e., processa pedido, entrega uma resposta)

Em termos práticos, **APIs são muito utilizadas para transmissão de dados.** Pacotes e funções das linguagens são usados para:

Fazer uma **requisição** para a API (i.e., apontar uma URL base e parâmetros/filtros da requisição de dados);

Obter uma **resposta** da API (i.e., resultado bem/mal sucedido, conteúdo/corpo);

Transformar a resposta em um **objeto da linguagem** (i.e., de XML/JSON/etc. para uma tabela de dados).

JSON

JSON (JavaScript Object Notation) é um formato de dados que armazena informações estruturadas em texto. É usado, principalmente, para transferir dados entre sistemas.

Características

- É um formato de dados simples e leve
- É compatível com diversas linguagens de programação
- É fácil de ler, escrever e interpretar
- É uma notação para a transferência de dados

Função ExpressJS express.json()

Última atualização: 26 de fevereiro de 2025

A função **express.json()** é um middleware integrado no Express que é usado para analisar solicitações de entrada com payload JSON. O **middleware express.json** é importante para analisar payloads JSON de entrada e disponibilizar esses dados no **req.body** ou processamento posterior dentro das rotas. Sem usar express.json, o Express não analisará automaticamente os dados JSON no corpo da solicitação.

Ao usar o **middleware express.json** , você pode manipular solicitações POST, PUT ou PATCH que enviam dados JSON do cliente para o servidor.

Sintaxe

```
ExpressJSon( [opções] )
```

- **Parâmetros:** O parâmetro options tem várias propriedades como inflate, limit, type, etc.
- **Valor de retorno:** Retorna um objeto .

O que são payloads?
No contexto de APIs e comunicação entre sistemas, um payload é o conteúdo real da requisição ou resposta que está sendo enviado. Ou seja, é a parte útil da mensagem, excluindo cabeçalhos, metadados ou outras informações adicionais.
Por exemplo, ao fazer uma requisição POST para uma API, o payload pode ser um objeto JSON com os dados que queremos enviar:

Por que ExpressJSon() é importante?

Quando você envia uma solicitação JSON para o servidor, os dados devem ser analisados antes que possam ser usados. O middleware ExpressJSon() é usado para analisar dados formatados JSON de entrada no corpo da solicitação.

Sem ele, os dados JSON no corpo da solicitação não estariam disponíveis em req.body como um objeto utilizável. Este middleware converte automaticamente os dados JSON brutos em objetos JavaScript que podem ser manipulados em seu código.

Sem usar ExpressJSON()

Saída:

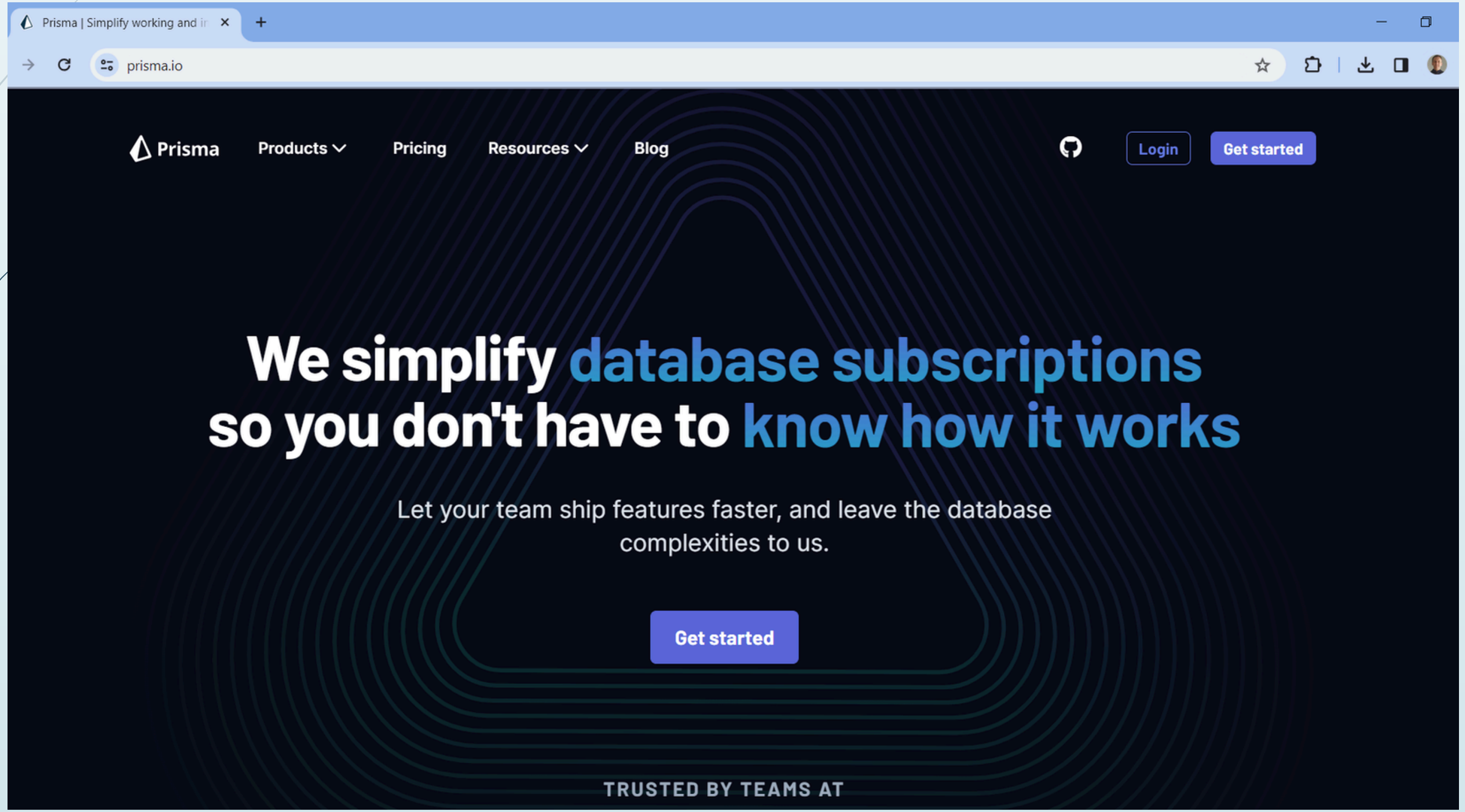
Quando uma solicitação POST é feita para http://localhost:3000/ com o cabeçalho Content-Type: application/json e o corpo {“name”:”GeeksforGeeks”}, a seguinte saída é exibida no console:

```
PS C:\Users\GFG0532\code> node index.js
Server listening on PORT 3000
TypeError: Cannot read properties of undefined (reading 'name')
    at C:\Users\GFG0532\code\index.js:8:26
    at Layer.handle [as handle_request] (C:\Users\GFG0532\code\node_modules\express\lib\router\layer.js:95:5)
    at next (C:\Users\GFG0532\code\node_modules\express\lib\router\route.js:149:13)
    at Route.dispatch (C:\Users\GFG0532\code\node_modules\express\lib\router\route.js:119:3)
    at Layer.handle [as handle_request] (C:\Users\GFG0532\code\node_modules\express\lib\router\layer.js:95:5)
    at C:\Users\GFG0532\code\node_modules\express\lib\router\index.js:284:15
    at Function.process_params (C:\Users\GFG0532\code\node_modules\express\lib\router\index.js:346:12)
    at next (C:\Users\GFG0532\code\node_modules\express\lib\router\index.js:280:10)
    at expressInit (C:\Users\GFG0532\code\node_modules\express\lib\middleware\init.js:40:5)
```

Sem utilizar express.json(), o req.body será indefinido, e tentar acessar req.body.name resultará em um TypeError.

https://www.geeksforgeeks.org/express-js-express-json-function/

APIs com Banco de Dados



Criar um novo projeto

```
C:\aulas24\api>md aula1

C:\aulas24\api>cd aula1

C:\aulas24\api\aula1>npm init -y
Wrote to C:\aulas24\api\aula1\package.json:
```

Criar um novo projeto: dependências

```
C:\aulas24\api\aula1>npm i typescript ts-node-dev --save-dev
```

npm i typescript ts-node-dev --save-dev

```
C:\aulas24\api\aula1>npm i @types/node @types/express --save-dev
```

npm i @types/node @types/express --save-dev

```
C:\aulas24\api\aula1>npm i express
```

npm i express

Inicializar Arquivo de Dependências do TS

```
C:\apis_251\manha\aula1>npx tsc --init
```

npx tsc --init

Gera um arquivo (tsconfig.json) com diversas configurações que controlam como o TypeScript compila o código .ts

Script: dev

C: > aulas24 > api > aula1 > {} package.json > ...

```
{
  "name": "aula1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "dev": "npx ts-node-dev --respawn index.ts"
  },
  "keywords": [],
  "author": "",
```

npx ts-node-dev --respawn index.ts

Exemplo Hello World

Este é essencialmente o aplicativo mais simples do Express que é possível criar. Ele é um aplicativo de arquivo único — ***não*** é o que você iria obter usando o Gerador Express, que cria a estrutura para um aplicativo completo com inúmeros arquivos JavaScript, modelos Jade, e subdiretórios para vários propósitos.

Primeiro crie um diretório chamado `myapp`, mude para ele e execute o `npm init`. Em seguida instale o `express` como uma dependência, de acordo com o guia de instalação.

No diretório `myapp`, crie um arquivo chamado `app.js` e inclua o seguinte código:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

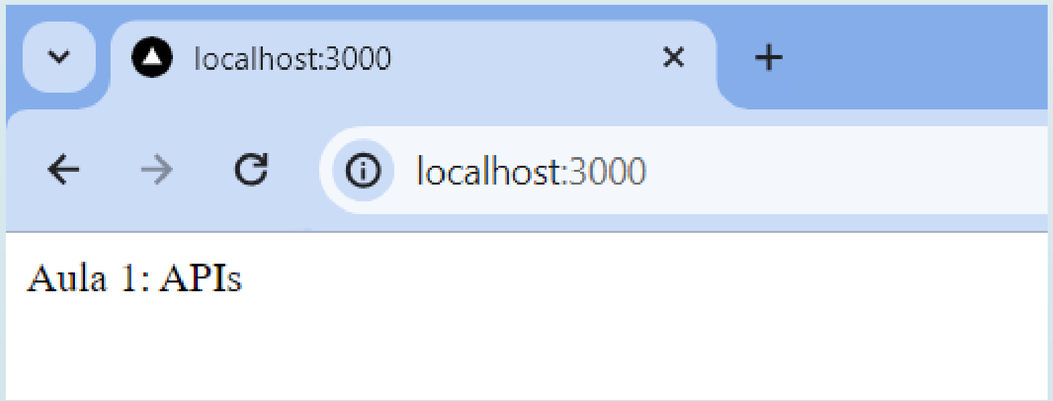
O aplicativo inicia um servidor e escuta a porta 3000 por conexões. O aplicativo responde com "Hello World!" à solicitações para a URL raiz (/) ou ***rota***. Para todos os outros caminhos, ele irá responder com um **404 Não Encontrado**.

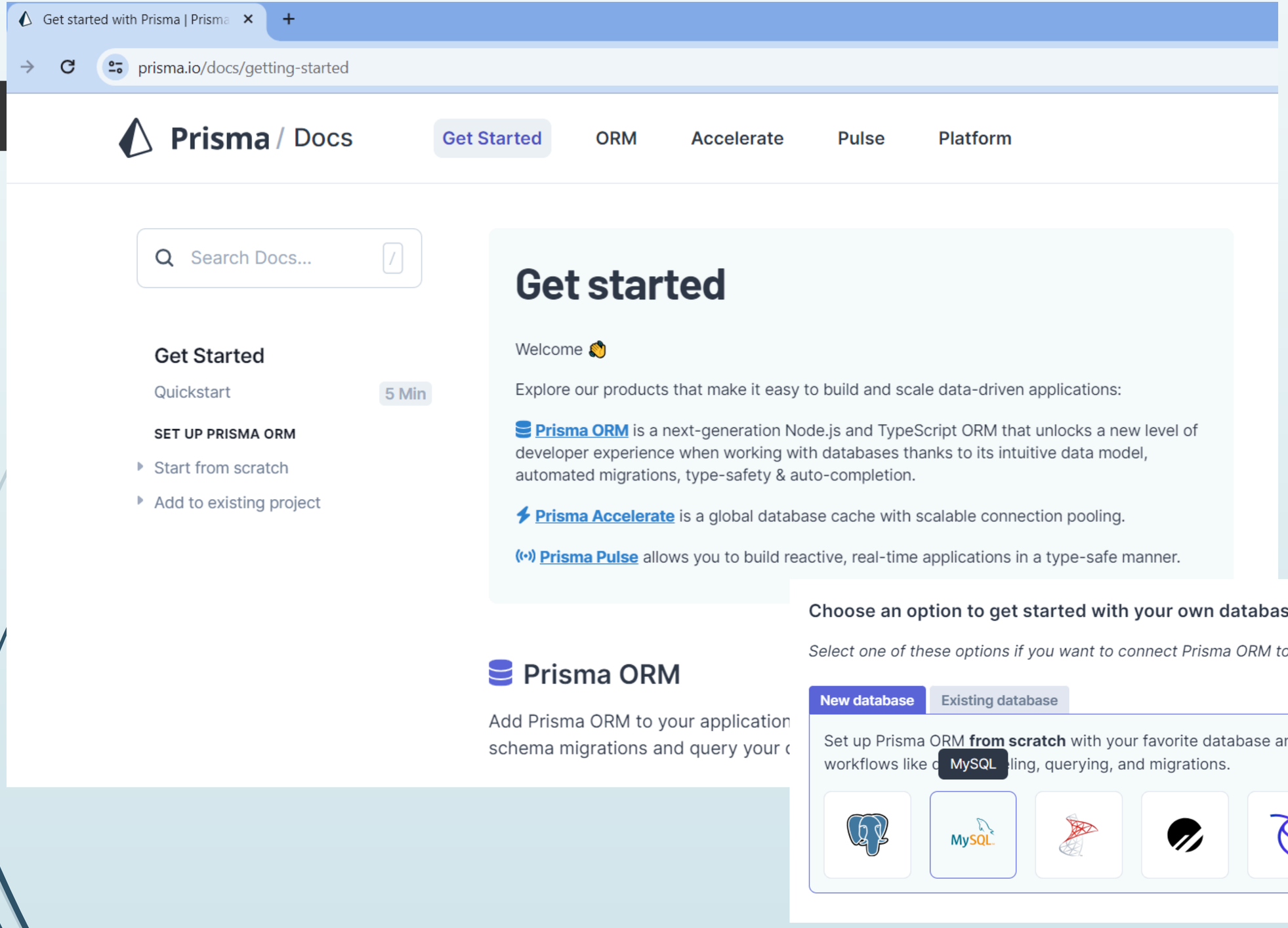
TS *index.ts* ×

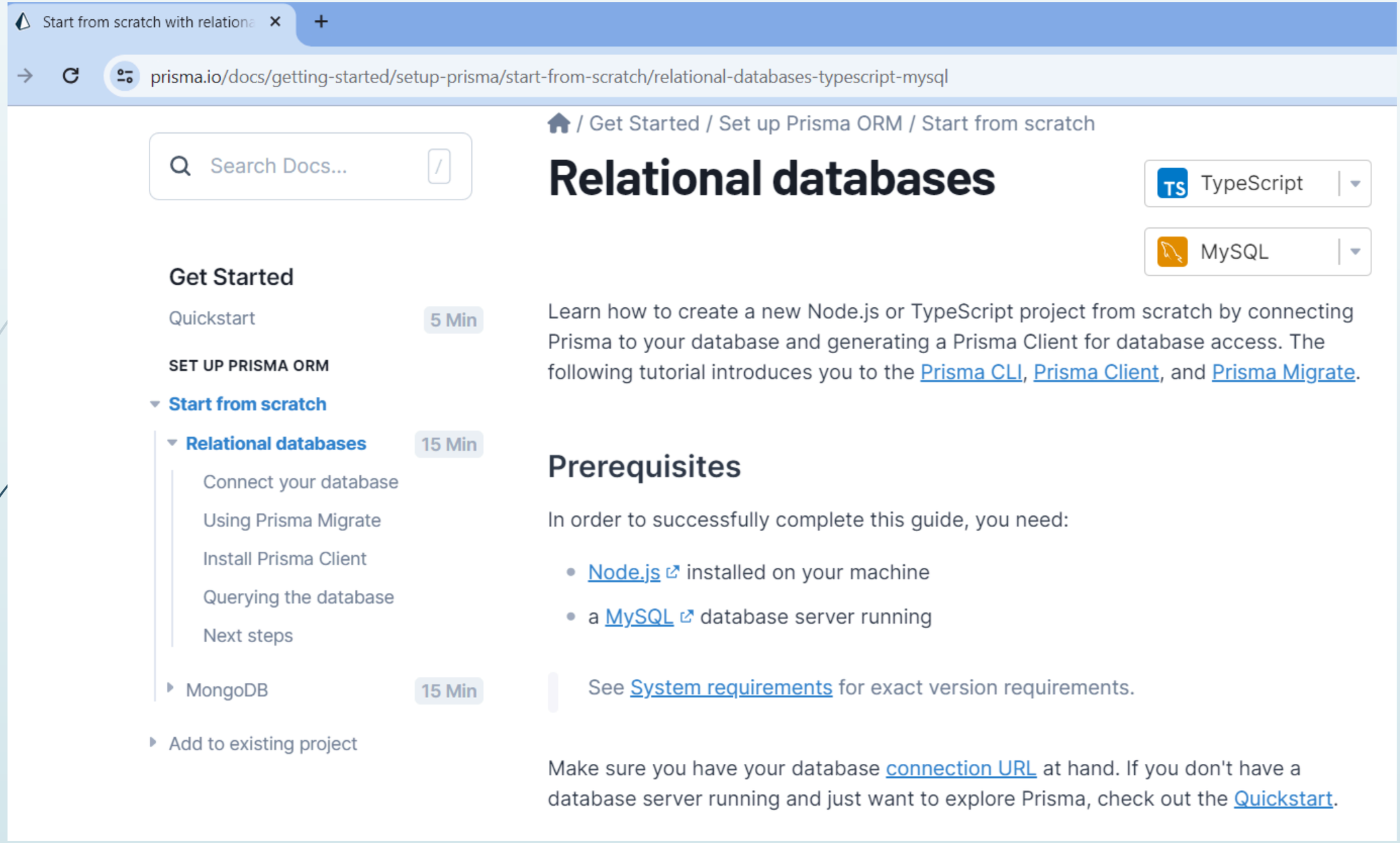
TS index.ts > ...

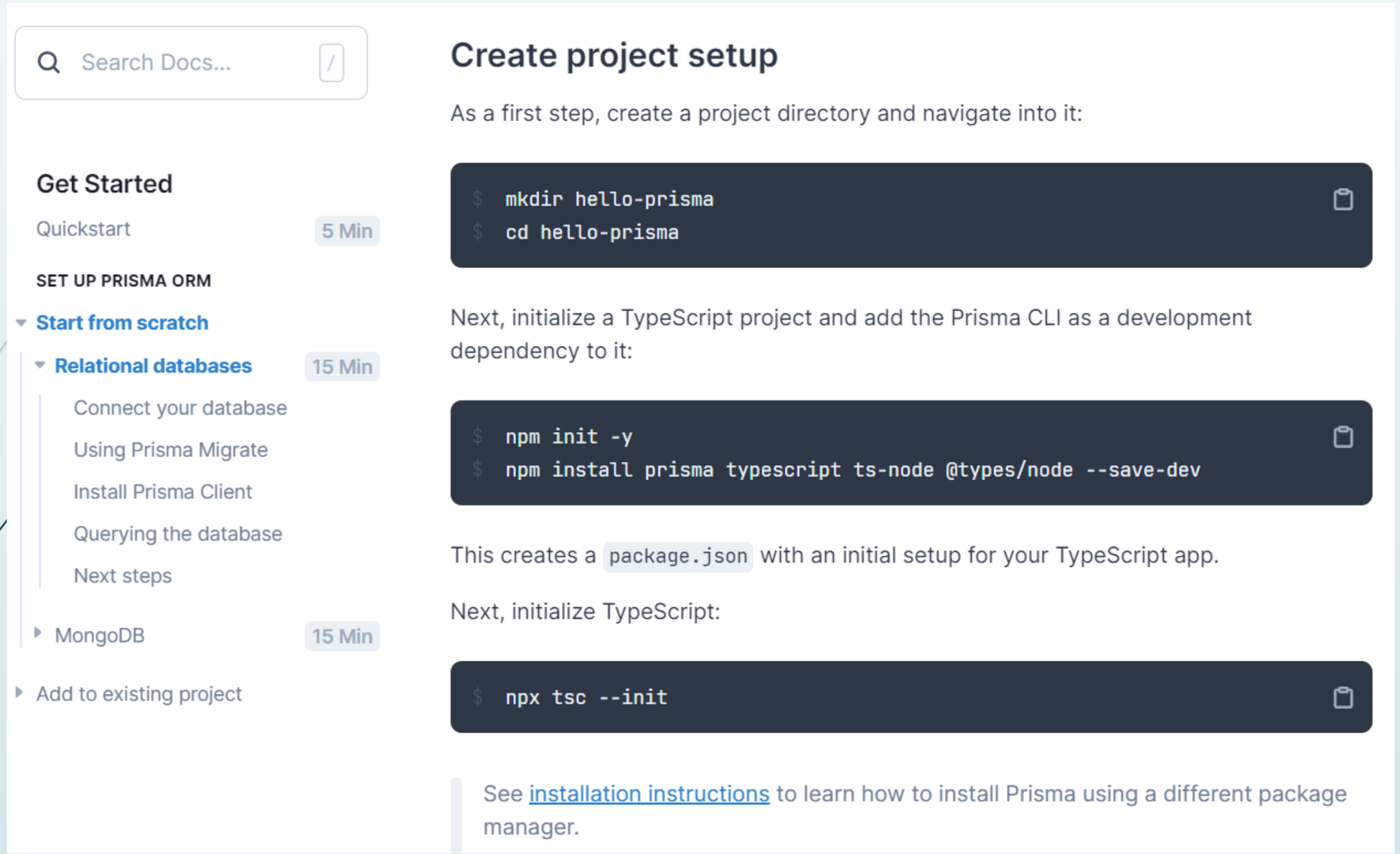
```ts
import express from 'express'
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Aula 1: APIs')
})

app.listen(port, () => {
  console.log(`Servidor rodando na porta ${port}`)
})

```

```
C:\aulas24\api\aula1>npm run dev

> aula1@1.0.0 dev
> nodemon --exec ts-node index.ts

[nodemon] 3.0.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node src/index.ts`
Servidor rodando em http://localhost:3000
```




Get started with Prisma | Prisma
prisma.io/docs/getting-started
Prisma / Docs
Get Started
ORM
Accelerate
Pulse
Platform
Search Docs...
/
Get Started
Quickstart
5 Min
SET UP PRISMA ORM
Start from scratch
Add to existing project
Get started
Welcome
Explore our products that make it easy to build and scale data-driven applications:
Prisma ORM is a next-generation Node.js and TypeScript ORM that unlocks a new level of developer experience when working with databases thanks to its intuitive data model, automated migrations, type-safety & auto-completion.
Prisma Accelerate is a global database cache with scalable connection pooling.
Prisma Pulse allows you to build reactive, real-time applications in a type-safe manner.
Prisma ORM
Add Prisma ORM to your application
schema migrations and query your
Choose an option to get started with your own database
Select one of these options if you want to connect Prisma ORM to your own database.
New database
Existing database
Set up Prisma ORM from scratch with your favorite database and learn basic workflows like
MySQL
ling, querying, and migrations.
MySQL


Start from scratch with relationa
prisma.io/docs/getting-started/setup-prisma/start-from-scratch/relational-databases-typescript-mysql
Search Docs...
/
Get Started
Quickstart
5 Min
SET UP PRISMA ORM
Start from scratch
Relational databases
15 Min
Connect your database
Using Prisma Migrate
Install Prisma Client
Querying the database
Next steps
MongoDB
15 Min
Add to existing project
/ Get Started / Set up Prisma ORM / Start from scratch
Relational databases
TypeScript
MySQL
Learn how to create a new Node.js or TypeScript project from scratch by connecting Prisma to your database and generating a Prisma Client for database access. The following tutorial introduces you to the Prisma CLI, Prisma Client, and Prisma Migrate.
Prerequisites
In order to successfully complete this guide, you need:
Node.js installed on your machine
a MySQL database server running
See System requirements for exact version requirements.
Make sure you have your database connection URL at hand. If you don't have a database server running and just want to explore Prisma, check out the Quickstart.




```
npm i prisma -D
```

Next, set up your Prisma ORM project by creating your Prisma Schema file with the following command:

```
$ npx prisma init
```

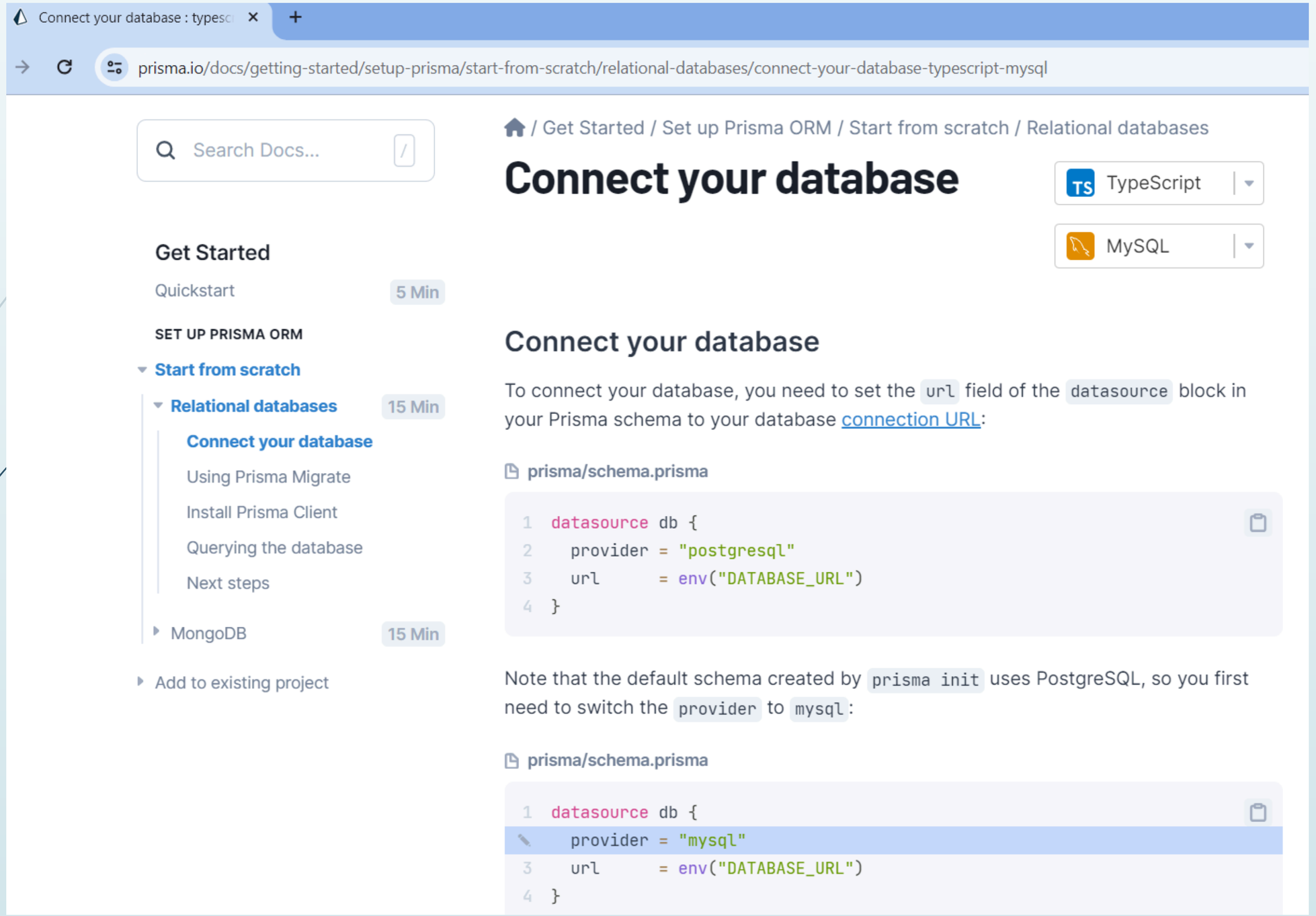
This command does two things:

- creates a new directory called `prisma` that contains a file called `schema.prisma`, which contains the Prisma schema with your database connection variable and schema models
- creates the `.env` file in the root directory of the project, which is used for defining environment variables (such as your database connection)

npx prisma init

Conecte-se ao MySQL e crie o banco de dados do projeto

```
C:\MariaDB107\bin>mysql -u root -p
Enter password: ****
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 210
Server version: 10.7.3-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> create database primevideo_manha;
Query OK, 1 row affected (0.014 sec)

MariaDB [(none)]>
```


Connect your database : typesc
prisma.io/docs/getting-started/setup-prisma/start-from-scratch/relational-databases/connect-your-database-typescript-mysql
Search Docs...
/
Get Started
Quickstart
5 Min
SET UP PRISMA ORM
Start from scratch
Relational databases
15 Min
Connect your database
Using Prisma Migrate
Install Prisma Client
Querying the database
Next steps
MongoDB
15 Min
Add to existing project
/ Get Started / Set up Prisma ORM / Start from scratch / Relational databases
Connect your database
TypeScript
MySQL
Connect your database
To connect your database, you need to set the url field of the datasource block in your Prisma schema to your database connection URL:
prisma/schema.prisma
1 datasource db {
2 provider = "postgresql"
3 url = env("DATABASE_URL")
4 }
Note that the default schema created by prisma init uses PostgreSQL, so you first need to switch the provider to mysql:
prisma/schema.prisma
1 datasource db {
provider = "mysql"
3 url = env("DATABASE_URL")
4 }

In this case, the `url` is set via an environment variable which is defined in `.env`:

.env

```
DATABASE_URL="mysql://johndoe:randompassword@localhost:3306/mydb"
```

> We recommend adding `.env` to your `.gitignore` file to prevent committing your environment variables.

You now need to adjust the connection URL to point to your own database.

The format of the connection URL for your database typically depends on the database you use. For MySQL, it looks as follows (the parts spelled all-uppercased are *placeholders* for your specific connection details):

```
mysql://USER:PASSWORD@HOST:PORT/DATABASE
```

Here's a short explanation of each component:

- `USER`: The name of your database user
- `PASSWORD`: The password for your database user
- `PORT`: The port where your database server is running (typically `3306` for MySQL)
- `DATABASE`: The name of the database

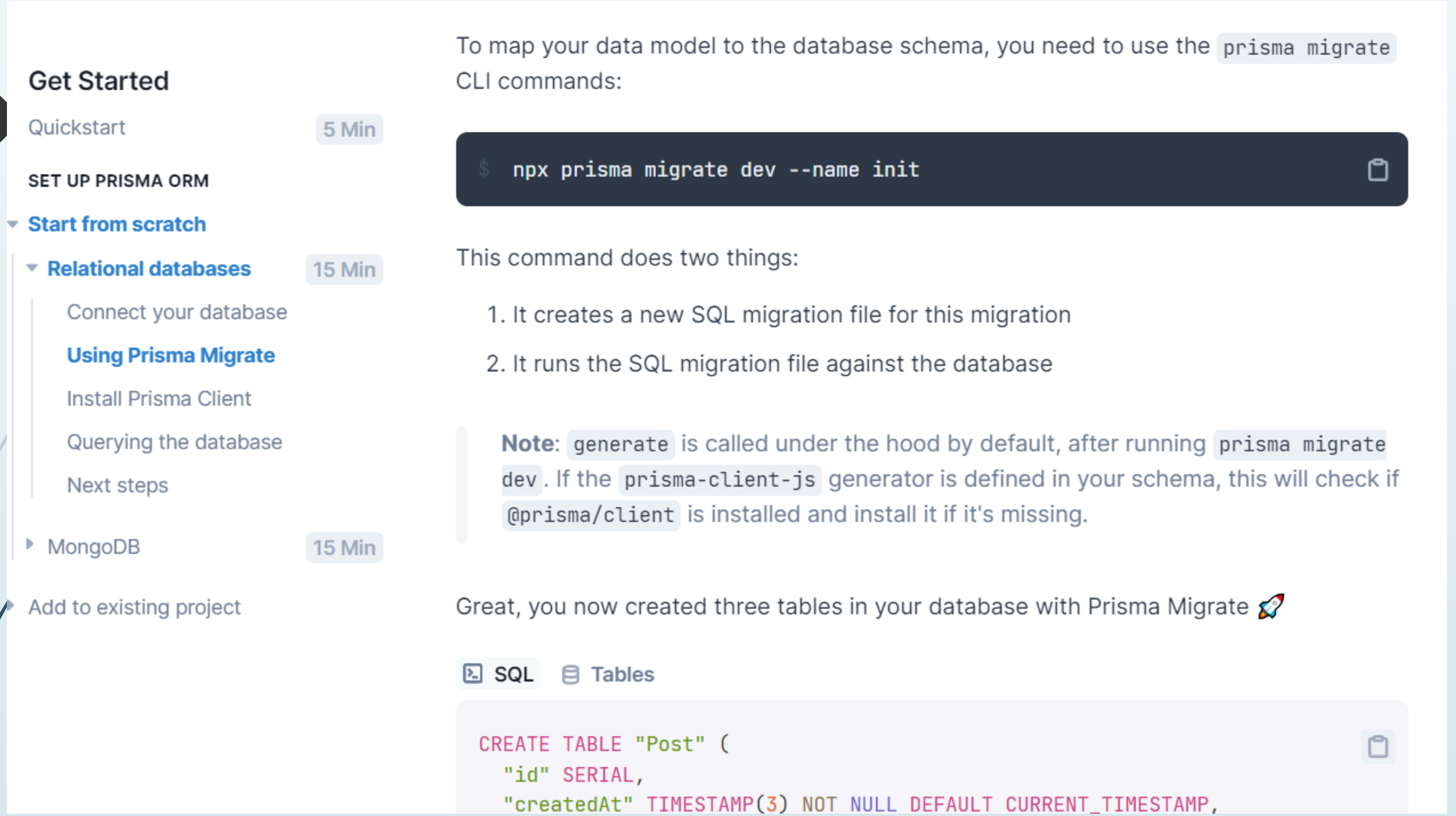
Using Prisma Migrate

Creating the database schema

In this guide, you'll use Prisma Migrate to create the tables in your database. Add the following Prisma data model to your Prisma schema in `prisma/schema.prisma`:

prisma/schema.prisma

```
model Post {
  id        Int      @id @default(autoincrement())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  title     String   @db.VarChar(255)
  content   String?
  published Boolean  @default(false)
  author    User     @relation(fields: [authorId], references: [id])
  authorId  Int
}

model Profile {
  id     Int     @id @default(autoincrement())
  bio    String?
  user   User    @relation(fields: [userId], references: [id])
```



npx prisma migrate dev --name nome_migration

Search Docs... /

Get Started

Quickstart 5 Min

SET UP PRISMA ORM

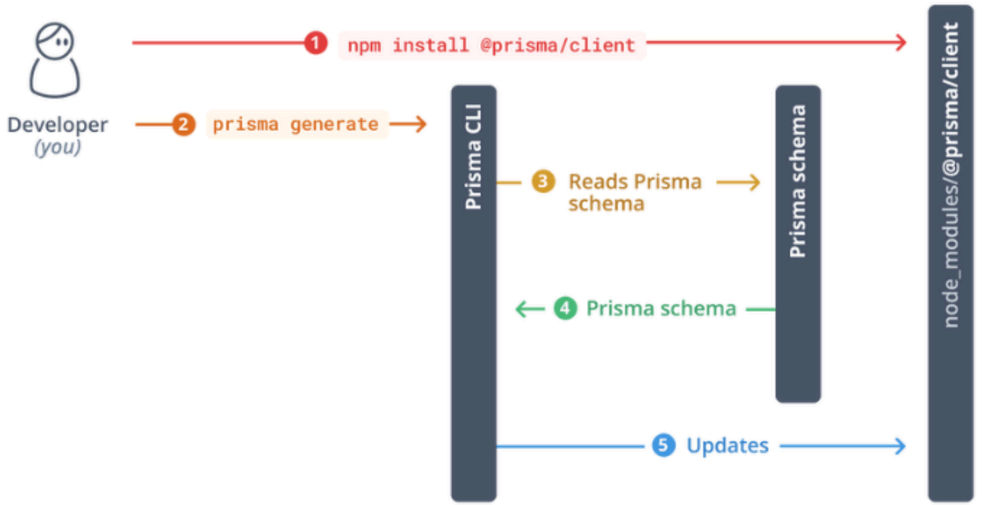
- Start from scratch
  - Relational databases 15 Min
    - Connect your database
    - Using Prisma Migrate
    - Install Prisma Client
    - Querying the database
    - Next steps
  - MongoDB 15 Min
- Add to existing project

Install and generate Prisma Client

To get started with Prisma Client, you need to install the `@prisma/client` package:

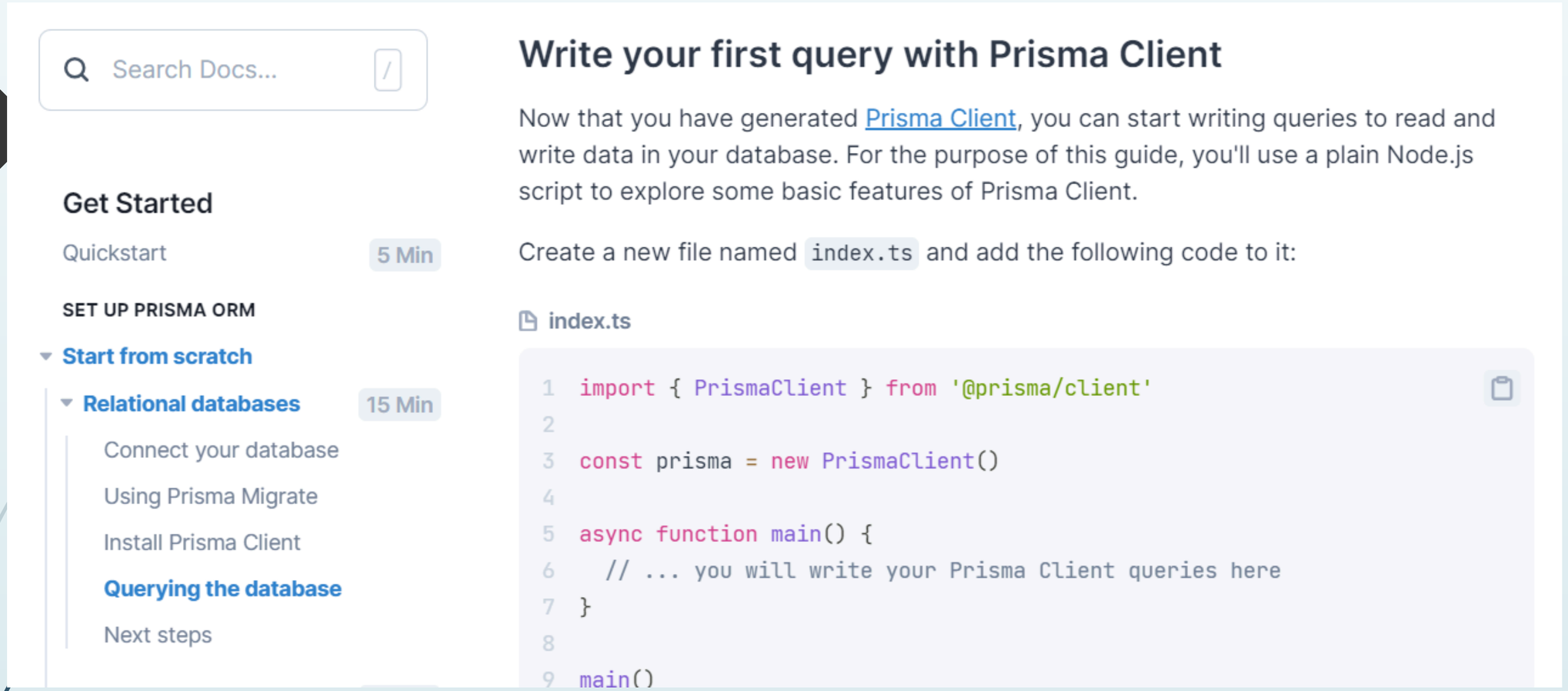
```
$ npm install @prisma/client
```

The install command invokes `prisma generate` for you which reads your Prisma schema and generates a version of Prisma Client that is *tailored* to your models.

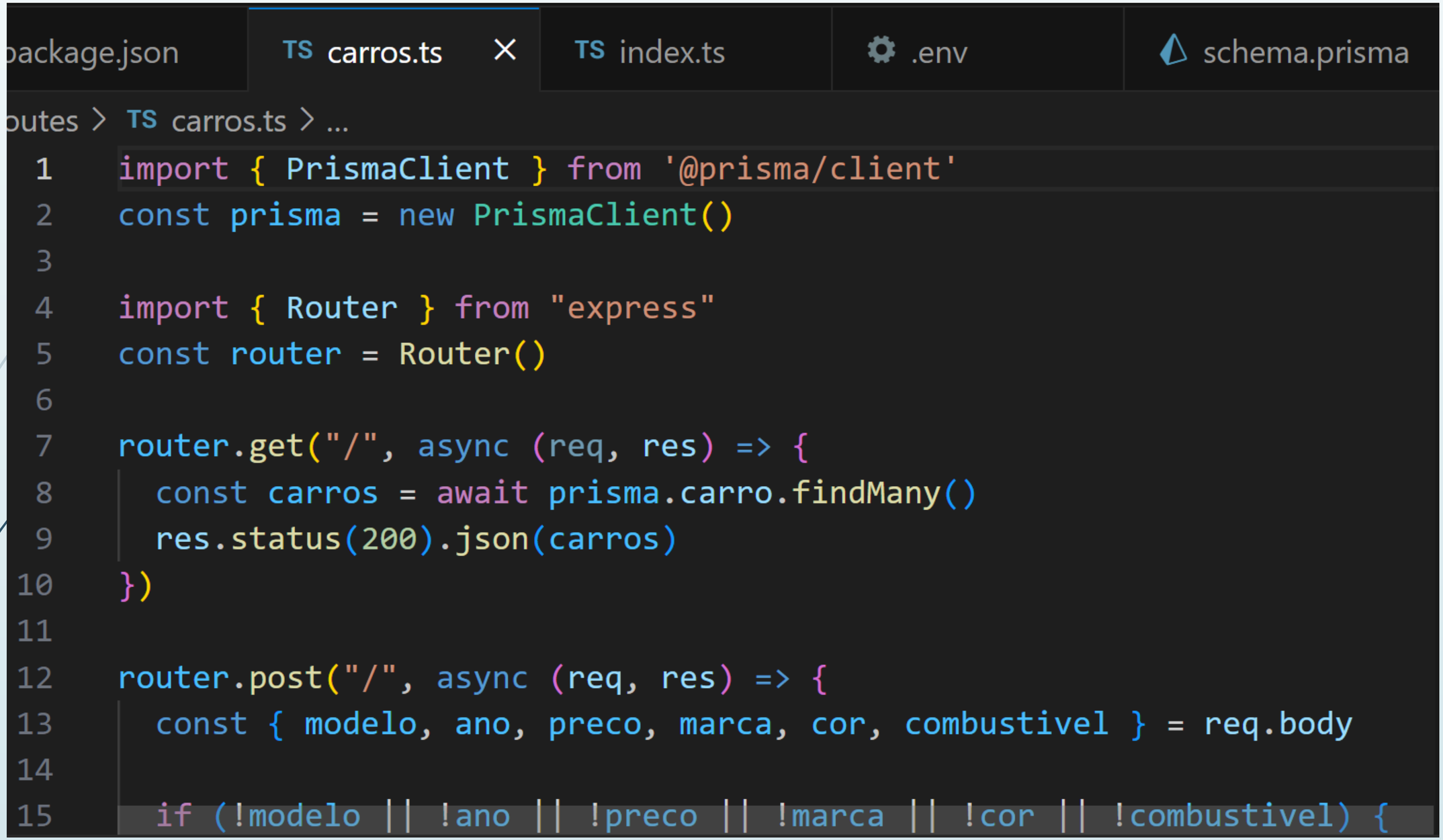


Whenever you update your Prisma schema, you will have to update your database schema using either `prisma migrate dev` or `prisma db push`. This will keep your database schema in sync with your Prisma schema. The commands will also regenerate Prisma Client.

```
npm i @prisma/client
```




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

const prisma = new PrismaClient()
```



* Cria-se um arquivo para cada conjunto de rotas (em uma pasta routes)

Search Docs... /

Get Started

Quickstart 5 Min

SET UP PRISMA ORM

Start from scratch

Relational databases 15 Min

Connect your database

Using Prisma Migrate

Install Prisma Client

Querying the database

Next steps

MongoDB 15 Min

Add to existing project

Here's a quick overview of the different parts of the code snippet:

1. Import the `PrismaClient` constructor from the `@prisma/client` node module
2. Instantiate `PrismaClient`
3. Define an `async` function named `main` to send queries to the database
4. Call the `main` function
5. Close the database connections when the script terminates

Inside the `main` function, add the following query to read all `User` records from the database and print the result:

index.ts

```
async function main() {
  // ... you will write your Prisma Client queries here
+   const allUsers = await prisma.user.findMany()
+   console.log(allUsers)
}
```

Now run the code with this command:

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

This should print an empty array because there are no `User` records in the database yet:

Search Docs... /

Get Started

Quickstart 5 Min

SET UP PRISMA ORM

Start from scratch

Relational databases 15 Min

Connect your database

Using Prisma Migrate

Install Prisma Client

Querying the database

Next steps

MongoDB 15 Min

Add to existing project

Write data into the database

The `findMany` query you used in the previous section only *reads* data from the database (although it was still empty). In this section, you'll learn how to write a query to *write* new records into the `Post` and `User` tables.

Adjust the `main` function to send a `create` query to the database:

index.ts

```
1  async function main() {
+    await prisma.user.create({
+      data: {
+        name: 'Alice',
+        email: 'alice@prisma.io',
+        posts: {
+          create: { title: 'Hello World' },
+        },
+        profile: {
+          create: { bio: 'I like turtles' },
+        },
+      },
+    })
+
```

Get Started

Quickstart 5 Min

SET UP PRISMA ORM

Start from scratch

Relational databases 15 Min

Connect your database

Using Prisma Migrate

Install Prisma Client

Querying the database

Next steps

MongoDB 15 Min

Before moving on to the next section, you'll "publish" the `Post` record you just created using an `update` query. Adjust the `main` function as follows:

index.ts

```ts
async function main() {
  const post = await prisma.post.update({
    where: { id: 1 },
    data: { published: true },
  })
  console.log(post)
}
```

Now run the code using the same command as before:

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

Building REST APIs with Prisma

prisma.io/docs/orm/overview/prisma-in-your-stack/rest

Prisma / Docs

Get Started | ORM | Accelerate | Pulse | Platform

Search Docs... /

ORM

OVERVIEW

- Introduction
- Prisma in your stack
  - REST
  - GraphQL
  - Fullstack
  - Is Prisma an ORM?
- Databases

PRISMA SCHEMA

- Overview
- Data model
- Introspection
- PostgreSQL extensions Preview

PRISMA CLIENT

/ ORM / Overview / Prisma in your stack

REST

When building REST APIs, Prisma Client can be used inside your *route controllers* to send databases queries.

Supported libraries

As Prisma Client is "only" responsible for sending queries to your database, it can be combined with any HTTP server library or web framework of your choice.

Here's a non-exhaustive list of libraries and frameworks you can use with Prisma:

- Express
- koa
- hapi
- Fastify

Search Docs... /

ORM

OVERVIEW

- Introduction
- Prisma in your stack
  - REST
  - GraphQL
  - Fullstack
  - Is Prisma an ORM?
- Databases

PRISMA SCHEMA

- Overview
- Data model
- Introspection
- PostgreSQL extensions Preview

PRISMA CLIENT

- Setup & configuration
- Queries
- Fields & types
- Extensions

REST API server example

Assume you have a Prisma schema that looks similar to this:

```
datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}

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

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

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

You can now implement route controller (e.g. using Express) that use the generated Prisma Client API to perform a database operation when an incoming HTTP request

`GET`

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

Note that the `feed` endpoint in this case returns a nested JSON response of `Post` objects that *include* an `author` object. Here's a sample response:

```
[
  {
    "id": "21",
    "title": "Hello World",
    "content": "null",
```

POST

```
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)
})
```

PUT

```
app.put('/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)
})
```

DELETE

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