Getting Started > Adapters > Prisma

Prisma Adapter

Resources

- Prisma documentation

Setup

Installation

npm pnpm yarn bun

```
npm install @prisma/client @auth/prisma-adapter
npm install prisma --save-dev
```

Environment Variables

Prisma needs to set up the environment variable to establish a connection with your database and retrieve data. Prisma requires the `DATABASE_URL` environment variable to create the connection. For more information, read the docs.

```
DATABASE_URL=postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=SCHEMA
```

Configuration

To improve performance using `Prisma ORM`, we can set up the Prisma instance to ensure only one instance is created throughout the project and then import it from any file as needed. This approach avoids recreating instances of PrismaClient every time it is used. Finally, we can import the Prisma instance from the `auth.ts` file configuration.

prisma.ts

```ts
import { PrismaClient } from "@prisma/client"

const globalForPrisma = globalThis as unknown as { prisma: PrismaClient }

export const prisma = globalForPrisma.prisma || new PrismaClient()

if (process.env.NODE_ENV !== "production") globalForPrisma.prisma = prisma
```

⚠️ We recommend using version `@prisma/client@5.12.0` or above if using middleware or any other edge runtime(s). See edge compatibility below for more information.

Next.js Qwik SvelteKit Express

./auth.ts

```ts
import NextAuth from "next-auth"
import { PrismaAdapter } from "@auth/prisma-adapter"
import { prisma } from "@/prisma"

export const { handlers, auth, signIn, signOut } = NextAuth({
  adapter: PrismaAdapter(prisma),
  providers: [],
})
```

Edge Compatibility

Prisma has shipped edge runtime support for their client in version `5.12.0`. You can read more about it on their edge documentation. This requires specific database drivers and therefore is only compatible with certain database types / hosting providers. Check their list of supported drivers before getting started. You can check out an example Auth.js application with `next-auth` and Prisma on the edge here.

For more about edge compatibility in general, check out our edge compatibility guide.

The original database edge-runtime workaround, to split your `auth.ts` configuration into two, will be kept below.

Old Edge Workaround

At the moment, Prisma is still working on being fully compatible with edge runtimes like Vercel's. See the issue being tracked here, and Prisma's announcement about early edge support in the `5.9.1` changelog. There are two options to deal with this issue:

- Use the Prisma's Accelerate feature
- Follow our Edge Compatibility page as the workaround. This uses the `jwt` session strategy and separates the `auth.ts` configuration into two files.

Using Prisma with the `jwt` session strategy and `@prisma/client@5.9.1` or above doesn't require any additional modifications, other than ensuring you don't do any database queries in your middleware.

Since `@prisma/client@5.9.1`, Prisma no longer throws about being incompatible with the edge runtime at instantiation, but at query time. Therefore, it is possible to import it in files being used in your middleware as long as you do not execute any queries in your middleware.

Schema

You need to use at least Prisma `2.26.0`. Create a schema file at `prisma/schema.prisma` with the following models.

PostgreSQL

prisma/schema-postgres.prisma

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

```
}

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

model User {
  id            String          @id @default(cuid())
  name          String?
  email         String          @unique
  emailVerified DateTime?
  image         String?
  accounts      Account[]
  sessions      Session[]
  // Optional for WebAuthn support
  Authenticator Authenticator[]

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

model Account {
  userId            String
  type              String
  provider          String
  providerAccountId String
  refresh_token     String?
  access_token      String?
  expires_at        Int?
  token_type        String?
  scope             String?
  id_token          String?
  session_state     String?

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  user User @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@id([provider, providerAccountId])
}

model Session {
  sessionToken String   @unique
  userId       String
  expires      DateTime
  user         User     @relation(fields: [userId], references: [id], onDelete

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

model VerificationToken {
  identifier String
  token      String
  expires    DateTime

  @@id([identifier, token])
}

// Optional for WebAuthn support
model Authenticator {
  credentialID         String  @unique
  userId               String
  providerAccountId    String
  credentialPublicKey  String
  counter              Int
  credentialDeviceType String
  credentialBackedUp   Boolean
  transports           String?

  user User @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@id([userId, credentialID])
}
```

> MySQL

> SQLite

> MongoDB

Apply Schema

This will create an SQL migration file and execute it:

npm pnpm yarn bun

```
npm exec prisma migrate dev
```

Note that you will need to specify your database connection string in the environment variable `DATABASE_URL`. You can do this by setting it in a `.env` file at the root of your project.

Generate Prisma Client

`prisma migrate dev` will also generate the Prisma client, but if you need to generate it again manually you can run the following command.

npm pnpm yarn bun

```
npm exec prisma generate
```

Development Workflow

When you're working on your application and making changes to your database schema, you'll need to run the migrate command again every time you make changes to the schema in order for Prisma to (1) generate a migration file and apply it to the underlying database and (2) regenerate the Prisma client in your project with the latest types and model methods.

npm pnpm yarn bun

```
npm exec prisma migrate dev
```

Naming Conventions

If mixed `snake_case` and `camelCase` column names is an issue for you and/or your underlying database system, we recommend using Prisma's `@map()` feature to change the field names. This won't affect Auth.js, but will allow you to customize the column names to whichever naming convention you prefer.

For example, moving to `snake_case` and plural table names.

schema.prisma

```
model Account {
  id                 String  @id @default(cuid())
  userId             String  @map("user_id")
  type               String
  provider           String
  providerAccountId  String  @map("provider_account_id")
  refresh_token      String? @db.Text
  access_token       String? @db.Text
  expires_at         Int?
  token_type         String?
  scope              String?
  id_token           String? @db.Text
  session_state      String?

  user User @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@unique([provider, providerAccountId])
  @@map("accounts")
}

model Session {
  id           String   @id @default(cuid())
  sessionToken String   @unique @map("session_token")
  userId       String   @map("user_id")
  expires      DateTime
  user         User     @relation(fields: [userId], references: [id], onDelete:

  @@map("sessions")
}

model User {
  id            String    @id @default(cuid())
  name          String?
  email         String?   @unique
  emailVerified DateTime? @map("email_verified")
  image         String?
  accounts      Account[]
  sessions      Session[]

  @@map("users")
}
```

```
model VerificationToken {
  identifier String
  token      String
  expires    DateTime

  @@unique([identifier, token])
  @@map("verification_tokens")
}
```

Last updated on January 9, 2025

About Auth.js

Introduction

Security

Discord Community

Download

GitHub

NPM

Acknowledgements

Contributors

Sponsors