Integrating Prisma with Express in a Clean and Scalable Manner

Camilo Salazar Follow 6 min read · Dec 24, 2023

10 2

> *Learn how to seamlessly integrate Prisma with Express for a clean and scalable Node.js backend. Explore efficient data modeling, type-safe operations, and robust database interactions.*

Photo by benjamin lehman on Unsplash

Welcome to a tutorial where we'll master the art of crafting an elegant and efficient Express application seamlessly integrated with Prisma your sturdy, type-safe database toolkit! Let's embark on a journey to not only ensure smooth communication between our Express application and the database but also to build a robust, structured, and scalable solution using the principles of Clean Architecture.

> *The complete code for this Node.js Express backend can be found in the following GitHub repository link, where you can explore all the files and code used to build it.*

GitHub - csalazar94/prisma-express-tutorial

Contribute to csalazar94/prisma-express-tutorial development by creating an account on GitHub.

github.com

Setting up our project

Install Node.js and npm

If you haven't already, follow the steps outlined in my previous tutorial, Building a Flexible and Scalable Node.js Backend with Express: A Step-by-Step Tutorial.

Building a Flexible and Scalable Node.js Backend with Express: A Step-by-Step Tutorial

This tutorial walks through the project structure, implementing the controller, service, router and using the...

medium.com

Before we dive into the integration, make sure you have Prisma installed on your project.

```
npm install prisma - save-dev
```

Initialize Prisma

Run the following command to initialize Prisma in your project:

```
npx prisma init
```

This command will create a `prisma` directory with a `schema.prisma` file.

Configure the Database

Let's proceed with configuring the database using SQLite and setting up the `.env` file with the `DATABASE_URL`.

Open the `prisma/schema.prisma` file and configure the SQLite datasource:

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

This configuration specifies SQLite as the provider and uses the

Open the `.env` file and configure the SQLite datasource:

```
DATABASE_URL="file:./dev.db"
```

Define Your Data Model

Let's define the data model for the `User` entity with `id`, `email`, `password`, and `age` fields in the `schema.prisma` file.

```
model User {
  id       Int      @id @default(autoincrement())
  email    String   @unique
  password String
  age      Int?
}
```

Explanation of the model:

- `User` : This is the name of the entity/model.
- `id` : An integer field that serves as the primary key and automatically increments.
- `email` : A unique string field for the user's email address.
- `password` : A string field to store the user's password.
- `age` : An optional integer field representing the user's age.

The `@id` and `@default(autoincrement())` annotations on the `id` field indicate that it is the primary key and will auto-increment with each new record.

The `@unique` annotation on the `email` field ensures that each email address in the database is unique.

Run Prisma Migrate

To create a migration and execute it using Prisma, follow these steps:

1.- Create a Migration: Run the following command to create a new migration. Prisma will analyze the changes in your schema (`schema.prisma`) and generate the necessary SQL migration files.

```
npx prisma migrate dev --name init
```

After running this command, Prisma will create a migration file named "init" if changes to the schema are detected and automatically apply those changes to the database. Additionally, the Prisma Client will be regenerated to reflect any modifications to the schema.

```
-- CreateTable
CREATE TABLE "User" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "email" TEXT NOT NULL,
    "password" TEXT NOT NULL,
    "age" INTEGER
);

-- CreateIndex
CREATE UNIQUE INDEX "User_email_key" ON "User"("email");
```

Using `migrate dev` is a practical choice during development, but in production environments, you might prefer a more controlled approach with `deploy` command.

Prisma Implementation as a Service

In this section, we'll seamlessly integrate Prisma into the existing structure of our Node.js backend, as outlined in the blog post Building a Flexible and Scalable Node.js Backend with Express: A Step-by-Step Tutorial. This

integration will empower our application with a robust and type-safe database layer.

Updating the Constructor

To begin, let's update our constructor by adding the Prisma Client:

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

class UserService {
  constructor() {
    this.prisma = new PrismaClient();
  }

  // ... Existing constructor code ...
}
```

This addition ensures that our `UserService` is now equipped with the Prisma Client, granting seamless access to interact with the database. It also facilitates ease of mocking our Prisma client for testing purposes—refer to my tutorial: From Unit to End-to-End Testing with Jest and Supertest.

Updating the getUsers Method

Moving forward, we enhance our `getUsers` method, leveraging the Prisma Client for streamlined user retrieval:

```
// ...

  getUsers = () => this.prisma.user.findMany({
    select: {
      id: true,
      email: true,
      age: true,
    },
  });
```

```
// ...
```

By leveraging Prisma, we've simplified our user retrieval process, making it more efficient and type-safe.

Updating the addUser Method

Now, let's transform our `addUser` method into an asynchronous function, taking advantage of Prisma's capabilities:

```
// ...

  addUser = async (user) => {
    const { password, ...createdUser } = await this.prisma.user.create({
      data: user,
    });
    return createdUser;
  };

// ..
```

This modification ensures that adding a new user is an asynchronous operation, allowing for better handling of database interactions.

Updating the getUser Method

Lastly, let's update our `getUser` method to use the Prisma Client for fetching a unique user based on their ID:

```
// ...

  getUser = (id) => this.prisma.user.findUnique({
    where: { id },
    select: {
      id: true,
      email: true,
      age: true,
    },
  });

// ...
```

By incorporating Prisma, we enhance the reliability and type safety of our user retrieval process.

With these updates, our Express backend now seamlessly integrates Prisma, providing a robust and efficient mechanism for interacting with the underlying database.

Updating our Controller

With the seamless integration of Prisma into our backend, our controller now undergoes a refinement to align with the capabilities of the Prisma Client. We no longer need the `user.entities.js` file, as Prisma handles our data modeling requirements.

Refining Controller Functions

Let's enhance our `UserController` by removing the instantiation of the `User` class, as it's no longer needed. Additionally, we'll convert our functions to asynchronous to fully leverage Prisma's asynchronous nature. We'll also incorporate the use of our `UserService`:

```
class UserController {
  constructor(userService) {
    this.userService = userService;
  }

  createUser = async (req, res) => {
    try {
      const user = await this.userService.addUser(req.body);
      return res.status(201).send(user);
    } catch (error) {
      return res.status(500).json({ error: 'Internal Server Error' });
    }
  };

  getUsers = async (_, res) => {
```

```
    try {
      const users = await this.userService.getUsers();
      return res.status(200).send(users);
    } catch (error) {
      return res.status(500).json({ error: 'Internal Server Error' });
    }
  };

  getUser = async (req, res) => {
    try {
      const { id } = req.params;
      const user = await this.userService.getUser(Number(id));
      return res.status(200).send(user);
    } catch (error) {
      return res.status(500).json({ error: 'Internal Server Error' });
    }
  };
}

export default UserController;
```

These adjustments ensure that our controller seamlessly integrates with the enhanced data handling capabilities provided by Prisma. Error handling has also been incorporated to gracefully manage potential issues during user creation and retrieval.

Conclusion

In this tutorial, we've successfully integrated Prisma into our Node.js backend, augmenting our Express application with a robust and type-safe database layer. By following the step-by-step implementation, we've harnessed the power of Prisma to simplify database interactions, enhance reliability, and ensure type safety throughout our application.

Key Takeaways:

1. Efficient Data Modeling: Prisma empowers us to define and model our data effortlessly, providing a clear and concise representation of our database schema.
2. Seamless Database Interactions: With the Prisma Client, our backend seamlessly interacts with the database, resulting in concise and readable code for database operations.
3. Type-Safe Operations: Prisma's type-safe queries and auto-generated client code eliminate runtime errors, ensuring that our application is robust and reliable.

What's Next?

As you continue to refine and expand your Node.js backend, consider exploring advanced features of Prisma, such as migrations, transactions, and relationships. Additionally, delve into performance optimizations and explore Prisma's compatibility with various databases.

For further insights into testing strategies, don't forget to check out my tutorial on From Unit to End-to-End Testing with Jest and Supertest, where we explore comprehensive testing approaches to ensure the resilience of your backend.

Keep Building, Keep Scaling

From Unit to End-to-End Testing with Jest and Supertest

In this tutorial we covered how to write unit tests using Jest, with a step-by-step guide to creating a test case for a...

medium.com

Keep Building, Keep Scaling

If you have questions or want to share your experiences, feel free to connect in the comments section below. Happy coding!

Prisma Expressjs Nodejs Database Scalability

Written by Camilo Salazar

53 followers · 6 following

Driven to share powerful ideas and experiences that ignite inspiration, deepen knowledge, and empower others

Follow

Responses (2)

Write a response

What are your thoughts?

Muhammed Shibili N
Jan 16, 2025

hloo sir could you please share about clean architecture

Reply

Yomicepa
Feb 12, 2024

Great article! It was really helpful. Would be amazing if you could add a new article to build authentication and handle JWT on top of this repo

2 replies Reply

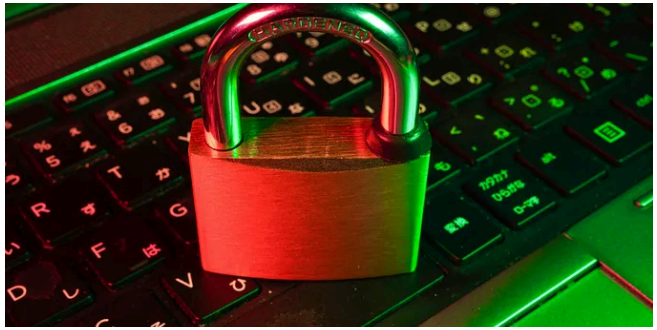
More from Camilo Salazar

Camilo Salazar

My Experience Solving Database Connection Issues with Sequelize...

This blog post details my solution to database connection issues in a Node.js app...

May 25, 2024 3 1

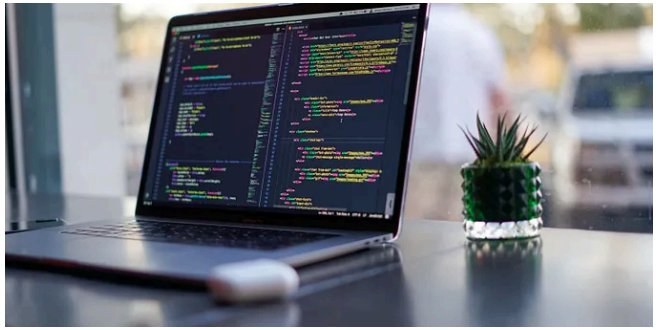


Camilo Salazar

Adding Authentication to Your Prisma-Integrated Express...

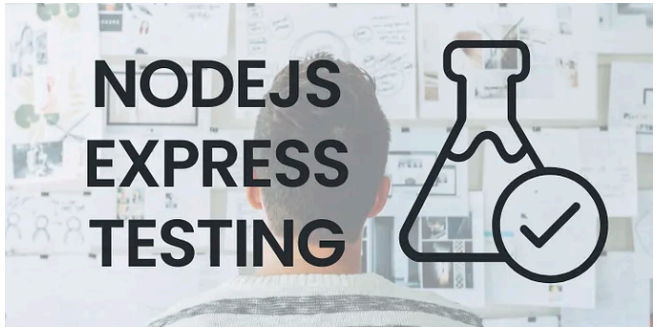
Secure your Express app with rock-solid authentication! Learn to use password...

Apr 16, 2024 6



Camilo Salazar

Building a Flexible and Scalable Node.js Backend with Express: A...



Camilo Salazar

From Unit to End-to-End Testing with Jest and Supertest

This tutorial walks through the project structure, implementing the controller,...

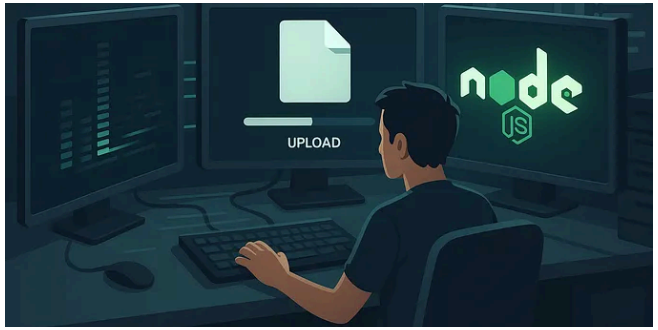
Feb 27, 2023 147 8

In this tutorial we covered how to write unit tests using Jest, with a step-by-step guide t...

Mar 16, 2023 122

See all from Camilo Salazar

Recommended from Medium



In Stackademic by Dipak Ahirav

Handling Large File Uploads in Node.js Without Crashing Your...

In JavaScript in Plain English by Vusal Huseynov

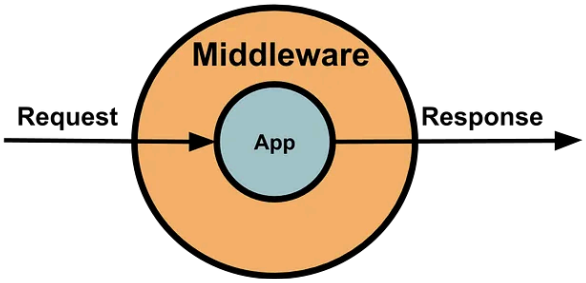
Dependency Injection in Express.js

Learn how to process massive file uploads in Node.js efficiently with streams, buffers, and...

Sep 8, 2025 10

When Express apps grow, wiring dependencies by hand (newing classes or...

Oct 12, 2025 12

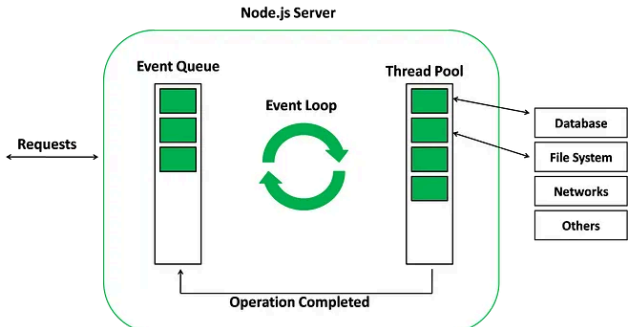


Priyanshu Rajput

🧩 Mastering Express.js Middleware: The Hidden Power...

Every request that hits your Express server takes a journey—a silent tour through your...

Oct 13, 2025 52 1

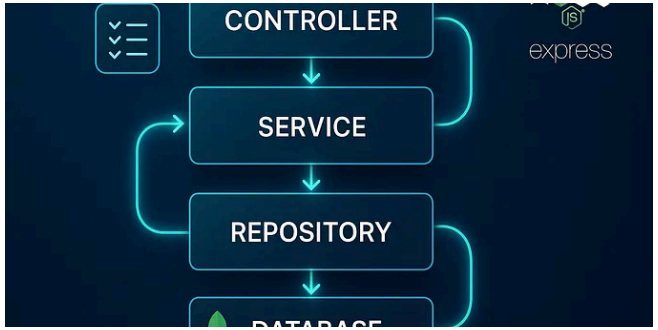


In CodeToDeploy by Shahid Islam

The Node.js Event Loop: How One Thread Handles Thousands of...

Unpacking Non-Blocking I/O, the Libuv Thread Pool, and Asynchronous JavaScript...

Oct 6, 2025 51

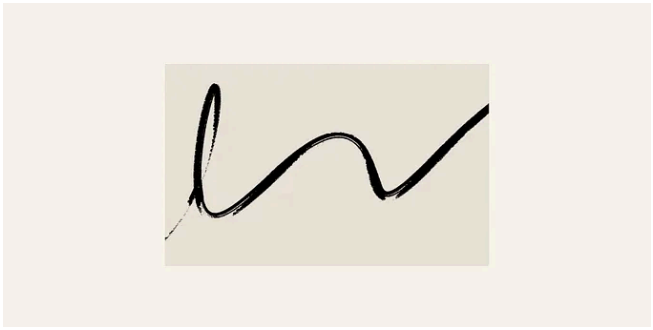


Shamsuddeen Omacy

Implementing the Repository Design Pattern in Node.js &...

What's Covered:

Oct 25, 2025



Mike Code

Nodejs | ExpressJs | Javascripe | Socketlo | React| Build a Real tim...

We will learn to use nodejs , expressjs and socketio and React to build a realtime chat...

Nov 6, 2025 1 1

See more recommendations