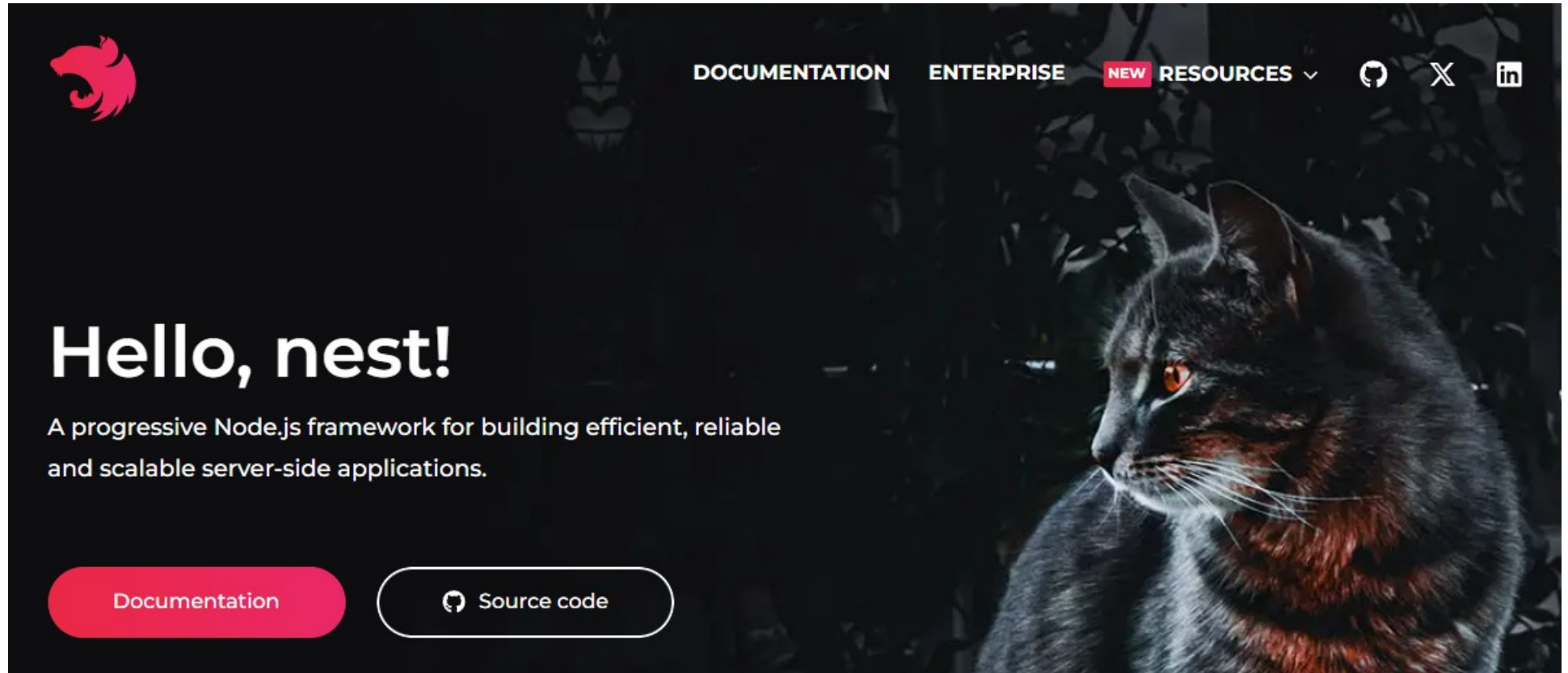


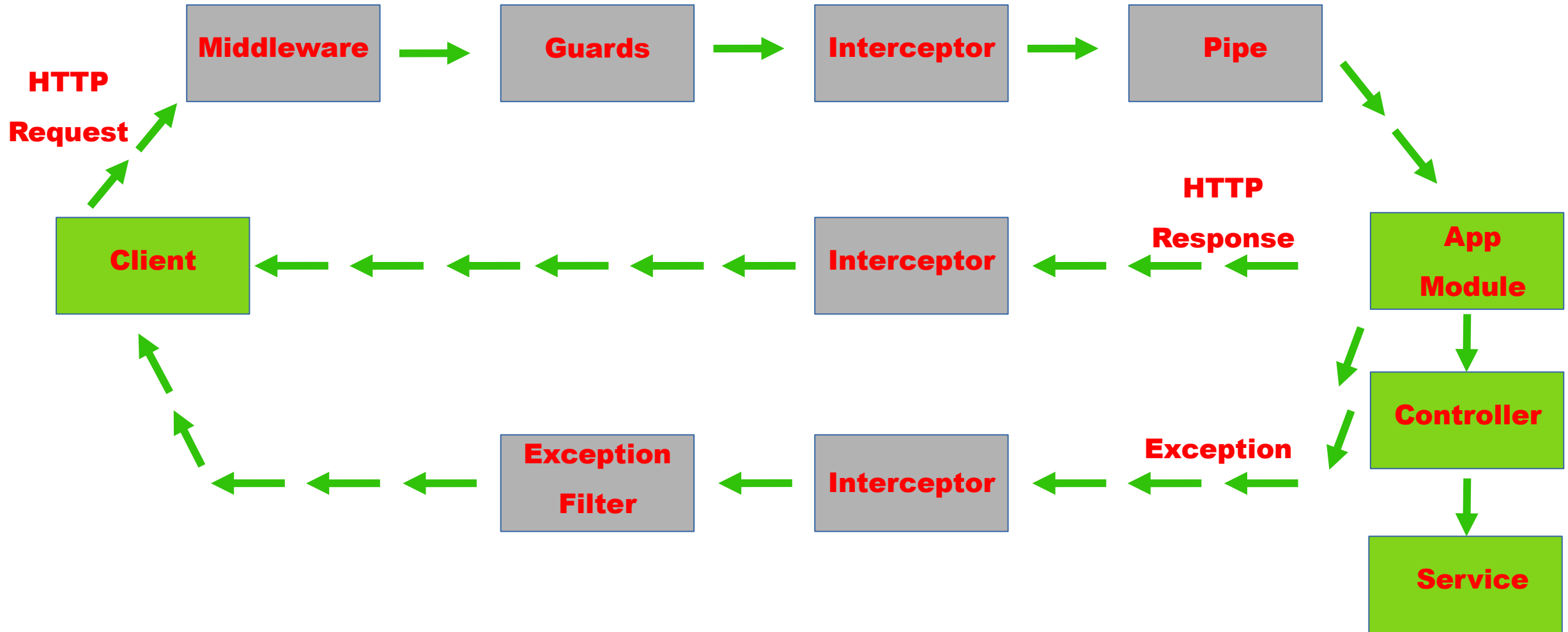
Nest.js



Prof. Dr. Robyson Aggio

www.aggioirati.com.br

Estrutura





INTRODUCTION

OVERVIEW

FUNDAMENTALS

TECHNIQUES

SECURITY

GRAPHQL

WEBSOCKETS

MICROSERVICES

NEW DEPLOYMENT

STANDALONE APPS

CLI

OPENAPI

RECIPES

REPL

CRUD generator

SWC (fast compiler)

Passport (auth)

Hot reload

MikroORM

TypeORM

Mongoose

Sequelize

Router module

Swagger

Health checks

CQRS

Compodoc

Prisma

Sentry

Serve static

Prisma



Prisma is an **open-source** ORM for Node.js and TypeScript. It is used as an **alternative** to writing plain SQL, or using another database access tool such as SQL query builders (like **knex.js**) or ORMs (like **TypeORM** and **Sequelize**). Prisma currently supports PostgreSQL, MySQL, SQL Server, SQLite, MongoDB and CockroachDB (**Preview**).

While Prisma can be used with plain JavaScript, it embraces TypeScript and provides a level to type-safety that goes beyond the guarantees other ORMs in the TypeScript ecosystem. You can find an in-depth comparison of the type-safety guarantees of Prisma and TypeORM **here**.

NOTE

If you want to get a quick overview of how Prisma works, you can follow the **Quickstart** or read the **Introduction** in the **documentation**. There also are ready-to-run examples for **REST** and **GraphQL** in the **prisma-examples** repo.

Getting started

In this recipe, you'll learn how to get started with NestJS and Prisma from scratch. You are going to build a sample NestJS application with a REST API that can read and write data in a database.

For the purpose of this guide, you'll use a **SQLite** database to save the overhead of setting up a database server. Note that you can still follow this guide, even if you're using PostgreSQL or MySQL – you'll get extra instructions for using these databases at the right places.

NOTE

If you already have an existing project and consider migrating to Prisma, you can follow the guide for **adding Prisma to an existing project**. If you are migrating from TypeORM, you can read the guide **Migrating from TypeORM to Prisma**.

Create your NestJS project

To get started, install the NestJS CLI and create your app skeleton with the following commands:

Prisma

Getting started

Create your NestJS project

Set up Prisma

Set the database connection

Create two database tables with Prisma Migrate

Install and generate Prisma Client

Use Prisma Client in your NestJS services

Summary



Design and Development tips in your inbox. Every weekday.

ADS VIA CARBON

- INTRODUCTION
- OVERVIEW
- FUNDAMENTALS
- TECHNIQUES
- SECURITY
- GRAPHQL
- WEBSOCKETS
- MICROSERVICES
- NEW DEPLOYMENT
- STANDALONE APPS
- CLI
- OPENAPI
- RECIPES
 - REPL
 - CRUD generator
 - SWC (fast compiler)
 - Passport (auth)
 - Hot reload
 - MikroORM
 - TypeORM
 - Mongoose
 - Sequelize

Set up Prisma

Start by installing the Prisma CLI as a development dependency in your project:

```
$ cd hello-prisma  
$ npm install prisma --save-dev
```

In the following steps, we'll be utilizing the **Prisma CLI**. As a best practice, it's recommended to invoke the CLI locally by prefixing it with `npx`:

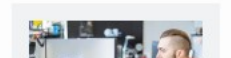
```
$ npx prisma
```

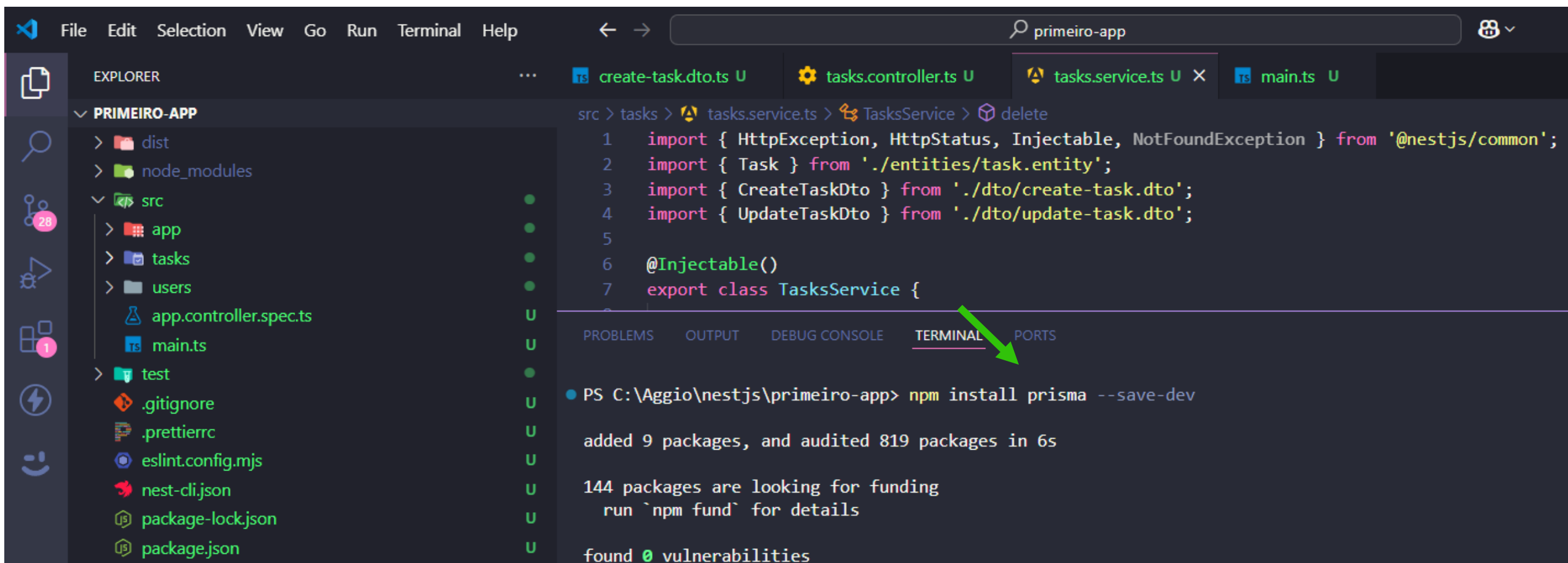
► Expand if you're using Yarn

Now create your initial Prisma setup using the `init` command of the Prisma CLI:

```
$ npx prisma init
```

- Prisma
- Getting started
- Create your NestJS project
- Set up Prisma
- Set the database connection
- Create two database tables with Prisma Migrate
- Install and generate Prisma Client
- Use Prisma Client in your NestJS services
- Summary





The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left displaying the project structure of 'PRIMEIRO-APP'. The Explorer shows folders like 'dist', 'node_modules', 'src', and 'test', along with files like '.gitignore', '.prettierrc', 'eslint.config.mjs', 'nest-cli.json', 'package-lock.json', and 'package.json'. The main editor area displays the 'tasks.service.ts' file, which contains TypeScript code for a service class. The code includes imports for 'HttpException', 'HttpStatus', 'Injectable', and 'NotFoundException' from '@nestjs/common', and imports for 'Task', 'CreateTaskDto', and 'UpdateTaskDto' from local DTO files. The service class 'TasksService' is decorated with '@Injectable()' and exports a class. The bottom panel shows the 'TERMINAL' tab with the command 'npm install prisma --save-dev' and its output, indicating successful installation and no vulnerabilities found.

File Edit Selection View Go Run Terminal Help

primeiro-app

EXPLORER

PRIMEIRO-APP

- dist
- node_modules
- src
 - app
 - tasks
 - users
 - app.controller.spec.ts
 - main.ts
- test
- .gitignore
- .prettierrc
- eslint.config.mjs
- nest-cli.json
- package-lock.json
- package.json

src > tasks > tasks.service.ts > TasksService > delete

```
1 import { HttpException, HttpStatus, Injectable, NotFoundException } from '@nestjs/common';
2 import { Task } from './entities/task.entity';
3 import { CreateTaskDto } from './dto/create-task.dto';
4 import { UpdateTaskDto } from './dto/update-task.dto';
5
6 @Injectable()
7 export class TasksService {
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

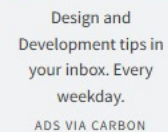
```
PS C:\Aggio\nestjs\primeiro-app> npm install prisma --save-dev
added 9 packages, and audited 819 packages in 6s
144 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

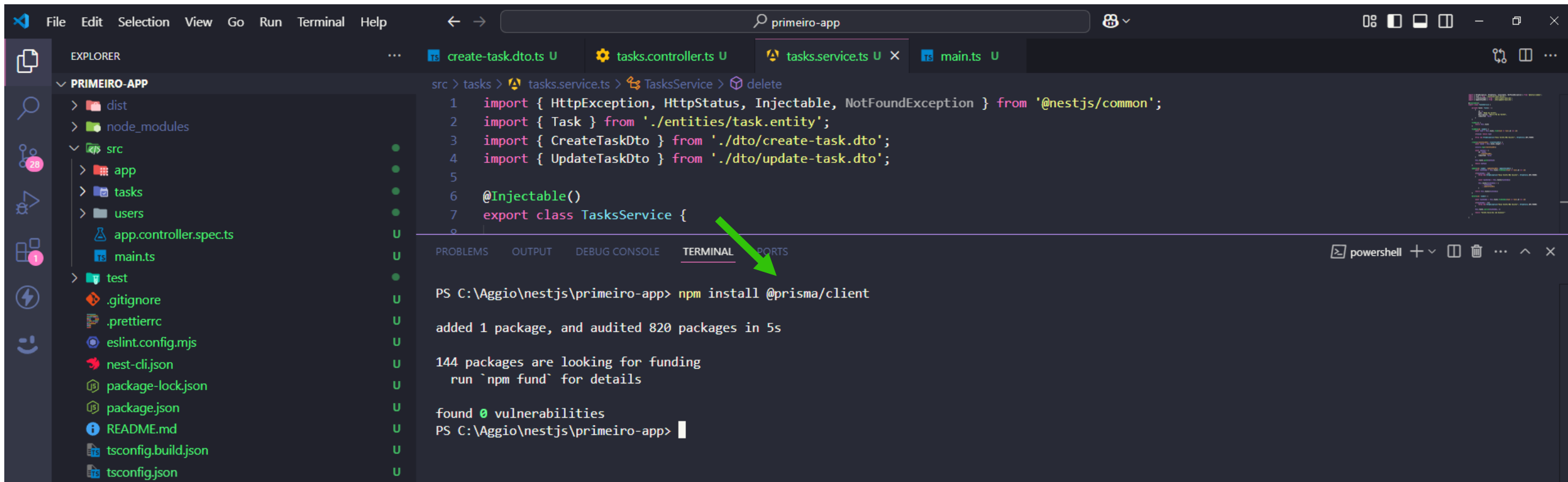
Commander

```
import { Injectable, OnModuleInit } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
```

Summary





VS Code Explorer sidebar showing project structure:

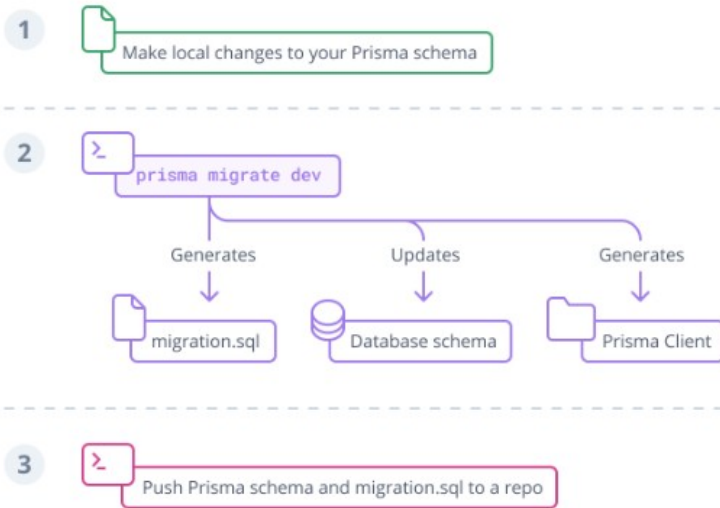
- PRIMEIRO-APP
 - dist
 - node_modules
 - src
 - app
 - tasks
 - users
 - app.controller.spec.ts
 - main.ts
 - test
 - .gitignore
 - .prettierrc
 - eslint.config.mjs
 - nest-cli.json
 - package-lock.json
 - package.json
 - README.md
 - tsconfig.build.json
 - tsconfig.json

Main editor showing `tasks.service.ts` file:

```
src > tasks > tasks.service.ts > TasksService > delete
1 import { HttpException, HttpStatus, Injectable, NotFoundException } from '@nestjs/common';
2 import { Task } from '../entities/task.entity';
3 import { CreateTaskDto } from '../dto/create-task.dto';
4 import { UpdateTaskDto } from '../dto/update-task.dto';
5
6 @Injectable()
7 export class TasksService {
8
```

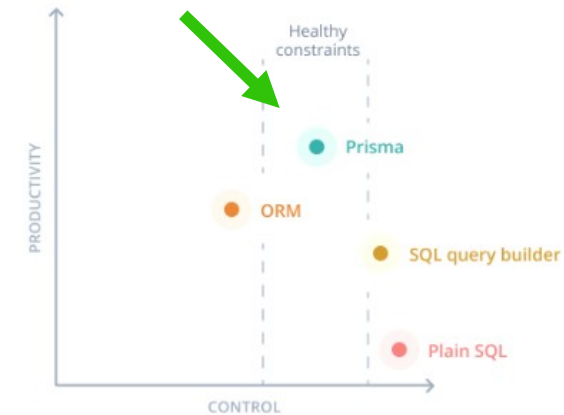
Terminal output:

```
PS C:\Aggio\nestjs\primeiro-app> npm install @prisma/client
added 1 package, and audited 820 packages in 5s
144 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
PS C:\Aggio\nestjs\primeiro-app>
```



Prisma ORM makes developers productive

Prisma ORM's main goal is to make application developers more productive when working with databases. Considering the tradeoff between productivity and control again, this is how Prisma ORM fits in:





Want real-time updates from your database without manual polling?

Home / ORM / Overview / Prisma ORM 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 ORM:

- [Express](#)
- [koa](#)
- [hapi](#)
- [Fastify](#)
- [Sails](#)
- [AdonisJs](#)
- [NestJS](#)

ON THIS PAGE

[Supported libraries](#)

[REST API server example](#)

[Ready-to-run example projects](#)

ORM

OVERVIEW

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

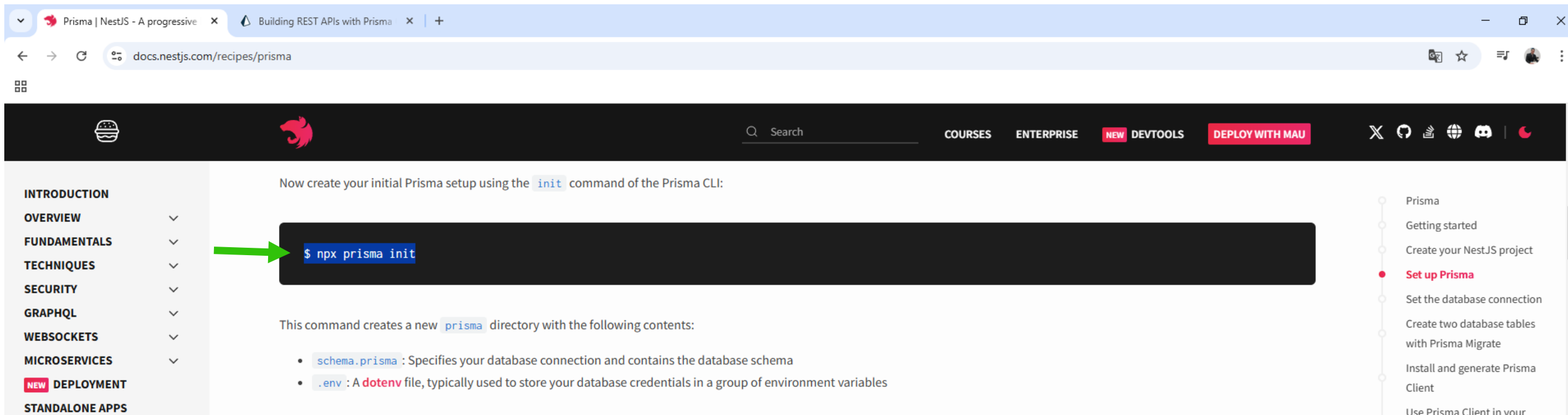
PRISMA SCHEMA

- Overview
- Data model
- Introspection
- PostgreSQL extensions

Preview

PRISMA CLIENT

- Setup & configuration
- Queries
- Write your own SQL



The screenshot shows a web browser window with two tabs: 'Prisma | NestJS - A progressive' and 'Building REST APIs with Prisma'. The address bar shows 'docs.nestjs.com/recipes/prisma'. The page has a dark header with a search bar and navigation links: COURSES, ENTERPRISE, NEW DEVTOOLS, and DEPLOY WITH MAU. A sidebar on the left lists navigation items: INTRODUCTION, OVERVIEW, FUNDAMENTALS, TECHNIQUES, SECURITY, GRAPHQL, WEBSOCKETS, MICROSERVICES, and a new section for DEPLOYMENT and STANDALONE APPS. The main content area explains how to create a Prisma setup using the 'init' command. A green arrow points to a code block containing '\$ npx prisma init'. Below this, it states that the command creates a new 'prisma' directory with specific contents: 'schema.prisma' for the database schema and '.env' for storing credentials. A right sidebar shows a table of contents with 'Set up Prisma' highlighted as the current step.

Now create your initial Prisma setup using the `init` command of the Prisma CLI:

```
$ npx prisma init
```

This command creates a new `prisma` directory with the following contents:

- `schema.prisma`: Specifies your database connection and contains the database schema
- `.env`: A `dotenv` file, typically used to store your database credentials in a group of environment variables

Prisma

Getting started

Create your NestJS project

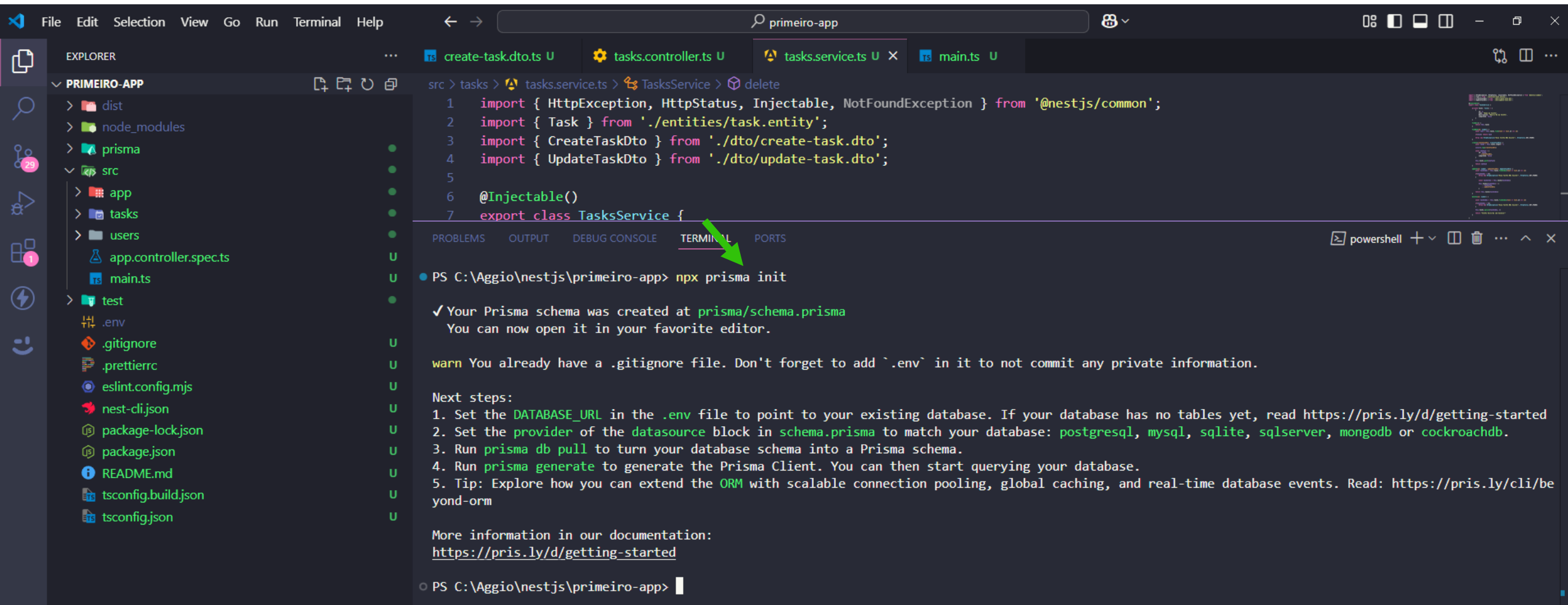
Set up Prisma

Set the database connection

Create two database tables with Prisma Migrate

Install and generate Prisma Client

Use Prisma Client in your



VS Code Explorer: PRIMEIRO-APP

- dist
- node_modules
- prisma
- src
 - app
 - tasks
 - users
 - app.controller.spec.ts
 - main.ts
- test
- .env
- .gitignore
- .prettierrc
- eslint.config.mjs
- nest-cli.json
- package-lock.json
- package.json
- README.md
- tsconfig.build.json
- tsconfig.json

VS Code Editor: tasks.service.ts

```
1 import { HttpException, HttpStatus, Injectable, NotFoundException } from '@nestjs/common';
2 import { Task } from '../entities/task.entity';
3 import { CreateTaskDto } from '../dto/create-task.dto';
4 import { UpdateTaskDto } from '../dto/update-task.dto';
5
6 @Injectable()
7 export class TasksService {
```

VS Code Terminal:

```
PS C:\Aggio\nestjs\primeiro-app> npm prisma init

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

warn You already have a .gitignore file. Don't forget to add '.env' in it to not commit any private information.

Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing database. If your database has no tables yet, read https://pris.ly/d/getting-started
2. Set the provider of the datasource block in schema.prisma to match your database: postgresql, mysql, sqlite, sqlserver, mongodb or cockroachdb.
3. Run prisma db pull to turn your database schema into a Prisma schema.
4. Run prisma generate to generate the Prisma Client. You can then start querying your database.
5. Tip: Explore how you can extend the ORM with scalable connection pooling, global caching, and real-time database events. Read: https://pris.ly/cli/byeond-orm

More information in our documentation:
https://pris.ly/d/getting-started

PS C:\Aggio\nestjs\primeiro-app>
```

Set the database connection

Your database connection is configured in the `datasource` block in your `schema.prisma` file. By default it's set to `postgresql`, but since you're using a SQLite database in this guide you need to adjust the `provider` field of the `datasource` block to `sqlite`:

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

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

Now, open up `.env` and adjust the `DATABASE_URL` environment variable to look as follows:

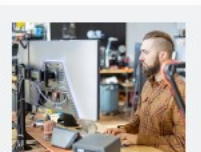
```
DATABASE_URL="file:./dev.db"
```

Make sure you have a `ConfigModule` configured, otherwise the `DATABASE_URL` variable will not be picked up from `.env`.

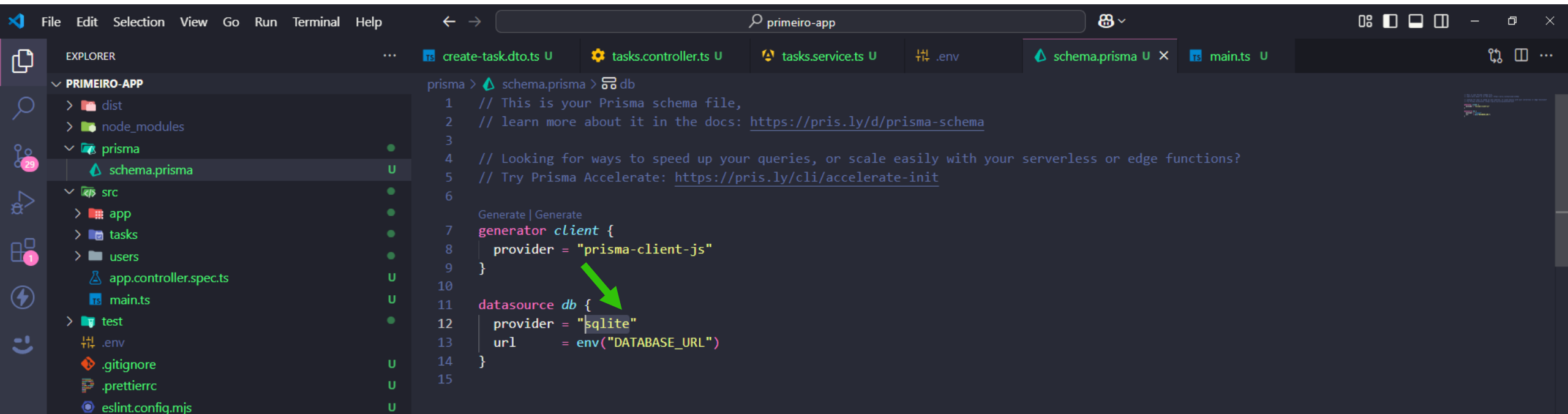
SQLite databases are simple files; no server is required to use a SQLite database. So instead of configuring a connection URL with a `host` and `port`, you can just point it to a local file which in this case is called `dev.db`. This file will be created in the next step.

► Expand if you're using PostgreSQL, MySQL, MsSQL or Azure SQL

- Prisma
- Getting started
- Create your NestJS project
- Set up Prisma
- Set the database connection**
- Create two database tables with Prisma Migrate
- Install and generate Prisma Client
- Use Prisma Client in your NestJS services
- Summary

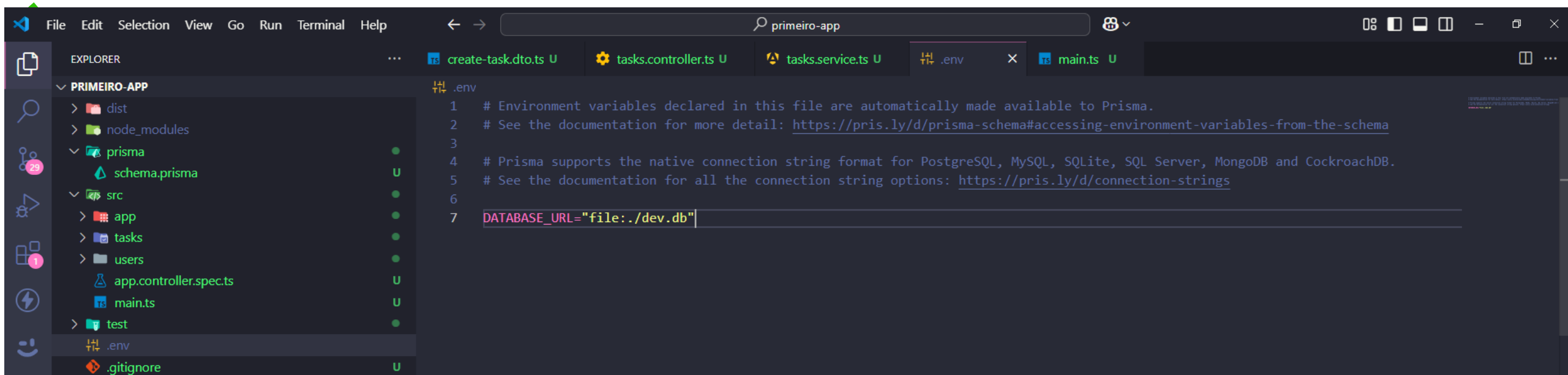


Design and Development tips in



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays the project structure for 'PRIMEIRO-APP', including folders like 'dist', 'node_modules', 'prisma', 'src', and 'test', along with various configuration files. The 'prisma' folder is expanded, showing 'schema.prisma'. The main editor area displays the content of 'schema.prisma', which is a Prisma schema file. The schema defines a 'client' generator and a 'db' data source using 'sqlite' as the provider. A green arrow points to the 'sqlite' provider in the 'db' data source definition. The top of the editor shows the file explorer with tabs for 'create-task.dto.ts', 'tasks.controller.ts', 'tasks.service.ts', '.env', 'schema.prisma', and 'main.ts'.

```
prisma > schema.prisma > db
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 // Looking for ways to speed up your queries, or scale easily with your serverless or edge functions?
5 // Try Prisma Accelerate: https://pris.ly/cli/accelerate-init
6
7 generator client {
8   provider = "prisma-client-js"
9 }
10
11 datasource db {
12   provider = "sqlite"
13   url      = env("DATABASE_URL")
14 }
15
```

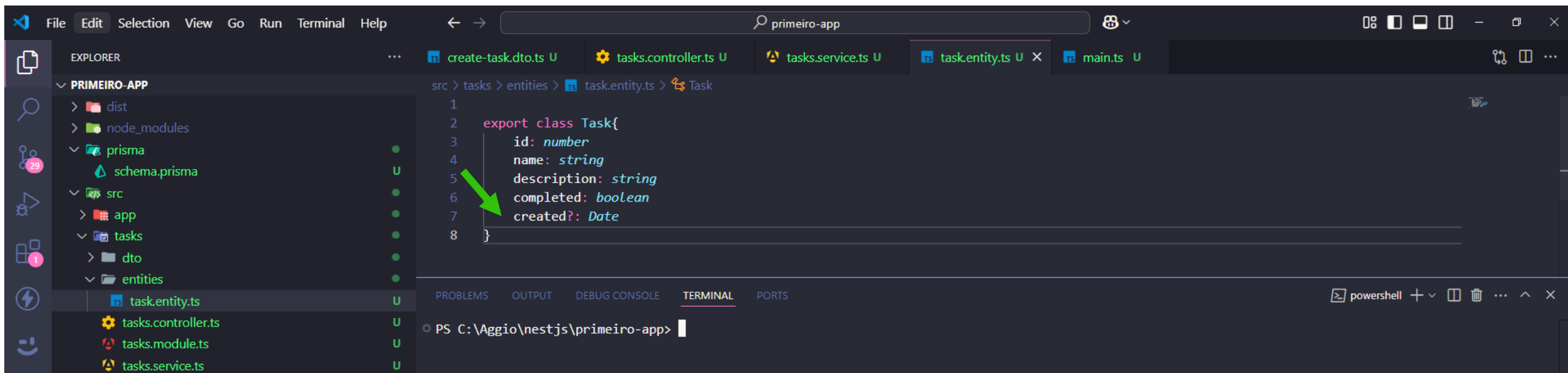


The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays the project structure for 'PRIMEIRO-APP':

- dist
- node_modules
- prisma
 - schema.prisma
- src
 - app
 - tasks
 - users
 - app.controller.spec.ts
 - main.ts
- test
- .env
- .gitignore

The main editor area shows the content of the `.env` file:

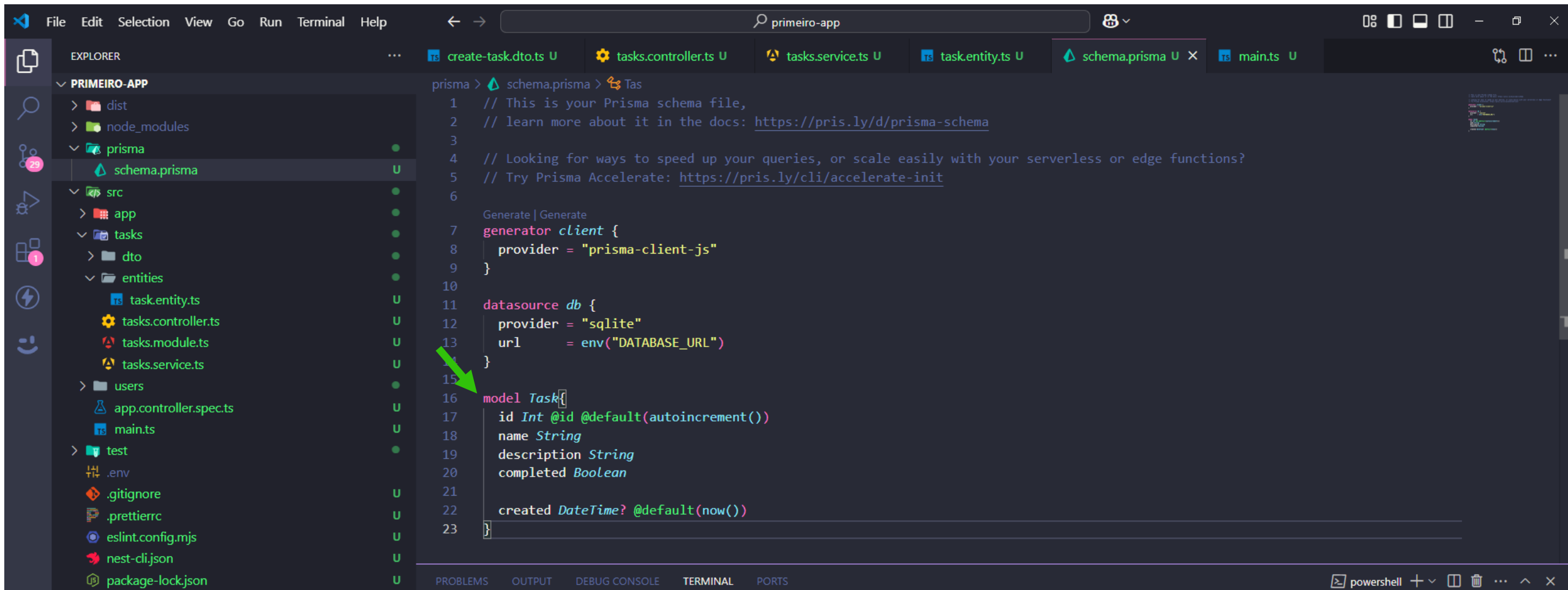
```
1 # Environment variables declared in this file are automatically made available to Prisma.
2 # See the documentation for more detail: https://pris.ly/d/prisma-schema#accessing-environment-variables-from-the-schema
3
4 # Prisma supports the native connection string format for PostgreSQL, MySQL, SQLite, SQL Server, MongoDB and CockroachDB.
5 # See the documentation for all the connection string options: https://pris.ly/d/connection-strings
6
7 DATABASE_URL="file:./dev.db"
```



The screenshot shows the Visual Studio Code editor interface. The Explorer panel on the left displays the project structure for 'PRIMEIRO-APP', including folders like 'dist', 'node_modules', 'prisma', 'src', 'tasks', 'dto', and 'entities'. The 'prisma' folder contains 'schema.prisma'. The 'src' folder contains 'app', 'tasks', 'dto', and 'entities'. The 'tasks' folder contains 'task.entity.ts', 'tasks.controller.ts', 'tasks.module.ts', and 'tasks.service.ts'. The 'entities' folder is expanded, showing 'task.entity.ts' selected. The main editor area displays the content of 'task.entity.ts', which is a TypeScript class definition for 'Task'. The class has four properties: 'id' (number), 'name' (string), 'description' (string), and 'completed' (boolean). The 'created?' property is of type 'Date'. A green arrow points to the 'completed' property. The bottom panel shows the 'TERMINAL' tab with a PowerShell prompt at 'C:\Aggio\nestjs\primeiro-app>'.

```
1
2 export class Task{
3   id: number
4   name: string
5   description: string
6   completed: boolean
7   created?: Date
8 }
```

PS C:\Aggio\nestjs\primeiro-app>



The image shows a Visual Studio Code editor window with a dark theme. The Explorer sidebar on the left displays the project structure for 'PRIMEIRO-APP'. The main editor area shows the 'schema.prisma' file, which contains a Prisma schema definition. A green arrow points to the 'model Task' definition on line 16.

EXPLORER

- PRIMEIRO-APP
 - dist
 - node_modules
 - prisma
 - schema.prisma
 - src
 - app
 - tasks
 - dto
 - entities
 - task.entity.ts
 - tasks.controller.ts
 - tasks.module.ts
 - tasks.service.ts
 - users
 - app.controller.spec.ts
 - main.ts
 - test
 - .env
 - .gitignore
 - .prettierrc
 - eslint.config.mjs
 - nest-cli.json
 - package-lock.json

schema.prisma

```
prisma > schema.prisma > Tas
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 // Looking for ways to speed up your queries, or scale easily with your serverless or edge functions?
5 // Try Prisma Accelerate: https://pris.ly/cli/accelerate-init
6
7 Generate | Generate
8 generator client {
9   provider = "prisma-client-js"
10 }
11
12 datasource db {
13   provider = "sqlite"
14   url      = env("DATABASE_URL")
15 }
16 model Task {
17   id Int @id @default(autoincrement())
18   name String
19   description String
20   completed Boolean
21
22   created DateTime? @default(now())
23 }
```



Search

COURSES

ENTERPRISE

NEW DEVTOOLS

DEPLOY WITH MAU



INTRODUCTION

OVERVIEW

FUNDAMENTALS

TECHNIQUES

SECURITY

GRAPHQL

WEBSOCKETS

MICROSERVICES

NEW DEPLOYMENT

STANDALONE APPS

CLI

OPENAPI

RECIPES

REPL

CRUD generator

SWC (fast compiler)

Passport (auth)

Hot reload

MikroORM

TypeORM

Mongoose

Sequelize

Router module

Swagger

Health checks

CQRS

Compodoc

Prisma

Create two database tables with Prisma Migrate

In this section, you'll create two new tables in your database using **Prisma Migrate**. Prisma Migrate generates SQL migration files for your declarative data model definition in the Prisma schema. These migration files are fully customizable so that you can configure any additional features of the underlying database or include additional commands, e.g. for seeding.

Add the following two models to your `schema.prisma` file:

```
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?
}
```

With your Prisma models in place, you can generate your SQL migration files and run them against the database. Run the following commands in your terminal:

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

Prisma

Getting started

Create your NestJS project

Set up Prisma

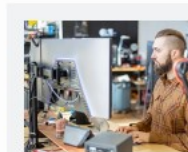
Set the database connection

Create two database tables with Prisma Migrate

Install and generate Prisma Client

Use Prisma Client in your NestJS services

Summary



Design and Development tips in your inbox. Every

Prisma

PRIMEIRO-APP

dist

node_modules

prisma

migrations

dev.db

dev.db-journal

schema.prisma

src

app

tasks

dto

entities

task.entity.ts

tasks.controller.ts

tasks.module.ts

tasks.service.ts

users

app.controller.spec.ts

main.ts

test

.env

.gitignore

.prettierrc

eslint.config.mjs

nest-cli.json

package-lock.json

package.json

README.md

tsconfig.build.json

tsconfig.json

```
prisma > schema.prisma > Tas
10
11 datasource db {
12   provider = "sqlite"
13   url       = env("DATABASE_URL")
14 }
15
16 model Task{
17   id Int @id @default(autoincrement())
18   name String
19   description String
20   completed Boolean
21
22   created DateTime? @default(now())
23 }
```

TERMINAL

PS C:\Aggio\nestjs\primeiro-app> npx prisma migrate dev

Environment variables loaded from .env

Prisma schema loaded from prisma\schema.prisma

Datasource "db": SQLite database "dev.db" at "file:./dev.db"

SQLite database dev.db created at file:./dev.db

✓ Enter a name for the new migration: ... create-table-tasks

Applying migration `20250226221720\_create\_table\_tasks`

The following migration(s) have been created and applied from new schema changes:

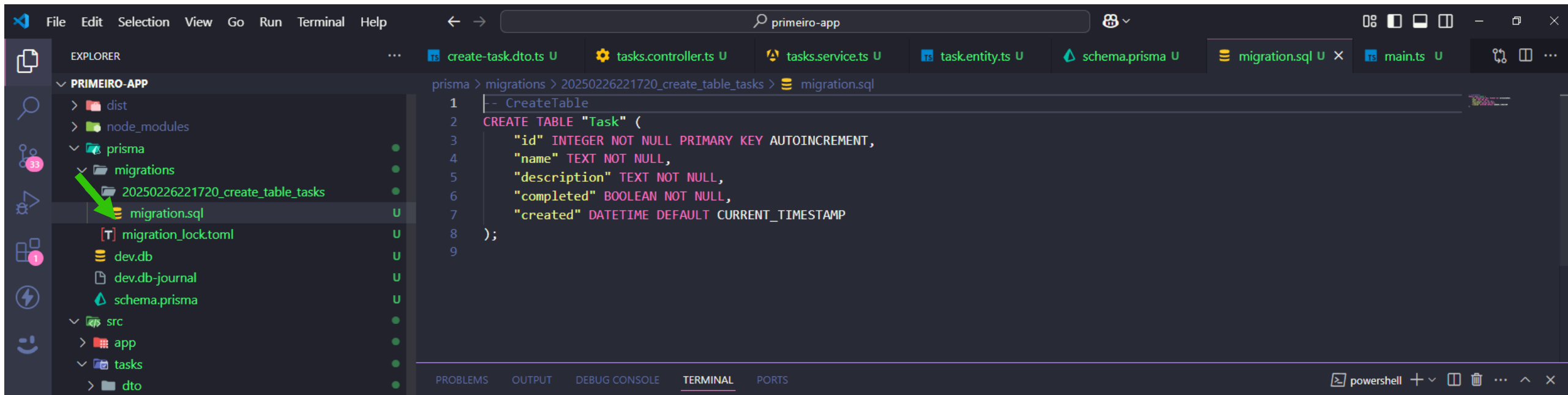
migrations/

- 20250226221720_create_table_tasks/
 - migration.sql

Your database is now in sync with your schema.

✓ Generated Prisma Client (v6.4.1) to .\node_modules\@prisma\client in 46ms

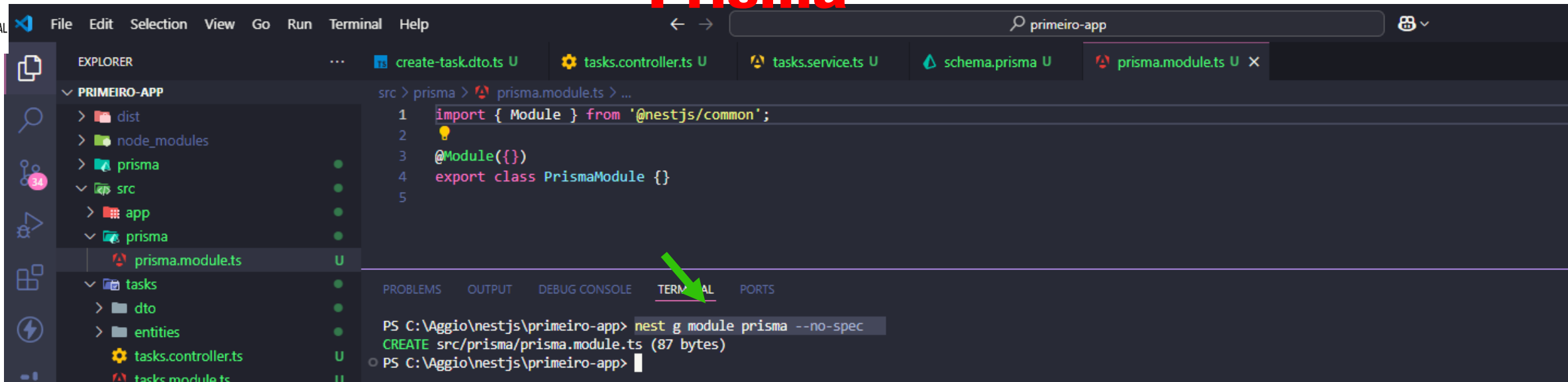
PS C:\Aggio\nestjs\primeiro-app>



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays the project structure for 'PRIMEIRO-APP'. A green arrow points to the 'migration.sql' file within the 'migrations' folder. The main editor window shows the content of 'migration.sql', which is a Prisma migration file. The file path in the breadcrumb is 'prisma > migrations > 20250226221720_create_table_tasks > migration.sql'. The code defines a 'Task' table with columns: 'id' (INTEGER, NOT NULL, PRIMARY KEY, AUTOINCREMENT), 'name' (TEXT, NOT NULL), 'description' (TEXT, NOT NULL), 'completed' (BOOLEAN, NOT NULL), and 'created' (DATETIME, DEFAULT CURRENT_TIMESTAMP). The bottom status bar shows the active terminal is 'powershell'.

```
prisma > migrations > 20250226221720_create_table_tasks > migration.sql
1  -- CreateTable
2  CREATE TABLE "Task" (
3      "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
4      "name" TEXT NOT NULL,
5      "description" TEXT NOT NULL,
6      "completed" BOOLEAN NOT NULL,
7      "created" DATETIME DEFAULT CURRENT_TIMESTAMP
8  );
9
```

Prisma



File Edit Selection View Go Run Terminal Help

primeiro-app

EXPLORER

- PRIMEIRO-APP
 - dist
 - node_modules
 - prisma
 - src
 - app
 - prisma
 - prisma.module.ts
 - tasks
 - dto
 - entities
 - tasks.controller.ts
 - tasks.module.ts

src > prisma > prisma.module.ts > ...

```

1 import { Module } from '@nestjs/common';
2
3 @Module({})
4 export class PrismaModule {}
5

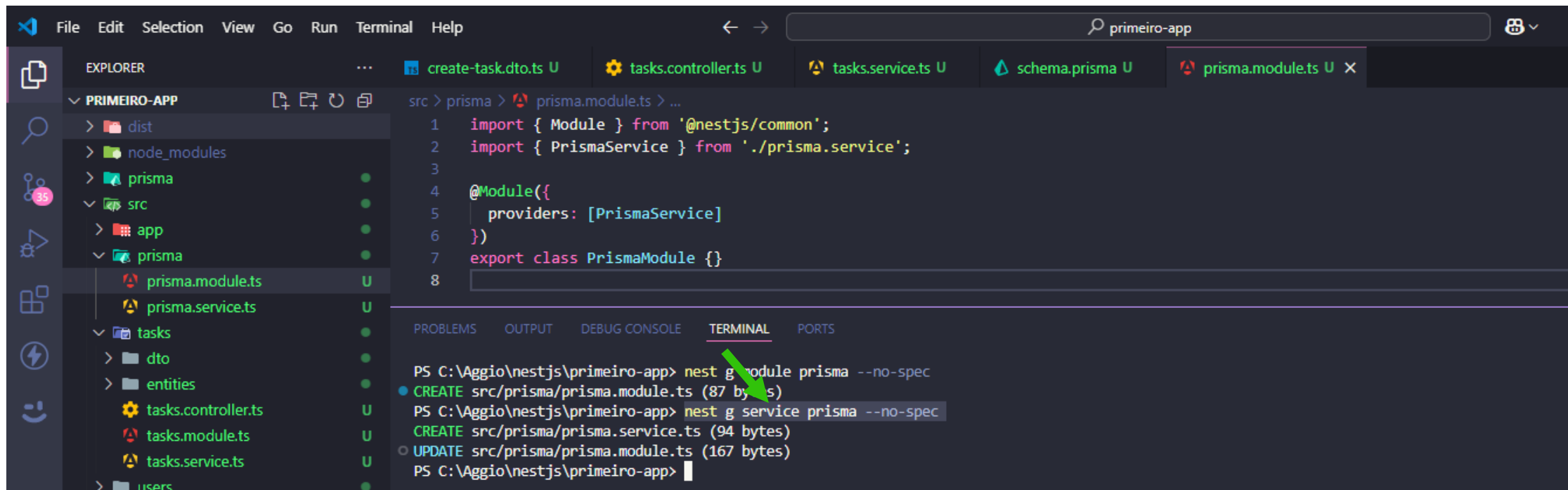
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

PS C:\Aggio\nestjs\primeiro-app> nest g module prisma --no-spec
CREATE src/prisma/prisma.module.ts (87 bytes)
PS C:\Aggio\nestjs\primeiro-app>

```



File Edit Selection View Go Run Terminal Help

primeiro-app

EXPLORER

- PRIMEIRO-APP
 - dist
 - node_modules
 - prisma
 - src
 - app
 - prisma
 - prisma.module.ts
 - prisma.service.ts
 - tasks
 - dto
 - entities
 - tasks.controller.ts
 - tasks.module.ts
 - tasks.service.ts

src > prisma > prisma.module.ts > ...

```

1 import { Module } from '@nestjs/common';
2 import { PrismaService } from './prisma.service';
3
4 @Module({
5   providers: [PrismaService]
6 })
7 export class PrismaModule {}
8

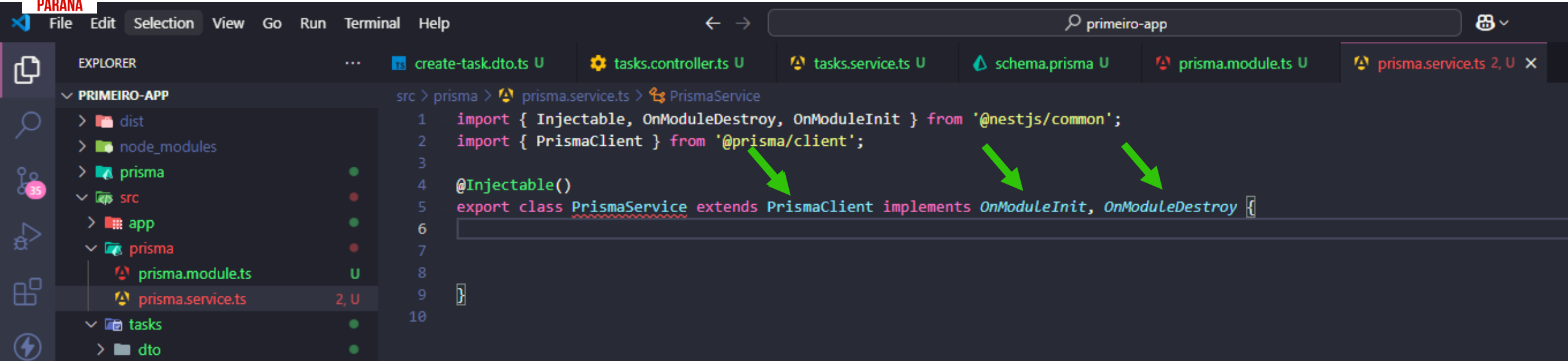
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

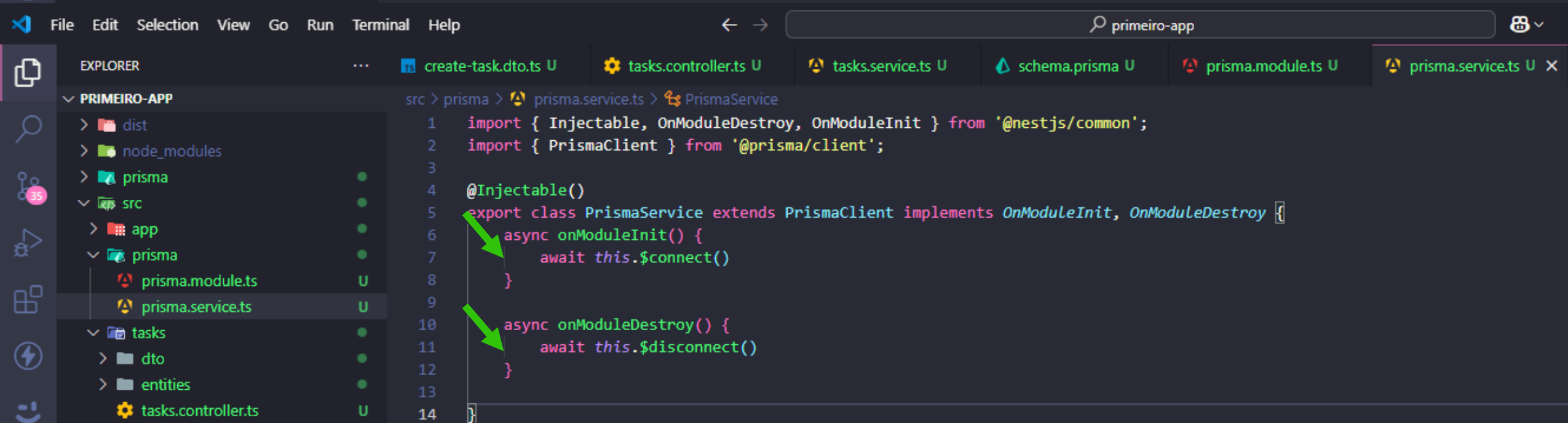
```

PS C:\Aggio\nestjs\primeiro-app> nest g module prisma --no-spec
CREATE src/prisma/prisma.module.ts (87 bytes)
PS C:\Aggio\nestjs\primeiro-app> nest g service prisma --no-spec
CREATE src/prisma/prisma.service.ts (94 bytes)
UPDATE src/prisma/prisma.module.ts (167 bytes)
PS C:\Aggio\nestjs\primeiro-app>

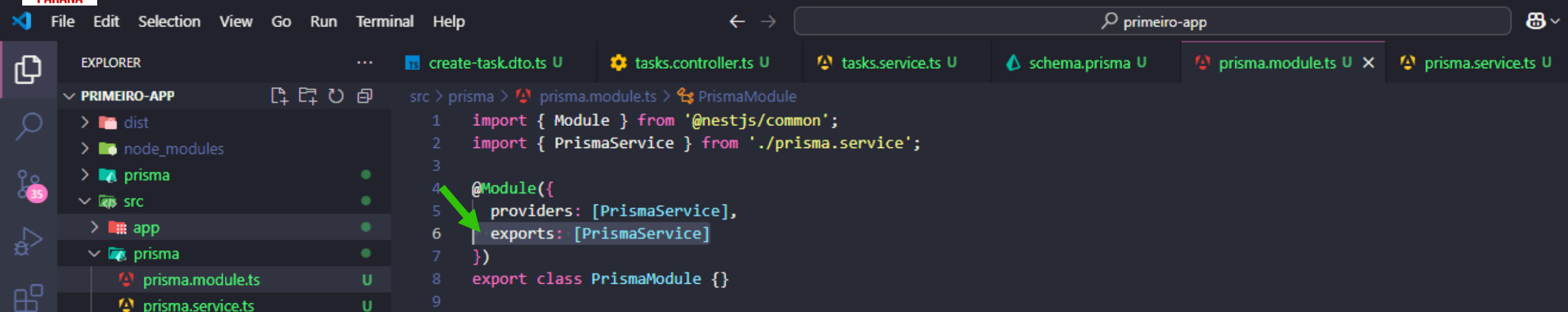
```



```
src > prisma > prisma.service.ts > PrismaService
1  import { Injectable, OnModuleDestroy, OnModuleInit } from '@nestjs/common';
2  import { PrismaClient } from '@prisma/client';
3
4  @Injectable()
5  export class PrismaService extends PrismaClient implements OnModuleInit, OnModuleDestroy {
6
7
8
9
10
```

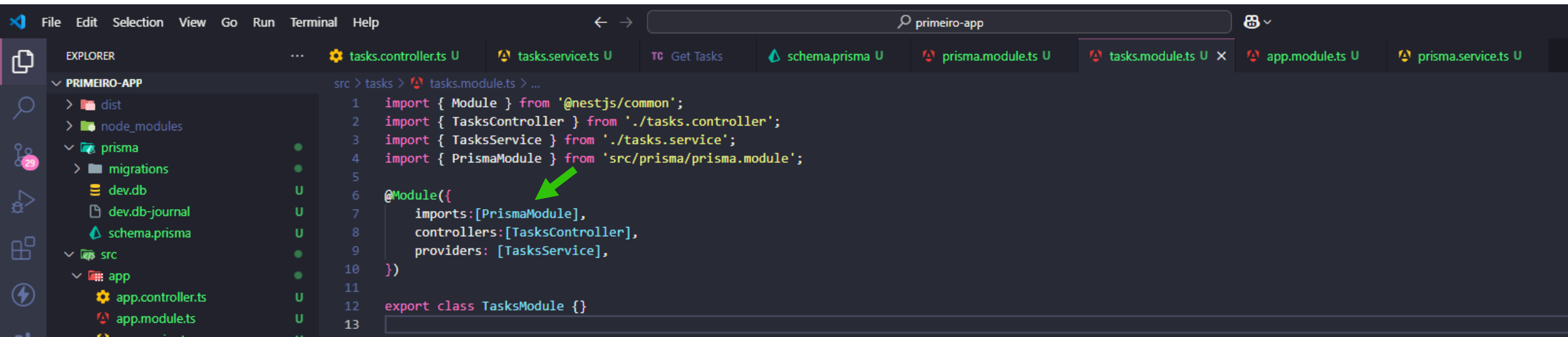


```
src > prisma > prisma.service.ts > PrismaService
1  import { Injectable, OnModuleDestroy, OnModuleInit } from '@nestjs/common';
2  import { PrismaClient } from '@prisma/client';
3
4  @Injectable()
5  export class PrismaService extends PrismaClient implements OnModuleInit, OnModuleDestroy {
6
7  async onModuleInit() {
8    await this.$connect()
9  }
10
11  async onModuleDestroy() {
12    await this.$disconnect()
13  }
14
```



The screenshot shows the Visual Studio Code editor interface for a project named 'primeiro-app'. The Explorer sidebar on the left displays the project structure, including folders like 'dist', 'node_modules', 'prisma', and 'src'. The 'src' folder is expanded, showing subfolders 'app' and 'prisma'. The 'prisma' folder contains files 'prisma.module.ts' and 'prisma.service.ts'. The main editor area displays the content of 'prisma.module.ts', which is a NestJS module definition. A green arrow points to the 'exports' property in the '@Module' decorator, which is set to '[PrismaService]'. The code in the editor is as follows:

```
1 import { Module } from '@nestjs/common';
2 import { PrismaService } from '../prisma.service';
3
4 @Module({
5   providers: [PrismaService],
6   exports: [PrismaService]
7 })
8 export class PrismaModule {}
9
```



The screenshot shows the Visual Studio Code editor interface for the same 'primeiro-app' project. The Explorer sidebar on the left shows the project structure, with the 'src' folder expanded to show the 'tasks' folder. The 'tasks' folder contains files 'tasks.module.ts', 'tasks.controller.ts', and 'tasks.service.ts'. The main editor area displays the content of 'tasks.module.ts', which is a NestJS module definition. A green arrow points to the 'imports' property in the '@Module' decorator, which is set to '[PrismaModule]'. The code in the editor is as follows:

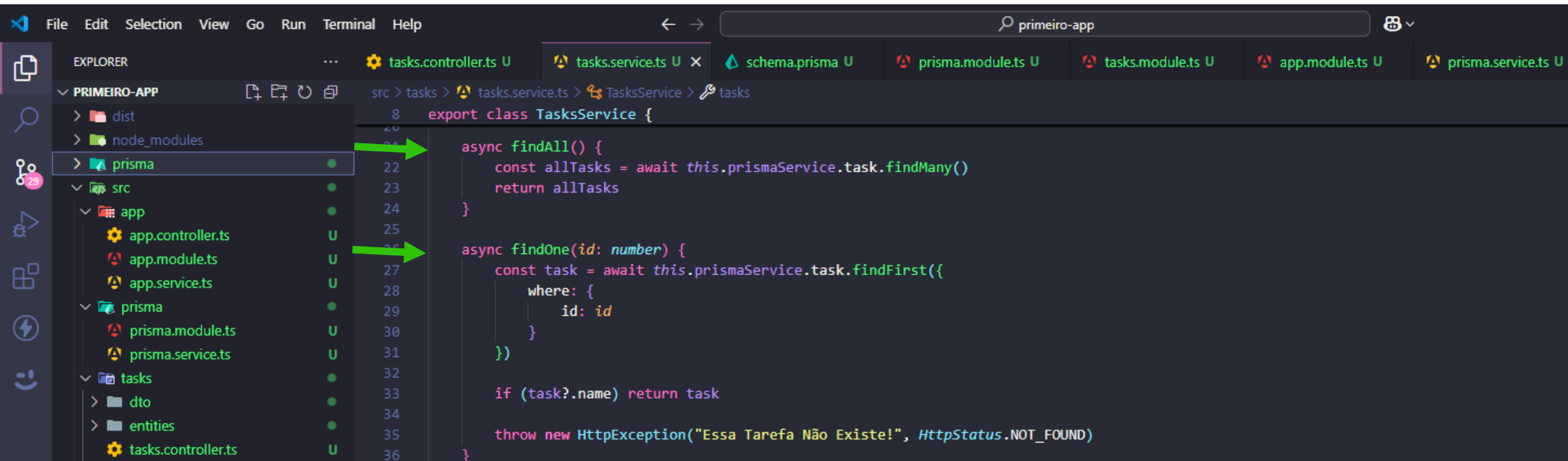
```
1 import { Module } from '@nestjs/common';
2 import { TasksController } from '../tasks.controller';
3 import { TasksService } from '../tasks.service';
4 import { PrismaModule } from 'src/prisma/prisma.module';
5
6 @Module({
7   imports: [PrismaModule],
8   controllers: [TasksController],
9   providers: [TasksService],
10 })
11
12 export class TasksModule {}
13
```



```

1 import { HttpException, HttpStatus, Injectable, NotFoundException } from '@nestjs/common';
2 import { Task } from '../entities/task.entity';
3 import { CreateTaskDto } from '../dto/create-task.dto';
4 import { UpdateTaskDto } from '../dto/update-task.dto';
5 import { PrismaService } from 'src/prisma/prisma.service';
6
7 @Injectable()
8 export class TasksService {
9
10   constructor(private readonly prismaService: PrismaService) {}
11

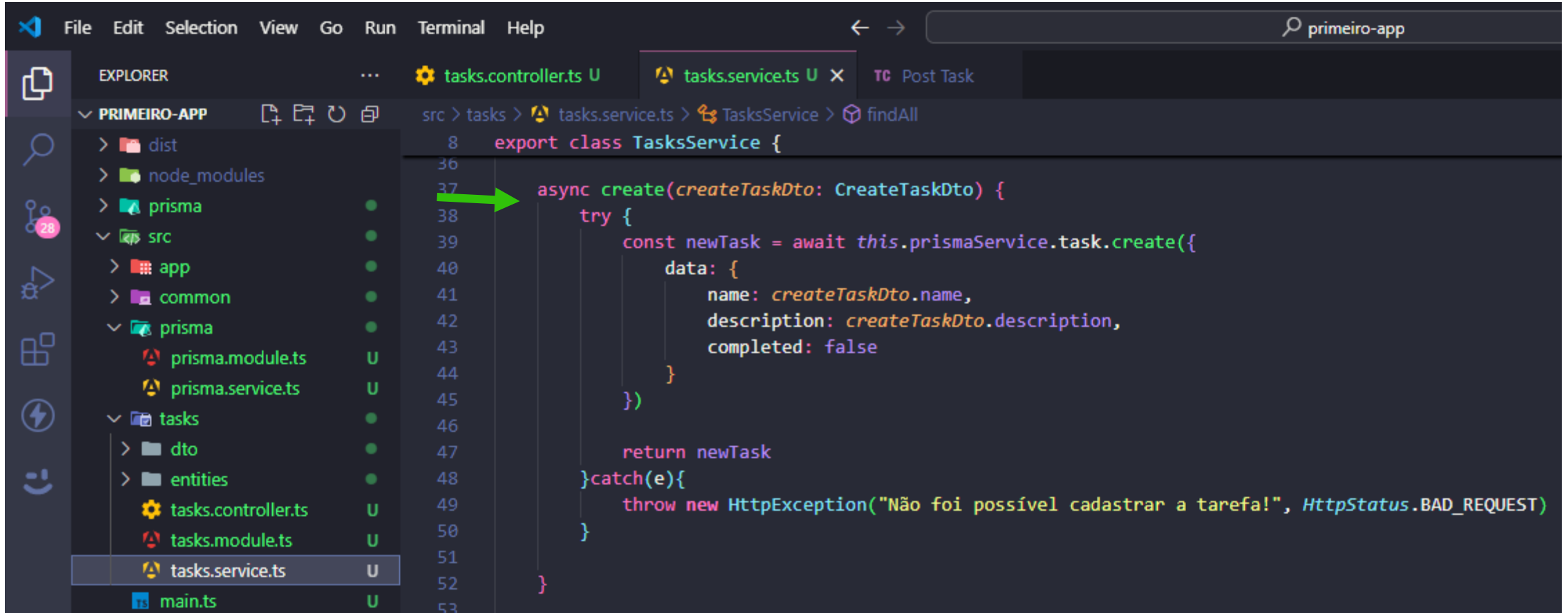
```



```

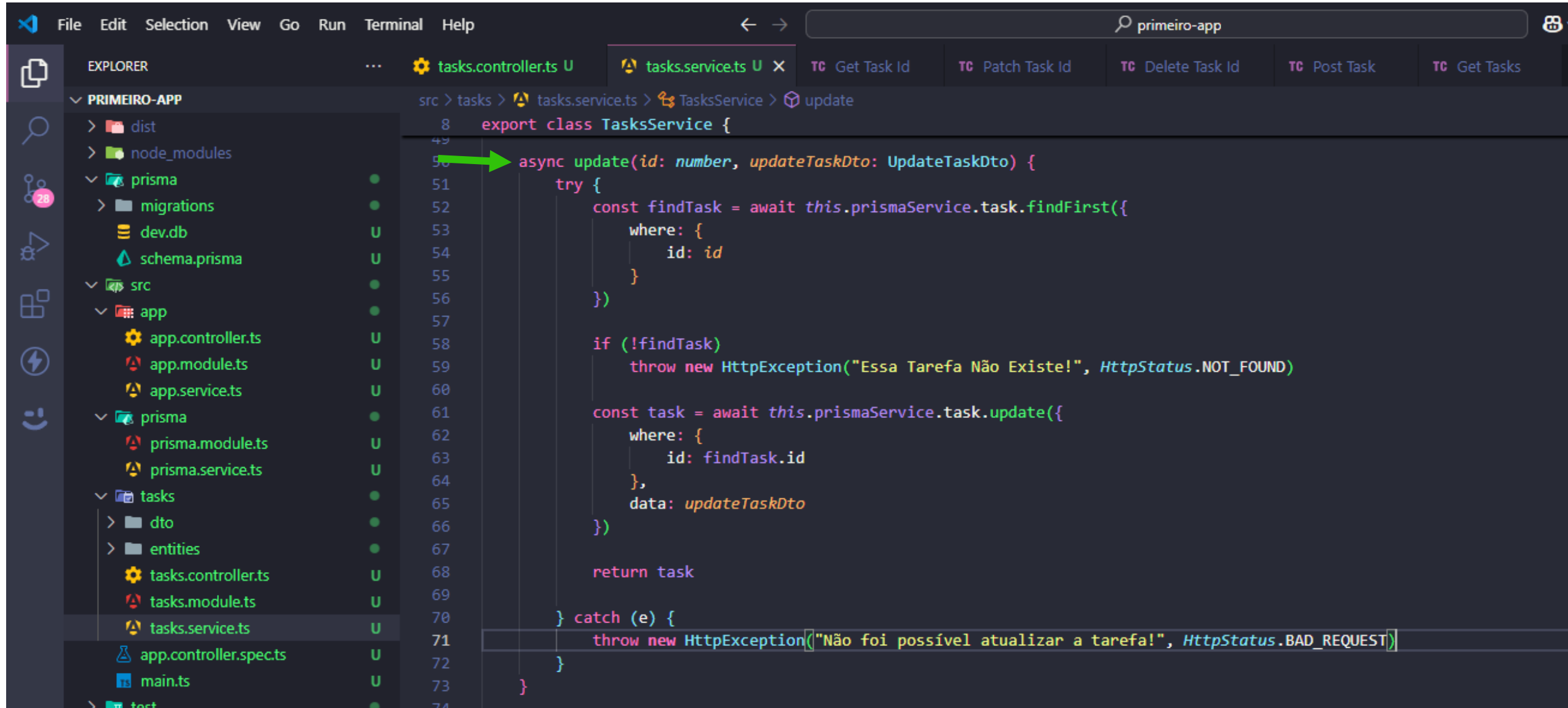
8 export class TasksService {
9
10   async findMany() {
11     const allTasks = await this.prismaService.task.findMany();
12     return allTasks;
13   }
14
15   async findOne(id: number) {
16     const task = await this.prismaService.task.findFirst({
17       where: {
18         id: id
19       }
20     });
21
22     if (task?.name) return task;
23
24     throw new HttpException("Essa Tarefa Não Existe!", HttpStatus.NOT_FOUND);
25   }
26 }

```



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays the project structure for 'PRIMEIRO-APP', with the file 'tasks.service.ts' selected under the 'tasks' directory. The main editor area shows the code for 'tasks.service.ts'. A green arrow points to the 'create' method, which is an asynchronous function that takes a 'createTaskDto' parameter and returns a 'Task' object. The method uses 'await this.prismaService.task.create' to interact with the database. The code is as follows:

```
8 export class TaskService {
36
37   async create(createTaskDto: CreateTaskDto) {
38     try {
39       const newTask = await this.prismaService.task.create({
40         data: {
41           name: createTaskDto.name,
42           description: createTaskDto.description,
43           completed: false
44         }
45       })
46
47       return newTask
48     } catch (e) {
49       throw new HttpException("Não foi possível cadastrar a tarefa!", HttpStatus.BAD_REQUEST)
50     }
51   }
52 }
53
```

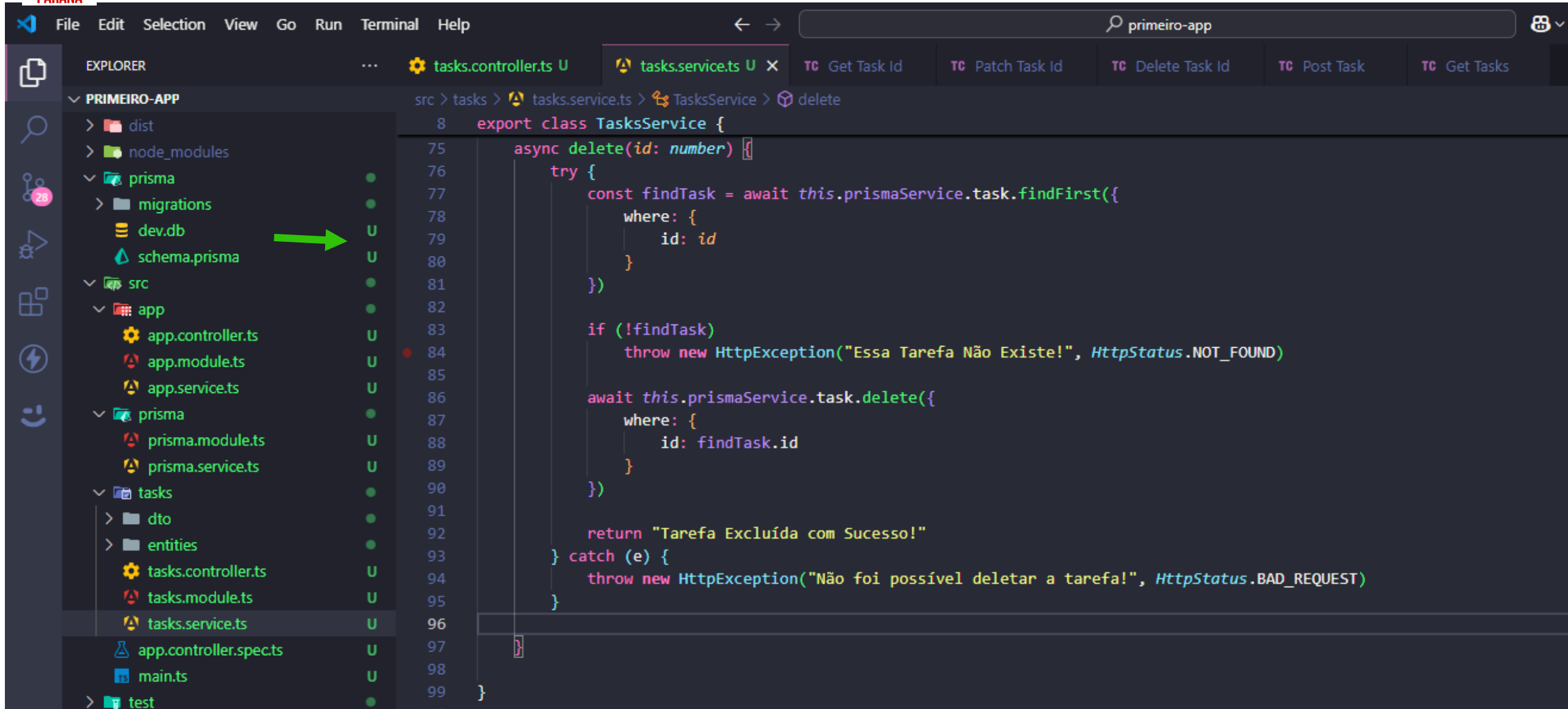


The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays the project structure for 'PRIMEIRO-APP'. The file 'tasks.service.ts' is selected and open in the editor. The breadcrumb navigation at the top indicates the path: 'src > tasks > tasks.service.ts > TasksService > update'. The code in the editor is as follows:

```
8 export class TasksService {
49
50   async update(id: number, updateTaskDto: UpdateTaskDto) {
51     try {
52       const findTask = await this.prismaService.task.findFirst({
53         where: {
54           id: id
55         }
56       })
57
58       if (!findTask)
59         throw new HttpException("Essa Tarefa Não Existe!", HttpStatus.NOT_FOUND)
60
61       const task = await this.prismaService.task.update({
62         where: {
63           id: findTask.id
64         },
65         data: updateTaskDto
66       })
67
68       return task
69
70     } catch (e) {
71       throw new HttpException("Não foi possível atualizar a tarefa!", HttpStatus.BAD_REQUEST)
72     }
73   }
74 }
```

A green arrow points to the 'async' keyword in the 'update' method signature on line 50.

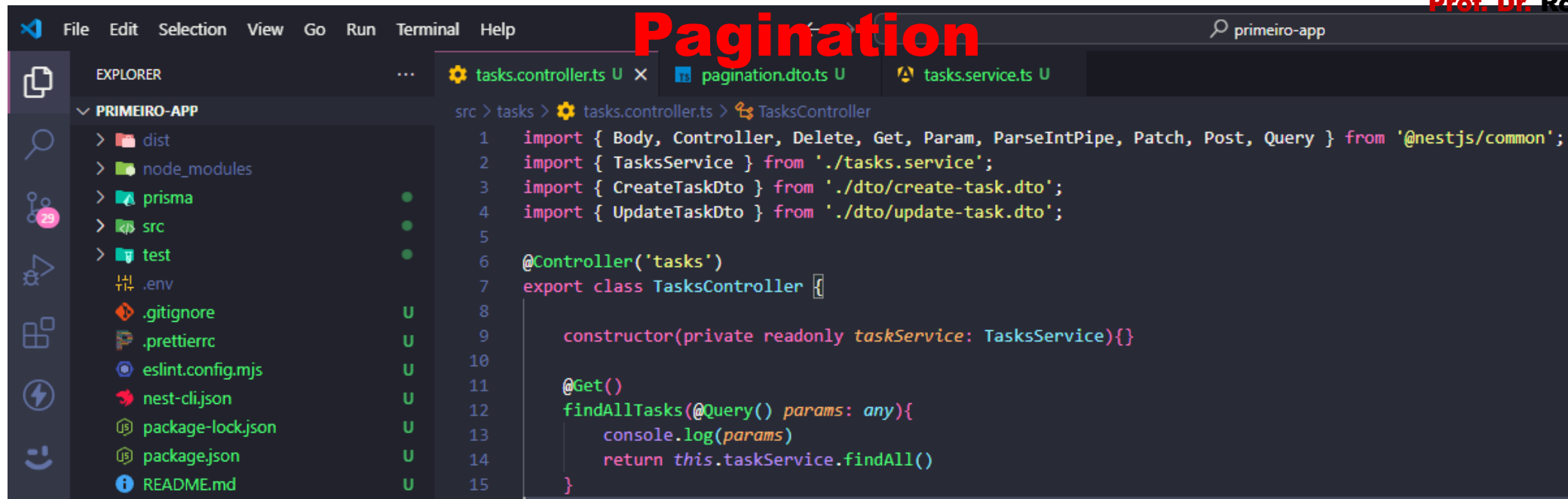
Delete



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the Editor window on the right. The Explorer sidebar displays the file structure of a project named "PRIMEIRO-APP". A green arrow points to the file "tasks.service.ts" under the "tasks" folder. The Editor window shows the code for "tasks.service.ts" with the following content:

```
src > tasks > tasks.service.ts > TasksService > delete
8  export class TasksService {
75  async delete(id: number) {
76      try {
77          const findTask = await this.prismaService.task.findFirst({
78              where: {
79                  id: id
80              }
81          })
82
83          if (!findTask)
84              throw new HttpException("Essa Tarefa Não Existe!", HttpStatus.NOT_FOUND)
85
86          await this.prismaService.task.delete({
87              where: {
88                  id: findTask.id
89              }
90          })
91
92          return "Tarefa Excluída com Sucesso!"
93      } catch (e) {
94          throw new HttpException("Não foi possível deletar a tarefa!", HttpStatus.BAD_REQUEST)
95      }
96  }
97  }
98
99 }
```

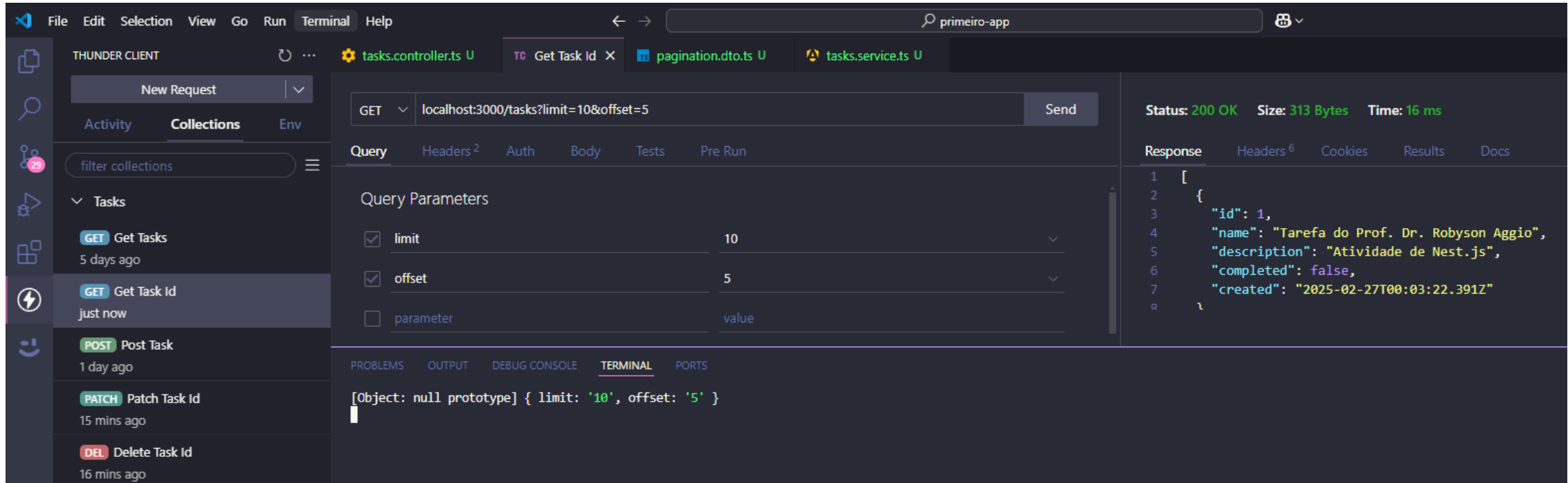
Pagination



```

1  import { Body, Controller, Delete, Get, Param, ParseIntPipe, Patch, Post, Query } from '@nestjs/common';
2  import { TasksService } from '../tasks.service';
3  import { CreateTaskDto } from '../dto/create-task.dto';
4  import { UpdateTaskDto } from '../dto/update-task.dto';
5
6  @Controller('tasks')
7  export class TasksController {
8
9      constructor(private readonly taskService: TasksService){}
10
11     @Get()
12     findAllTasks(@Query() params: any){
13         console.log(params)
14         return this.taskService.findAll()
15     }

```



THUNDER CLIENT

New Request | Activity | Collections | Env

filter collections

Tasks

- GET Get Tasks (5 days ago)
- GET Get Task Id (just now)**
- POST Post Task (1 day ago)
- PATCH Patch Task Id (15 mins ago)
- DEL Delete Task Id (16 mins ago)

GET localhost:3000/tasks?limit=10&offset=5 Send

Query Parameters

Parameter	Value
limit	10
offset	5
parameter	value

Status: 200 OK Size: 313 Bytes Time: 16 ms

Response

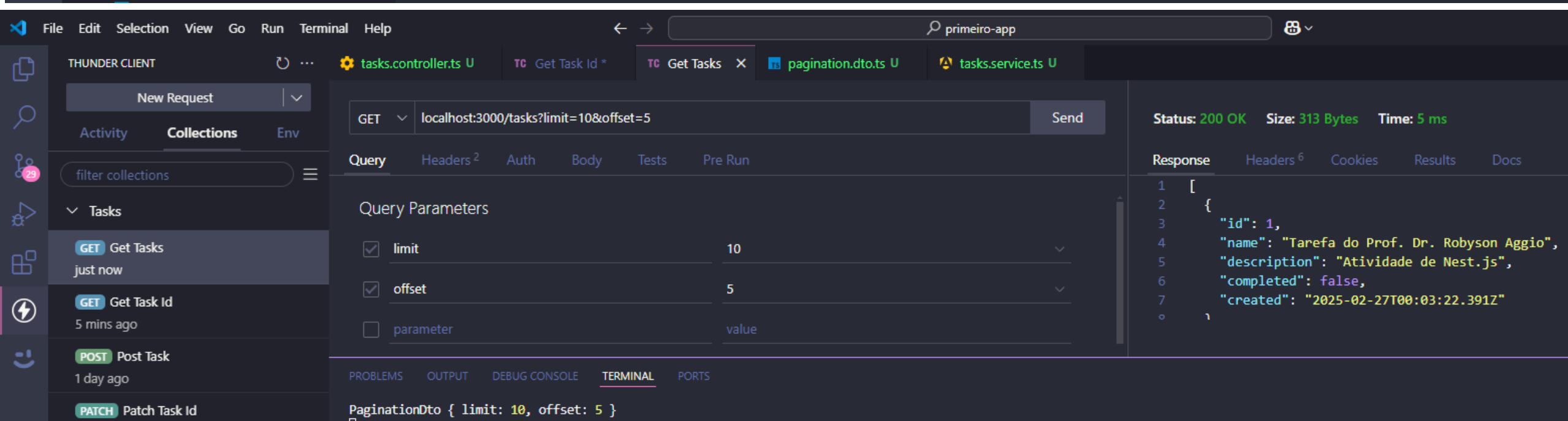
```

1  [
2    {
3      "id": 1,
4      "name": "Tarefa do Prof. Dr. Robyson Aggio",
5      "description": "Atividade de Nest.js",
6      "completed": false,
7      "created": "2025-02-27T00:03:22.391Z"
8    }
9  ]

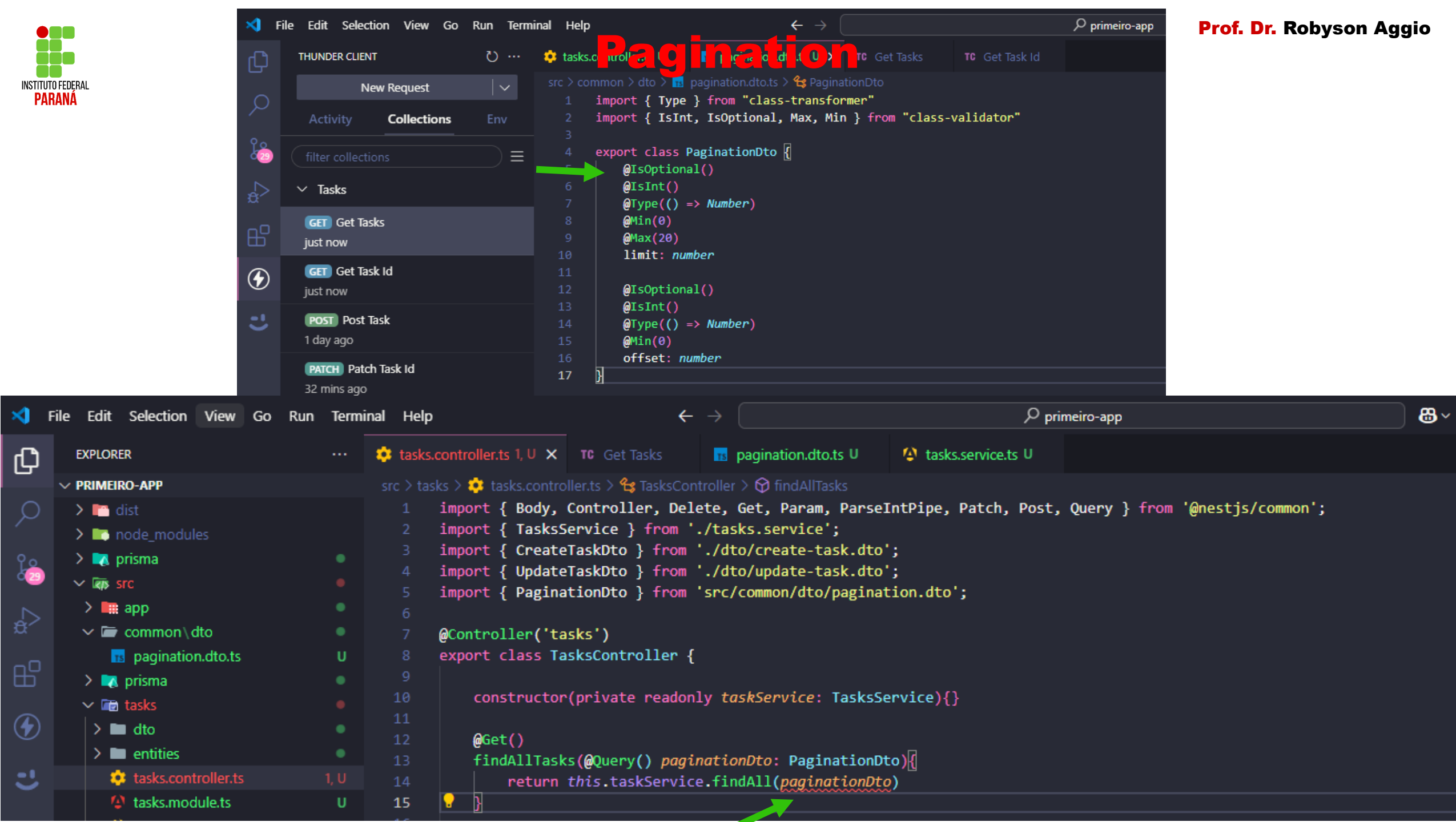
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

[Object: null prototype] { limit: '10', offset: '5' }



Pagination



The image displays a VS Code editor interface with three panels showing code related to pagination in a NestJS application.

Top Panel (Thunder Client): Shows a collection of tasks. The selected task is a **GET** request named **Get Tasks**, which was executed **just now**. Other tasks include **Get Task Id** (GET, just now), **Post Task** (POST, 1 day ago), and **Patch Task Id** (PATCH, 32 mins ago).

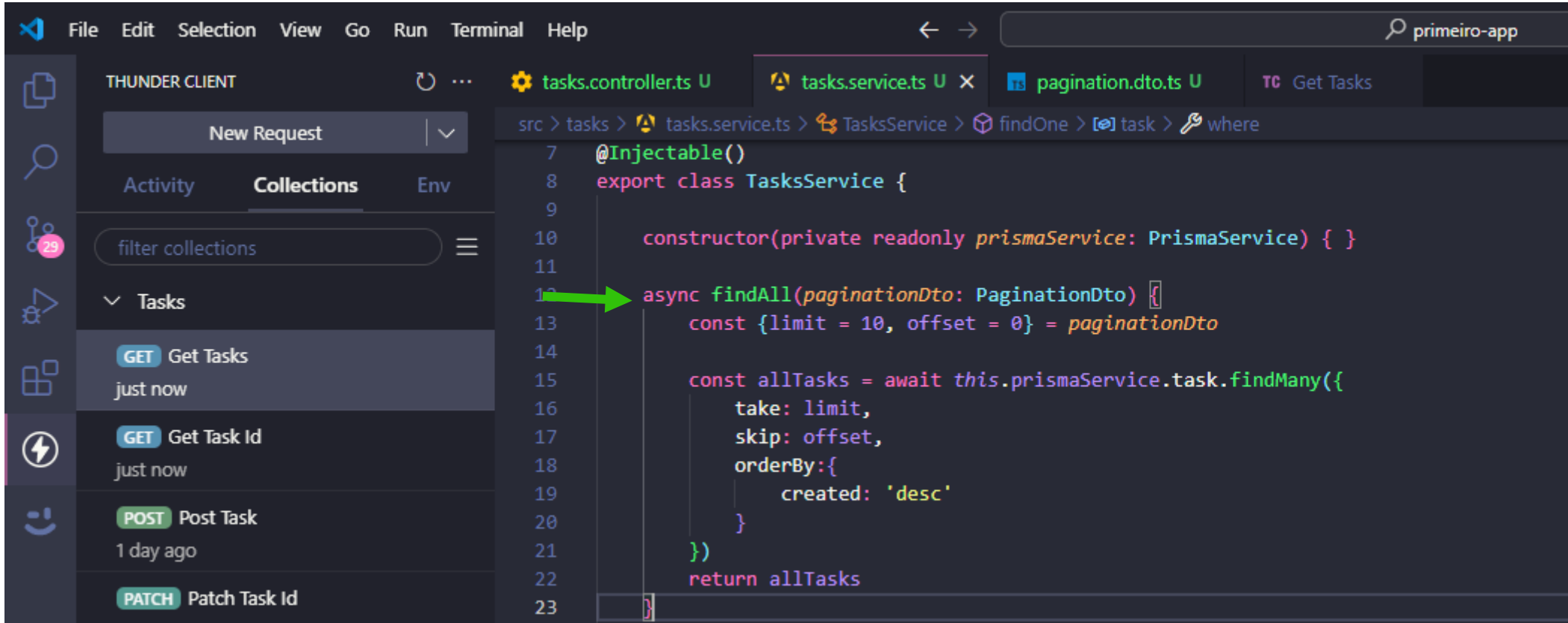
Middle Panel (pagination.dto.ts): Shows the **PaginationDto** class definition. A green arrow points to the **@IsOptional()** decorator on the **limit** property.

```
src > common > dto > pagination.dto.ts > PaginationDto
1 import { Type } from "class-transformer"
2 import { IsInt, IsOptional, Max, Min } from "class-validator"
3
4 export class PaginationDto {
5   @IsOptional()
6   @IsInt()
7   @Type(() => Number)
8   @Min(0)
9   @Max(20)
10  limit: number
11
12  @IsOptional()
13  @IsInt()
14  @Type(() => Number)
15  @Min(0)
16  offset: number
17 }
```

Bottom Panel (tasks.controller.ts): Shows the **TasksController** class. A green arrow points to the **findAllTasks** method, which uses the **PaginationDto** class for query parameters.

```
src > tasks > tasks.controller.ts > TasksController > findAllTasks
1 import { Body, Controller, Delete, Get, Param, ParseIntPipe, Patch, Post, Query } from '@nestjs/common';
2 import { TasksService } from '../tasks.service';
3 import { CreateTaskDto } from '../dto/create-task.dto';
4 import { UpdateTaskDto } from '../dto/update-task.dto';
5 import { PaginationDto } from 'src/common/dto/pagination.dto';
6
7 @Controller('tasks')
8 export class TasksController {
9
10  constructor(private readonly taskService: TasksService){}
11
12  @Get()
13  findAllTasks(@Query() paginationDto: PaginationDto){
14    return this.taskService.findAll(paginationDto)
15  }
```

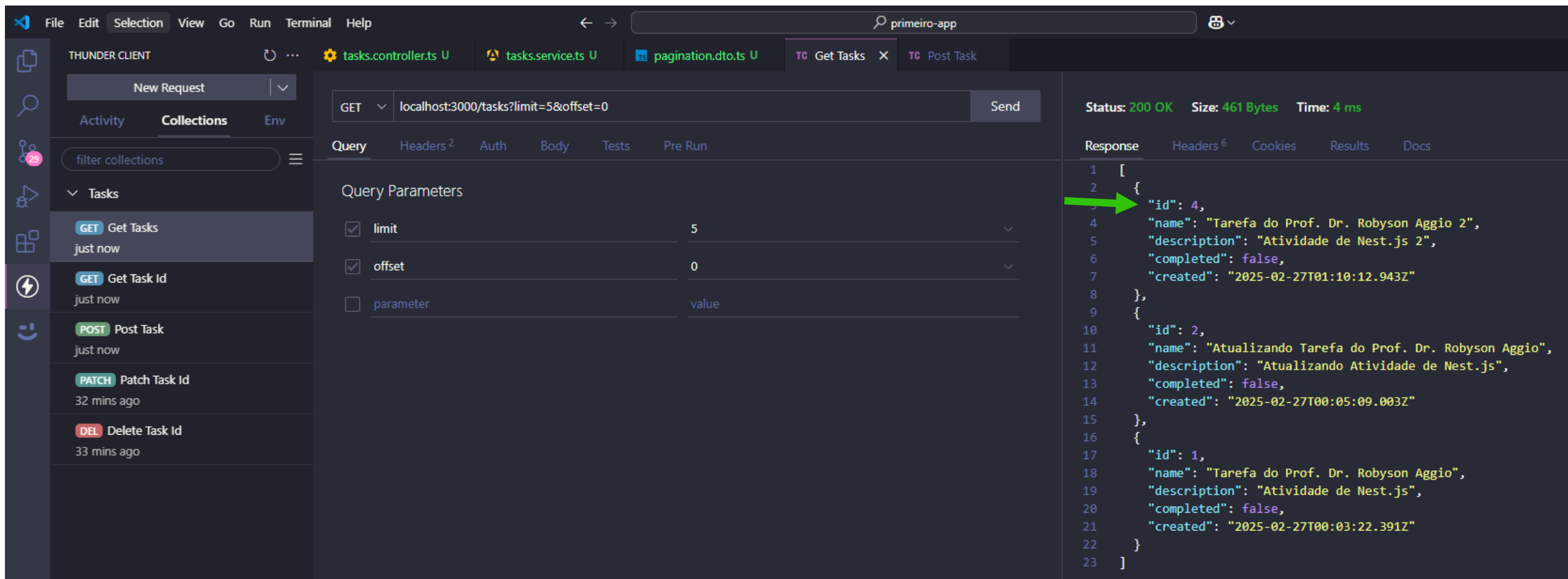
Pagination



The screenshot shows the Visual Studio Code interface with a REST client on the left and a TypeScript service file on the right. The REST client, titled 'THUNDER CLIENT', has a 'Collections' tab selected, showing a collection named 'Tasks'. The 'Tasks' collection contains five requests: 'GET Get Tasks' (just now), 'GET Get Task Id' (just now), 'POST Post Task' (1 day ago), and 'PATCH Patch Task Id'. The 'GET Get Tasks' request is selected. The right pane shows the 'tasks.service.ts' file, which defines the 'TasksService' class. A green arrow points to the 'findAll' method in the service.

```
7 @Injectable()
8 export class TasksService {
9
10     constructor(private readonly prismaService: PrismaService) { }
11
12     async findAll(paginationDto: PaginationDto) {
13         const {limit = 10, offset = 0} = paginationDto
14
15         const allTasks = await this.prismaService.task.findMany({
16             take: limit,
17             skip: offset,
18             orderBy:{
19                 created: 'desc'
20             }
21         })
22         return allTasks
23     }
```

Testar no Thunder ou Postman



The screenshot displays the Thunder Client interface with a REST client request and its response.

Request Details:

- Method: GET
- URL: localhost:3000/tasks?limit=5&offset=0
- Query Parameters:
 - limit: 5
 - offset: 0
 - parameter: value

Response Details:

- Status: 200 OK
- Size: 461 Bytes
- Time: 4 ms

Response Body (JSON):

```
[
  {
    "id": 4,
    "name": "Tarefa do Prof. Dr. Robyson Aggio 2",
    "description": "Atividade de Nest.js 2",
    "completed": false,
    "created": "2025-02-27T01:10:12.943Z"
  },
  {
    "id": 2,
    "name": "Atualizando Tarefa do Prof. Dr. Robyson Aggio",
    "description": "Atualizando Atividade de Nest.js",
    "completed": false,
    "created": "2025-02-27T00:05:09.003Z"
  },
  {
    "id": 1,
    "name": "Tarefa do Prof. Dr. Robyson Aggio",
    "description": "Atividade de Nest.js",
    "completed": false,
    "created": "2025-02-27T00:03:22.391Z"
  }
]
```

Atividade

- 1) Implemente o prisma**
- 2) No arquivo schema.prisma, crie os models:
Guest, Users, Teachers**
- 3) Para cada model do item 2), implemente o prisma
para os verbos (Get, Post, Patch e Delete)**
- 4) Crie a camada: common / dto / pagination.dto.ts**
 - 4.1) crie o pagination.dto.ts**
 - 4.2) Utilize o paginationDto no verbo get, no método findAll()**
- 5) Testar no Thunder Client ou Postman**