NestJS Task Manager API – Phase 1

A backend Proof of Concept built to understand **NestJS fundamentals**, clean architecture, and **PostgreSQL integration using Docker** (without installing Postgres locally).

This repository intentionally pauses at **Phase 1** to focus on understanding core concepts before enabling advanced NestJS features.

🚀 Tech Stack

- Node.js
- NestJS
- TypeScript
- TypeORM
- PostgreSQL (Docker)
- Docker & Docker Compose

🚢 Phase 1 Objectives

- Create a NestJS project using CLI
- Understand NestJS module structure
- Implement Controller → Service → Repository flow
- Integrate PostgreSQL using TypeORM
- Run PostgreSQL via Docker instead of local installation
- Build basic CRUD APIs
- Keep validation concepts understood but **not enabled yet**

Project Structure

```
task-manager/
├── src/
│   ├── app.module.ts
│   ├── main.ts
│   └── tasks/
│       ├── dto/
│       │   └── create-task.dto.ts
│       ├── task.entity.ts
│       ├── tasks.controller.ts
│       ├── tasks.module.ts
│       └── tasks.service.ts
├── docker-compose.yml
├── package.json
└── README.md
```

Prerequisites

Make sure the following are installed:

- Node.js (v18+ recommended)
- npm
- Docker
- Docker Compose
- NestJS CLI

Install NestJS CLI:

```
npm install -g @nestjs/cli
```

Create NestJS Project

```
nest new task-manager
cd task-manager
```

Choose **npm** (or yarn) when prompted.

Install Dependencies

TypeORM + PostgreSQL driver

```
npm install @nestjs/typeorm typeorm pg
```

Validation libraries (used later)

```
npm install class-validator class-transformer
```

> Note: Validation is **not enabled in Phase 1**.
> DTOs exist for learning and will be activated in Phase 2.

Generate Tasks Module

```
nest generate module tasks
nest generate controller tasks
nest generate service tasks
```

This follows the **NestJS modular architecture** pattern.

PostgreSQL Using Docker (No Local Install)

PostgreSQL runs inside Docker.

`docker-compose.yml`

```
version: "3.9"

services:
  postgres:
    image: postgres:16
    container_name: task_postgres
    ports:
      - "5438:5432"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
      POSTGRES_DB: taskdb
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d taskdb"]
      interval: 5s
      timeout: 5s
      retries: 5

volumes:
  postgres_data:
```

Start PostgreSQL

```
docker compose up -d
```

Verify container:

```
docker ps
```

Test database access:

```
docker exec -it task_postgres psql -U postgres -d taskdb
```

TypeORM Configuration

`src/app.module.ts`

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { TasksModule } from './tasks/tasks.module';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: '127.0.0.1',
      port: 5438,
      username: 'postgres',
      password: 'password',
      database: 'taskdb',
      autoLoadEntities: true,
      synchronize: true, // development only
      retryAttempts: 10,
      retryDelay: 3000,
    }),
    TasksModule,
  ],
})
export class AppModule {}
```

Task Entity

`src/tasks/task.entity.ts`

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class Task {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  title: string;

  @Column()
  description: string;

  @Column({ default: false })
```

```
  completed: boolean;
}
```

Tasks Module

`src/tasks/tasks.module.ts`

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { TasksController } from './tasks.controller';
import { TasksService } from './tasks.service';
import { Task } from './task.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Task])],
  controllers: [TasksController],
  providers: [TasksService],
})
export class TasksModule {}
```

Service Layer

`src/tasks/tasks.service.ts`

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Task } from './task.entity';

@Injectable()
export class TasksService {
  constructor(
    @InjectRepository(Task)
    private readonly taskRepository: Repository<Task>,
  ) {}

  findAll(): Promise<Task[]> {
    return this.taskRepository.find();
  }

  async findOne(id: number): Promise<Task> {
    const task = await this.taskRepository.findOne({ where: { id } });
    if (!task) {
      throw new NotFoundException(`Task with id ${id} not found`);
    }
```

```
    return task;
  }

  create(data: Partial<Task>): Promise<Task> {
    const task = this.taskRepository.create(data);
    return this.taskRepository.save(task);
  }

  async remove(id: number): Promise<void> {
    const result = await this.taskRepository.delete(id);
    if (result.affected === 0) {
      throw new NotFoundException(`Task with id ${id} not found`);
    }
  }
}
```

🚚Controller Layer

`src/tasks/tasks.controller.ts`

```
import { Controller, Get, Post, Delete, Param, Body } from '@nestjs/common';
import { TasksService } from './tasks.service';
import { CreateTaskDto } from './dto/create-task.dto';
import { Task } from './task.entity';

@Controller('tasks')
export class TasksController {
  constructor(private readonly tasksService: TasksService) {}

  @Get()
  getAll(): Promise<Task[]> {
    return this.tasksService.findAll();
  }

  @Get(':id')
  getOne(@Param('id') id: number): Promise<Task> {
    return this.tasksService.findOne(id);
  }

  @Post()
  create(@Body() dto: CreateTaskDto): Promise<Task> {
    return this.tasksService.create(dto);
  }

  @Delete(':id')
  delete(@Param('id') id: number): Promise<void> {
    return this.tasksService.remove(id);
```

```
  }
}
```

Run the Application

```
npm run start:dev
```

Expected output:

```
[Nest] Nest application successfully started
```

🚑Available APIs (Phase 1)

Method	Endpoint	Description
POST	`/tasks`	Create task
GET	`/tasks`	List all tasks
GET	`/tasks/:id`	Get task by ID
DELETE	`/tasks/:id`	Delete task

Key Learnings from Phase 1

- NestJS modular project structure
- Controller vs Service responsibility separation
- TypeORM Entity & Repository pattern
- Docker-based PostgreSQL setup
- Real-world DB connectivity debugging
- Clean backend foundation before adding validation

🛩️Phase 2 Completed

Phase 2 focused on making the application **safe, predictable, and production-ready**.

What was added in Phase 2

- **Global Validation Pipes**
- Enabled `ValidationPipe` to validate all incoming requests
- Invalid requests now fail early with proper `400 Bad Request`

- **DTO-driven Validation**

- `CreateTaskDto` enforces required fields using `@IsDefined` and `@IsNotEmpty`
- `UpdateTaskDto` supports partial updates using `@IsOptional`

- **PATCH Endpoint**

- Added `PATCH /tasks/:id` for partial updates
- Correct REST semantics implemented

- **Environment-based Configuration**

- Integrated `ConfigModule`
- Database credentials moved to `.env`
- No secrets hardcoded in source code

- **Updated Tooling & Documentation**

- Phase 2 Postman collection added
- README updated to reflect validation, PATCH, and config changes

---

Key Technical Learnings

- Repository pattern via TypeORM
- Difference between compile-time types and runtime validation
- Proper use of `async` / `await` (only when needed)
- DTOs as API contracts
- Validation as the first line of defense (before DB constraints)

Current Status

- ✅Phase 1 completed (core architecture & infrastructure)
- ✅Phase 2 completed (validation, update APIs, config)
- ⏭️Phase 3 will focus on:
- Dockerizing the NestJS API
- CI/CD pipeline using GitHub Actions
- Production deployment strategy

Author

Santosh Kumar
Backend / Blockchain Engineer
Exploring NestJS with enterprise-grade practices

🚀Phase 3 — Containers & CI/CD (Up to CI)

Phase 3 focuses on making the application **deployable and verifiable in an automated way**. Up to this stage, the goal is **containerization and continuous integration**, not deployment.

Step 3.1 — Dockerize the NestJS API

The NestJS application is packaged as a **Docker image** using a **multi-stage Dockerfile**.

Why multi-stage build? - Keeps the final image small - Separates build-time and runtime dependencies - Matches real production practices

What the Dockerfile does: - Builds the NestJS app (`npm run build`) - Copies only compiled output and production dependencies - Starts the app using `node dist/main.js`

This ensures the API can run consistently across environments.

Step 3.2 — Docker Compose (API + PostgreSQL)

Running the API container alone is not sufficient, because containers cannot access other services via `localhost`.

Docker Compose is used to: - Run **API and PostgreSQL together** - Attach both containers to the same Docker network - Allow the API to connect to Postgres using the service name `postgres`

Key concepts learned: - Containers do not use `localhost` to talk to each other - Docker service names act as DNS hostnames - Environment variables must be reloaded by recreating containers

The application now runs with a single command:

```
docker compose up -d --build
```

Step 3.3 — Continuous Integration (GitHub Actions)

A **GitHub Actions CI pipeline** is added to automatically verify the application on every push or pull request to `main`.

The CI pipeline performs the following steps:

1. Checkout repository
2. Setup Node.js (v18)
3. Install dependencies using `npm ci`
4. Build the NestJS application

5. Build the Docker image

If any step fails, the pipeline fails — preventing broken builds from reaching `main`.

CI workflow location:

```
.github/workflows/ci.yml
```

This ensures: - Reproducible builds - Early failure detection - CI parity with production Docker builds

---

🌆Current Project Status

- ✅Phase 1: Core architecture & database setup
- ✅Phase 2: Validation, PATCH APIs & configuration
- ✅Phase 3 (partial): Dockerization & CI pipeline
- ⏭️Phase 3 (next): Deployment strategy & release flow

Key Takeaways So Far

- Clean modular NestJS architecture
- Repository pattern using TypeORM
- DTO-based runtime validation
- Environment-driven configuration
- Production-grade Docker images
- Container networking fundamentals
- Automated CI verification with GitHub Actions

At this stage, the application is **buildable, containerized, and continuously verified**, which is the foundation required for safe production deployments.

🏢Phase 5 — Enterprise Hardening

Phase 5 focuses on **operability and observability**, which are critical for running backend services in production.

💈 Health Checks

A health endpoint is exposed to allow platforms and operators to verify application readiness.

Endpoint:

```
GET /health
```

What it checks: - Application process is running - Database connectivity via TypeORM

Sample healthy response:

```
{
  "status": "ok",
  "info": {
    "database": {
      "status": "up"
    }
  }
}
```

This endpoint is used by container platforms, load balancers, and monitoring systems.

💈 Structured Logging

The application uses structured, JSON-based logging suitable for centralized log systems.

Logging characteristics: - Timestamped log entries - Log levels (log, warn, error) - Context-aware messages

Structured logs make it easier to: - Debug production issues - Correlate events - Feed logs into systems like ELK or OpenSearch

---

💈 Metrics

Basic application metrics are exposed for monitoring and alerting.

Metrics endpoint:

```
GET /metrics
```

Metrics are exposed in **Prometheus-compatible format**, enabling: - Performance monitoring - Alerting - Capacity planning

This prepares the application for integration with monitoring stacks such as Prometheus + Grafana.

🏙️Project Maturity Status

- ✅Modular NestJS architecture
- ✅Dockerized API and database
- ✅CI pipeline (build + image verification)
- ✅Safe DB migrations
- ✅Health checks
- ✅Structured logging
- ✅Metrics exposure

At this stage, the project demonstrates **enterprise-grade backend engineering practices**.