# Task Management Application - Backend Documentation

## Table of Contents

## 1. Introduction

This documentation outlines the backend implementation of a Task Management Web Application. The appl

## 2. Architecture Overview

### 2.1 Clean Architecture Approach

Our application follows the Clean Architecture approach, emphasizing separation of concerns and depender

1. Domain Layer
2. Application Layer
3. Infrastructure Layer
4. Interface Layer

Each layer has a specific responsibility and depends only on the layers beneath it, ensuring a loosely couple

### 2.2 Design Practices

We've incorporated several design practices to enhance the robustness and flexibility of our application:

1. **Dependency Injection**: Utilizing TypeDI for managing dependencies.
2. **Repository Pattern**: Abstracting data access operations.
3. **Factory Pattern**: Creating complex objects consistently.
4. **Middleware Pattern**: Handling cross-cutting concerns.
5. **SOLID Principles**: Adhering to principles for better object-oriented design.

## 3. Technology Stack

■
- **Runtime Environment**: Node.js■
- **Web Framework**: Express.js■
- **Database**: MongoDB■
- **ODM (Object Document Mapper)**: Mongoose■
- **Authentication**: JSON Web Tokens (JWT)■
- **API Documentation**: Swagger■
- **Testing**: Jest■
- **Logging**: Winston■
- **Dependency Injection**: TypeDI■
■
## 4. Project Structure■
■
```■
src■
■■■ application■
■   ■■■ services■
■   ■   ■■■ AuthenticationService■
■   ■■■ use-cases■
■■■ domain■
■   ■■■ task■
■       ■■■ Task.ts■
■       ■■■ TaskFactory.ts■
■       ■■■ TaskRepository.ts■
■       ■■■ TaskService.ts■
■■■ config■
■   ■■■ swagger.ts■
■   ■■■ app.ts■
■   ■■■ db.ts■
■■■ infrastructure■
■   ■■■ persistence■
■       ■■■ mongodb■
■           ■■■ models■
■           ■   ■■■ TaskModel.ts■
■           ■■■ repositories■
■           ■   ■■■ MongodbTaskRepository.ts■
■           ■■■ MongoDbPersistenceConnection.ts■
■■■ interfaces■
■   ■■■ http■
■       ■■■ controllers■
■       ■■■ middlewares■
■       ■   ■■■ authMiddleware.ts■
■       ■   ■■■ loggingMiddleware.ts■
■       ■   ■■■ errorMiddleware.ts■
■       ■   ■■■ validationMiddleware.ts■
■       ■■■ routes■
■       ■   ■■■ taskRoutes.ts■
■       ■■■ validations■
■           ■■■ taskValidations.ts■
■■■ utility■
■■■ tests■
■   ■■■ unit■
■       ■■■ application■
■       ■■■ domain■
■       ■■■ config■
■       ■■■ infrastructure■
■       ■■■ interfaces■
■       ■■■ utility■
■■■ server.ts■
■■■ container.ts■
```

[Insert screenshot of folder structure here]

## 5. Core Components

### 5.1 Domain Layer

The domain layer contains all entities, value objects, and domain services. It's independent of any external d

Example of a domain entity:

```typescript
// src/domain/task/Task.ts
export class Task {
  constructor(
    public id: string,
    public title: string,
    public description: string,
    public status: 'TODO' | 'IN_PROGRESS' | 'DONE',
    public dueDate: Date,
    public userId: string
  ) {}
}
```

### 5.2 Application Layer

The application layer contains application logic and use cases. It depends on the domain layer but is indepen

Example of a use case:

```typescript
// src/application/use-cases/CreateTaskUseCase.ts
import { Task } from '../../domain/task/Task';
import { TaskRepository } from '../../domain/task/TaskRepository';

export class CreateTaskUseCase {
  constructor(private taskRepository: TaskRepository) {}

  async execute(taskData: Omit<Task, 'id'>): Promise<Task> {
    const task = new Task(
      Date.now().toString(), // simple ID generation
      taskData.title,
      taskData.description,
      taskData.status,
      taskData.dueDate,
      taskData.userId
    );
    return this.taskRepository.create(task);
  }
}
```

### 5.3 Infrastructure Layer

The infrastructure layer contains implementations of interfaces defined in the domain layer, such as reposito

Example of a repository implementation:

```typescript
// src/infrastructure/persistence/mongodb/repositories/MongodbTaskRepository.ts
import { TaskRepository } from '../../../../domain/task/TaskRepository';
import { Task } from '../../../../domain/task/Task';
import { TaskModel } from '../models/TaskModel';

export class MongodbTaskRepository implements TaskRepository {
  async create(task: Task): Promise<Task> {
    const createdTask = await TaskModel.create(task);
    return this.modelToDomain(createdTask);
  }

  private modelToDomain(model: any): Task {
    return new Task(
      model._id.toString(),
      model.title,
      model.description,
      model.status,
      model.dueDate,
      model.userId.toString()
    );
  }
  
  // Other repository methods...
}
```

### 5.4 Interface Layer

The interface layer contains controllers, routes, and other components that interact with external agents.

Example of a controller:

```typescript
// src/interfaces/http/controllers/TaskController.ts
import { Request, Response } from 'express';
import { CreateTaskUseCase } from '../../../application/use-cases/CreateTaskUseCase';

export class TaskController {
  constructor(private createTaskUseCase: CreateTaskUseCase) {}

  async createTask(req: Request, res: Response) {
    try {
      const task = await this.createTaskUseCase.execute(req.body);
      res.status(201).json(task);
    } catch (error) {
      res.status(400).json({ error: error.message });
    }
  }
  
  // Other controller methods...
}
```

## 6. Database Design

We use MongoDB as our database, with Mongoose as the ODM. Here's an example of our Task schema:

```typescript
// src/infrastructure/persistence/mongodb/models/TaskModel.ts
```

```
import mongoose from 'mongoose';

const taskSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String },
  status: { type: String, enum: ['TODO', 'IN_PROGRESS', 'DONE'], default: 'TODO' },
  dueDate: { type: Date },
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
}, { timestamps: true });

export const TaskModel = mongoose.model('Task', taskSchema);
```

## 7. API Design

### 7.1 RESTful Endpoints

Our application exposes the following RESTful API endpoints:

- `POST /api/tasks`: Create a new task
- `GET /api/tasks`: Retrieve all tasks
- `GET /api/tasks/:id`: Retrieve a specific task
- `PUT /api/tasks/:id`: Update a task
- `DELETE /api/tasks/:id`: Delete a task

Example of task routes implementation:

```typescript
// src/interfaces/http/routes/taskRoutes.ts

import express from 'express';
import { container } from 'typedi';
import { TaskController } from '../controllers/TaskController';
import { authMiddleware } from '../middlewares/authMiddleware';
import { validationMiddleware } from '../middlewares/validationMiddleware';
import { createTaskSchema, updateTaskSchema } from '../validations/taskValidations';

const router = express.Router();
const taskController = container.get(TaskController);

router.post('/', authMiddleware, validationMiddleware(createTaskSchema), taskController.createTask);
router.get('/', authMiddleware, taskController.getAllTasks);
router.get('/:id', authMiddleware, taskController.getTaskById);
router.put('/:id', authMiddleware, validationMiddleware(updateTaskSchema), taskController.updateTask);
router.delete('/:id', authMiddleware, taskController.deleteTask);

export default router;
```

### 7.2 Swagger Documentation

We've implemented Swagger for clear and interactive API documentation. Below is a screenshot of the Swa

[Insert screenshot of Swagger UI here]

This Swagger documentation provides a comprehensive overview of all API endpoints, request/response sc

## 8. Authentication and Authorization

We use JSON Web Tokens (JWT) for authentication. The `authMiddleware` verifies the token before allowin

```typescript
// src/interfaces/http/middlewares/authMiddleware.ts

import { Request, Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { config } from '../../../config/app';

export const authMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    const decoded = jwt.verify(token, config.jwtSecret);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
  }
};
```

## 9. Error Handling and Logging

We've implemented centralized error handling and logging to ensure consistent error responses and compre

Error Handling Middleware:

```typescript
// src/interfaces/http/middlewares/errorMiddleware.ts

import { Request, Response, NextFunction } from 'express';
import { logger } from '../../../utility/logger';

export const errorMiddleware = (err: Error, req: Request, res: Response, next: NextFunction) => {
  logger.error(err.stack);

  res.status(500).json({
    error: 'Internal Server Error',
    message: err.message
  });
};
```

Logging Configuration:

```typescript
// src/utility/logger.ts

import winston from 'winston';

export const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
  ],
});
```

## 10. Testing Strategy

We use Jest for unit and integration testing. Here's an example of a test for the TaskService:

```typescript
// src/tests/unit/domain/TaskService.test.ts

import { Container } from 'typedi';
import { TaskService } from '../../../domain/task/TaskService';
import { TaskRepository } from '../../../domain/task/TaskRepository';

describe('TaskService', () => {
  let taskService: TaskService;
  let mockTaskRepository: jest.Mocked<TaskRepository>;

  beforeEach(() => {
    mockTaskRepository = {
      create: jest.fn(),
      findById: jest.fn(),
      findAll: jest.fn(),
      update: jest.fn(),
      delete: jest.fn(),
    };

    Container.set(TaskRepository, mockTaskRepository);
    taskService = Container.get(TaskService);
  });

  it('should create a task', async () => {
    const taskData = { title: 'Test Task', description: 'Test Description' };
    mockTaskRepository.create.mockResolvedValue({ id: '1', ...taskData });

    const result = await taskService.createTask(taskData);

    expect(result).toEqual({ id: '1', ...taskData });
    expect(mockTaskRepository.create).toHaveBeenCalledWith(taskData);
  });

  // Add more tests for other methods...
});
```

## 11. Code Quality and Documentation

### 11.1 Coding Standards

We adhere to TypeScript best practices and use ESLint for code linting to ensure consistent code style acros

### 11.2 Comprehensive Docstrings

We use comprehensive docstrings throughout our codebase to improve readability, maintainability, and exte

```typescript
/**
 * Creates a new task.
 * 
 * This use case handles the creation of a new task in the system. It validates
 * the input, creates a Task entity, and persists it using the task repository.
 * 
 * @param {CreateTaskDTO} taskData - The data for creating a new task.
 * @returns {Promise<Task>} The created task.
 * @throws {ValidationError} If the task data is invalid.
 * @throws {DatabaseError} If there's an error persisting the task.
 * 
 * @example
 * const createTaskUseCase = new CreateTaskUseCase(taskRepository);
 * const newTask = await createTaskUseCase.execute({
 *   title: 'Complete project',
 *   description: 'Finish the task management project',
 *   status: 'TODO',
 *   dueDate: new Date('2023-12-31'),
 *   userId: '123456'
 * });
 */
async execute(taskData: CreateTaskDTO): Promise<Task> {
  // Implementation...
}
```

## 12. Deployment

Our application can be deployed to various cloud platforms. Here's a basic example using Docker:

```dockerfile
# Dockerfile

FROM node:14

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]
```

## 13. Performance Considerations

- Implement caching for frequently accessed data using Redis.
- Use pagination for API endpoints that return large datasets.
- Implement database indexing for frequently queried fields.

## 14. Security Measures

- Use HTTPS for all communications.
- Implement rate limiting to prevent abuse.

- Sanitize user inputs to prevent injection attacks.
- Keep dependencies up-to-date to avoid known vulnerabilities.

[Previous content remains unchanged...]

## 15. Scalability Approach

Our application is designed with scalability in mind, allowing it to handle increased load and grow with user d

1. **Horizontal Scaling**: Our stateless application design allows for easy deployment across multiple server

2. **Database Scaling**:
   - We use MongoDB, which supports horizontal scaling through sharding.
   - Implement database connection pooling to efficiently manage database connections.

3. **Caching Strategy**:
   - Implement Redis for caching frequently accessed data, reducing database load.
   - Use cache invalidation strategies to ensure data consistency.

4. **Asynchronous Processing**:
   - Utilize message queues (e.g., RabbitMQ) for handling background tasks and long-running processes.
   - This approach helps in maintaining responsiveness under high load.

5. **Microservices Architecture**:
   - While our current implementation is monolithic, the clean architecture allows for easy transition to micros

6. **Content Delivery Network (CDN)**:
   - Implement a CDN for serving static assets, reducing load on the application servers.

## 16. Setup and Installation

To set up and run the application locally, follow these steps:

1. Clone the repository:
   ```
   git clone https://github.com/your-repo/task-management-app.git
   cd task-management-app
   ```

2. Install dependencies:
   ```
   npm install
   ```

3. Set up environment variables:
   Create a `.env` file in the root directory with the following content:
   ```
   PORT=3000
   MONGODB_URI=mongodb://localhost:27017/task_manager
   JWT_SECRET=your_jwt_secret_here
   NODE_ENV=development
   ```

4. Start the MongoDB service on your local machine.

5. Run the application:
   ```
   npm run start
   ```

6. For development with hot-reloading:
   ```
   npm run dev
   ```

7. Run tests:
   ```
   npm test
   ```

8. Access the API documentation:
   Open a web browser and navigate to `http://localhost:3000/api-docs` to view the Swagger documentation.

## 17. Conclusion

This Task Management Application backend demonstrates a robust, scalable, and maintainable architecture

1. **Clean Architecture**: Ensuring separation of concerns and making the system highly maintainable and a

2. **RESTful API Design**: Providing a clear and intuitive interface for client applications to interact with our

3. **Comprehensive Documentation**: Both in-code (through docstrings) and external (Swagger), facilitating

4. **Security Measures**: Implementing authentication, authorization, and other security best practices to pro

5. **Scalability Considerations**: Designing the system to handle growth in users and data volume efficiently

6. **Testing Strategy**: Ensuring reliability and ease of refactoring through comprehensive unit and integratio

7. **Performance Optimization**: Implementing caching and database optimization techniques to ensure swi

8. **Code Quality**: Adhering to best practices and coding standards to maintain high code quality and reada

This implementation not only meets the current requirements for task management but also provides a solid

The use of Node.js, Express, and MongoDB, combined with modern JavaScript/TypeScript practices, ensure