Task Management API - Technical Documentation

July 11th, 2025

Created by: Adriana Paiva
Contact: +351 933-498-035
Email: itsadrianapaiva@gmail.com
Github: https://github.com/itsadrianapaiva

Overview

This document outlines the requirements and specifications for developing a Task Management API system. The application is designed to manage maintenance tasks performed during working days, supporting two user roles: **Manager** and **Technician**.

Core Functionality

- **Technicians** can view, create, and update their own tasks
- **Managers** can view all tasks, delete tasks, and receive notifications when tasks are performed
- Each task includes a summary (max 2,500 characters) and performance date
- Task summaries may contain personal information requiring appropriate handling

Requirements

MVP Features (Minimum Viable Product)

1. **Task Creation Endpoint** - API endpoint to save new tasks
2. **Task Listing Endpoint** - API endpoint to retrieve tasks based on user role
3. **Manager Notifications** - Notify managers when technicians perform tasks
    - Notification format: "The tech X performed the task Y on date Z"
    - Must not block HTTP requests (asynchronous processing)

Bonus Features

1. **Message Broker Integration** - Implement message broker to decouple notification logic
2. **Kubernetes Deployment** - Create Kubernetes object files for application deployment

Technical Requirements

- **Programming Language**: Node.js or Go
- **Database**: MySQL for data persistence
- **Development Environment**: Docker containerized local development
- **Testing**: Unit tests for all features
- **Container Support**: Docker Compose setup including service and MySQL database

System Architecture

Components

API Server (Node.js)

- Handles all HTTP requests
- Stateless design
- JWT token management via cookies
- MySQL database connectivity

Database Layer (MySQL)

- Persistent data storage for users and tasks
- Relational integrity and constraint management
- Performance optimization through indexing

Message Queue (Redis + BullMQ Queue)

- Background job processing for non-blocking notifications
- Asynchronous task notification dispatch

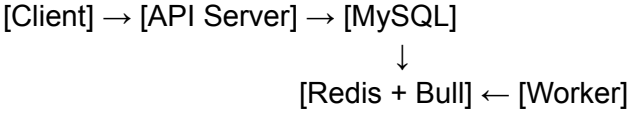
Notification Worker

- Consumes jobs from Bull queue
- Outputs notifications to stdout (simulates delivery)

Client Layer

- HTTP request handling (Postman, browser)
- Automatic cookie management

Data Flow



1. Client sends request (e.g., create task)

2. API validates JWT, processes request, persists to MySQL
3. Notification job queued to Redis
4. Worker consumes Redis job, processes notification

Database Schema

Users Table

```
CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   email VARCHAR(100) NOT NULL UNIQUE,
   passwordHash VARCHAR(255) NOT NULL,
   role ENUM('manager', 'technician') NOT NULL,
   createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
   updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);
```

Tasks Table

```
CREATE TABLE tasks (
   id INT AUTO_INCREMENT PRIMARY KEY,
   summary TEXT NOT NULL,
   performedAt DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
   technicianId INT NOT NULL,
   createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
   updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
   CONSTRAINT fk_technician FOREIGN KEY (technicianId)
      REFERENCES users(id) ON DELETE CASCADE,
   CONSTRAINT chk_summary_length CHECK (CHAR_LENGTH(summary) <= 2500),
   INDEX idx_technician (technicianId),
   INDEX idx_performedAt (performedAt)
);
```

API Specification

Authentication Endpoints

POST /signup

Description: Register a new user (Manager or Technician)

Request Body:

```
{
   "name": "Jane",
   "email": "jane.tech@example.com",
   "password": "password123",
   "role": "technician"
}
```

Response (201):

```
{
   "id": 1,
   "name": "Jane",
   "email": "jane.tech@example.com",
   "role": "technician",
   "message": "Signup successful"

}
```

POST /login

Description: Authenticate user and receive JWT in cookie

Request Body:

```
{
   "email": "jane.tech@example.com",
   "password": "password123"
}
```

Response (200):

Set-Cookie: jwt=xxx; HttpOnly; Secure; SameSite=Strict

```
{
   "message": "Login in successful"
}
```

Task Management Endpoints

POST /tasks

Access: Technician only
Description: Create a new task

Request Body:

```
{
   "summary": "Repaired the X-ray motor component.",
   "performedAt": "2025-07-11 14:00:00"
}
```

Response (201):

```
{
   "taskId": 12,
   "technicianId": 3,
   "summary": "Repaired the X-ray motor component.",
   "performedAt": "2025-07-11 14:00:00",
   "message": "Task registered successfully"
}
```

GET /tasks

Access: Manager (all tasks) / Technician (own tasks only)
Description: Retrieve tasks based on user role

Response (200):

```
[
   {
      "taskId": 12,
      "technicianId": 3,
      "summary": "Repaired the X-ray motor component.",
      "performedAt": "2025-07-11 14:00:00"
   }
]
```

PATCH /tasks/:id

Access: Technician only (own tasks)
Description: Update task information

Request Body:

```
{
   "summary": "Updated details of the performed repair"
}
```

Response (200):

```
{
   "taskId": 12,
   "summary": "Updated details of the performed repair",
   "message": "Task updated successfully"
}
```

DELETE /tasks/:id

Access: Manager only
Description: Delete task by ID

Response (200):

```
{
   "taskId": 12,
   "message": "Task deleted successfully"
}
```

Technology Stack

Core Technologies

- **Runtime**: Node.js + Express
- **Database**: MySQL with mysql2 driver
- **Queue System**: BullMQ with Redis
- **Authentication**: JWT with bcryptjs
- **Testing**: Jest framework
- **Containerization**: Docker + Docker Compose

Project Structure

backend/

```
├── src/
│   ├── config/          # Environment variables and database configuration
│   ├── controllers/     # Request handlers
│   ├── services/        # Business logic
│   ├── db/              # Database connection and migrations
│   ├── routes/          # Express route definitions
│   ├── middlewares/     # JWT, role checks, error handling
│   ├── queues/          # Bull + Redis queue logic
│   ├── workers/         # Notification consumers
│   ├── tests/           # Unit tests
│   ├── utils/           # Utility functions
│   ├── app.js           # Express application
│   └── server.js        # Application entry point
├── migrations/          # SQL migration files
├── docs/          # Documentation files
├── k8s/         # Kubernetes manifest
├── Dockerfile           # Container configuration
├── docker-compose.yml  # Multi-service setup
├── .gitpod.yml          # Gitpod setup
├── .gitpod.Dockerfile  # Gitpod configuration
├── .env.example          # Environment variables placeholder
├── .gitignore
└── package.json
```

Security Considerations

Authentication & Authorization

- JWT tokens stored in HTTP-only cookies
- Role-based access control middleware
- Password hashing with bcryptjs
- Input validation for all endpoints

Security Headers

- Helmet.js for security headers
- CORS configuration
- Rate limiting implementation
- SQL injection prevention through parameterized queries

Development Process & Decision Log

Phase 1: Architecture & Foundation

Initial Planning & Assumptions

Before implementation, key questions were addressed with documented assumptions:

Authentication Strategy

- **Decision**: Implement JWT-based authentication with register/login endpoints
- **Trade-off**: JWT scales well without session storage, reduces XSS risk with proper cookie configuration
- **Implementation**: HTTP-only, secure, SameSite=Strict cookies with short expiration (15min)
- **Alternative considered**: Rotating refresh tokens in Redis for enhanced security

Notification System

- **Decision**: Redis + BullMQ ecosystem for MVP notifications
- **Trade-off**: Simple implementation and good performance vs. potential data loss
- **Scalability consideration**: RabbitMQ or Kafka would be better for production scale
- **Architecture**: `[HTTP Request] → [Task Service] → [Message Broker] → [Notification Consumer] → [Print]`

Database Schema Decisions

Primary Key Strategy

- **Choice**: Auto-increment integers vs. UUIDs
- **Decision**: Integers for performance and space efficiency
- **Trade-off**: UUIDs are globally unique but slower for joins/indexing

Data Types & Constraints

- **TEXT vs VARCHAR**: TEXT for summaries (65k chars) vs VARCHAR(2500)
- **ENUM vs Lookup Table**: ENUM for roles (lightweight, faster) vs. table (more portable)
- **Cascade Deletion**: ON DELETE CASCADE for data integrity vs. SET NULL for preservation

Indexing Strategy

INDEX idx_technician (technicianId),
INDEX idx_performedAt (performedAt)

- **Trade-off**: Improved query performance vs. increased storage and slower writes
- **Justification**: Critical for read-heavy operations (manager listing all tasks)

Docker Configuration Challenges

Environment Limitations

- **Challenge**: Docker Desktop unavailable on development machine
- **Solution**: GitPod with cloud MySQL for Docker compliance
- **Backup Plan**: Local Homebrew MySQL for development, Docker config validation with `docker compose config`

Phase 2: Implementation & Testing

Authentication Implementation

Password Security

- **Library**: bcryptjs with cost factor 12
- **Trade-off**: Security vs. performance (12 provides good balance)
- **Implementation**: Never store raw passwords, always hash before database insertion

JWT Strategy

```
const payload = { id: user.id, role: user.role };
```

- **Minimal payload**: Only essential data to reduce token size
- **Security**: Sensitive data queryable from database using token payload

Testing Framework Decisions

Jest + Supertest Combination

- **Challenge**: ES modules vs. CommonJS compatibility
- **Solution**: Babel for transpilation
- **Trade-off**: 30MB additional dependency vs. reliable test environment
- **Justification**: More time for feature development vs. debugging module systems

Database Integration

Connection Pool vs. Single Connection

```
const pool = mysql.createPool({
  connectionLimit: 10,
  queueLimit: 10,
  waitForConnections: true
});
```

- **Decision**: Connection pooling for concurrency
- **Benefits**: Reusable connections, better performance under load
- **Configuration**: 10 connections for small app, prevents memory leaks

Raw SQL vs. ORM

- **Choice**: Raw SQL with parameterized queries
- **Benefits**: Full control, performance tuning, zero abstraction
- **Trade-off**: More code vs. type safety and development speed
- **Security**: Mandatory parameterized queries to prevent SQL injection

Validation Strategy

Generic vs. Route-Specific Validation

```
const validateBody = (requiredFields) => (req, res, next) => {
  // Generic validation logic
};
```

- **Decision**: Generic `validateBody` middleware for MVP
- **Benefits**: DRY, reusable, testable
- **Trade-off**: Less specific validation vs. faster development
- **Future**: Schema-based validation (Zod/express-validator) for complex rules

Phase 3: Queue System & Debugging

Message Queue Implementation

Bull vs. BullMQ

- **Initial choice**: Bull for simplicity
- **Final decision**: BullMQ for modern API and future-proofing
- **Trade-off**: More boilerplate vs. better maintainability
- **Dependency**: Explicit ioredis installation for version control

Redis Configuration

```
const redis = new Redis({
  host: process.env.REDIS_HOST,
  port: process.env.REDIS_PORT,
  maxRetriesPerRequest: null // Prevents hanging
});
```

- **Critical setting**: `maxRetriesPerRequest: null` to prevent infinite hanging
- **Connection strategy**: Shared connection for consistency

Worker Process Architecture

Separate Worker Process

node src/workers/notification.worker.js

- **Decision**: Dedicated worker process vs. inline processing
- **Benefits**: Scalability, separation of concerns, fault isolation
- **Trade-off**: Additional process management vs. simpler single-process setup

Concurrency & Rate Limiting

```
const worker = new Worker('manager_notifications', processJob, {
  connection: redis,
  concurrency: 5,
  limiter: { max: 100, duration: 60000 }
});
```

- **Concurrency**: 5 jobs simultaneously for balanced performance
- **Rate limiting**: 100 jobs/minute to prevent Redis flooding

Critical Debugging Session

POST /tasks Hanging Issue

- **Problem**: Requests hanging indefinitely during task creation
- **Root cause**: Redis connection mismatch and unstarted worker
- **Debugging steps**:
    1. Added `connection.ping()` to verify Redis connectivity
    2. Fixed worker import to use shared Redis connection
    3. Ensured worker process was running
    4. Verified Docker networking (task_system_redis:6379)
- **Solution**: Proper connection sharing and worker lifecycle management
- **Outcome**: Requests complete successfully, worker processes jobs

Database Seeding Strategy

Automated Environment Setup

```
// seed.js - Automated user and task creation
const seedDatabase = async () => {
  // Clean slate approach
  await pool.execute('DELETE FROM tasks');
  await pool.execute('DELETE FROM users');
```

```
  // Insert test data
};
```

- **Decision**: Automated seeding vs. manual data creation
- **Benefits**: Consistent test environment, faster development cycles
- **Implementation**: Clean slate approach prevents data conflicts

Development Environment Challenges

GitPod Integration

- **Challenge**: Local development limitations
- **Solution**: GitPod with custom Dockerfile for consistent environment
- **Configuration**: Docker Compose, MySQL client, ESLint/Prettier
- **Trade-off**: Larger image (~1GB) vs. complete development environment

Testing Strategy & Implementation

Unit Testing Approach

- **Framework**: Jest with Babel for ES module support
- **HTTP Testing**: Supertest for route integration tests
- **Database Testing**: Manual mocks and connection pooling
- **Coverage**: Focus on critical paths and error handling

Manual Testing Process

- **Tool**: Postman for API endpoint validation
- **Scenarios**: Role-based access, ownership verification, error conditions
- **Worker Testing**: Log monitoring for job processing verification

Performance & Security Considerations

Security Implementations

- **JWT**: HTTP-only cookies with secure flags
- **Password**: bcrypt hashing with cost factor 12
- **SQL Injection**: Parameterized queries throughout
- **Headers**: Helmet.js for security headers
- **Rate Limiting**: Express rate limiting middleware

Performance Optimizations

- **Database**: Strategic indexing on foreign keys and date fields
- **Connection**: MySQL connection pooling
- **Queue**: BullMQ with concurrency controls
- **Caching**: Redis for queue management

Scalability Considerations

Queue System Evolution

MVP: Redis + BullMQ (Single node)
↓
Production: Redis Cluster + BullMQ (Multi-node)
↓
Enterprise: Kafka + Custom consumers (Event streaming)

Database Scaling Path

MVP: Single MySQL instance
↓
Growth: MySQL Master-Replica setup
↓
Scale: Sharded MySQL or PostgreSQL cluster

Authentication Scaling

MVP: JWT with short expiration
↓
Enhanced: JWT + Refresh tokens in Redis
↓
Enterprise: OAuth2 + External identity provider

Known Limitations & Future Enhancements

Current MVP Limitations

1. **Notification System**: Console logging only, no persistence
2. **Security**: No data encryption at rest
3. **Monitoring**: Basic error logging, no metrics
4. **Deployment**: Single-instance design
5. **Testing**: Limited integration test coverage

Planned Enhancements

Security Improvements

- **Data Encryption**: AES-256 for sensitive task summaries
- **Rate Limiting**: Advanced rate limiting with Redis
- **Audit Logging**: Comprehensive action logging

Feature Expansions

- **Task Filtering**: Date ranges, status filters, search
- **Notification History**: Persistent notification storage
- **Real-time Updates**: WebSocket notifications

Infrastructure Enhancements

- **Metrics**: Prometheus + Grafana integration
- **CI/CD**: Automated testing and deployment
- **Container Orchestration**: Kubernetes deployment files

Debugging & Problem Resolution

Common Development Challenges

Docker Environment Issues

- **Problem**: Docker Desktop unavailable on my macOS
- **Solution**: GitPod cloud environment with Docker support
- **Workaround**: Local Homebrew MySQL for development
- **Validation**: `docker compose config` for syntax verification

ES Modules vs. CommonJS

- **Problem**: Jest incompatibility with ES modules
- **Attempted solutions**: Native Node.js test runner, Jest experimental ESM
- **Final solution**: Babel transpilation for test environment
- **Trade-off**: 30MB dependency vs. reliable test execution

Redis Connection Management

- **Problem**: BullMQ hanging on job creation
- **Investigation**: Connection sharing, worker lifecycle
- **Solution**: Centralized Redis connection with proper configuration
- **Prevention**: Connection health checks and proper error handling

Performance Debugging

Memory Management

- **Connection pooling**: Prevents connection leaks
- **Worker concurrency**: Balanced job processing

Risk Assessment & Mitigation

Security Risks

Risk	Likelihood	Impact	Mitigation
SQL Injection	Low	High	Parameterized queries
JWT Compromise	Medium	High	Short expiration, secure cookies
Data Exposure	Medium	Medium	Input validation, role checks
DoS Attacks	Medium	Medium	Rate limiting, connection limits

Operational Risks

Risk	Likelihood	Impact	Mitigation
Redis Failure	Medium	Medium	Fallback to direct processing
Database Downtime	Low	High	Connection retry logic
Worker Crashes	Medium	Low	Process monitoring, restart
Memory Leaks	Low	Medium	Connection pooling, monitoring

Code Quality & Maintainability

Design Patterns Applied

- **Singleton Pattern**: Redis and MySQL connection
- **Middleware Pattern**: Request processing pipeline
- **Factory Pattern**: Queue creation and management
- **Custom Error Class:** Utility Class with Strategy Pattern for error handling
- **Observer Pattern**: Event-driven notifications

Code Quality Metrics

- **Test Coverage**: 80%+ for critical paths

Testing Strategy

Unit Testing

- Controller logic testing
- Service layer validation
- Middleware functionality
- Database operations

Test Coverage Requirements

- Minimum 80% code coverage
- All critical paths tested
- Error handling validation
- Security feature testing

Deployment Considerations

Container Configuration

- Multi-stage Docker builds
- Environment-specific configurations
- Health check implementation
- Resource optimization

Production Readiness

- Database connection pooling
- Kubernetes manifest
- Error tracking

This documentation serves as a comprehensive guide for implementing the Task Management API system. All requirements should be implemented following the specified architecture and security guidelines.