**COSC 310**

# System Setup & Deployment Overview

Khizar Aamir, Abdullah Naqvi, Abdur Rehman, Haider Ali, Abdullah Munir

# Architecture

The task management system implements a modern full-stack web application architecture that separates concerns between frontend and backend while enabling efficient deployment as a single unit:

## Frontend (React.js Application)

- Framework: React.js with functional components and hooks
- State Management: Redux for centralized application state
- Build System: Vite for fast development and optimized production builds
- Styling: Tailwind CSS for utility-first styling approach
- Real-time Communication: Socket.IO client for bidirectional event-based communication
- Routing: React Router for client-side navigation
- Data Visualization: React Chartjs-2 for dashboard analytics

## Backend (Node.js Server)

- Framework: Express.js for RESTful API endpoints
- Database: MongoDB with Mongoose ODM for data modeling and validation
- Authentication: JSON Web Tokens (JWT) with access and refresh token strategy
- Real-time Communication: Socket.IO server with JWT authentication integration
- API Organization: Modular route structure organized by domain (auth, users, tasks, admin)
- Error Handling: Centralized middleware for consistent error responses
- Logging: Winston for structured application logging
- Testing: Jest for unit and integration testing

## Application Flow

1. Frontend sends authenticated requests to backend API endpoints
1. Backend validates requests, processes business logic, and interacts with the database
1. Real-time updates are pushed to clients via Socket.IO connections
1. Authentication state is maintained with HTTP-only cookies for refresh tokens and local storage for access tokens

## Containerization Strategy

The application uses a multi-stage Docker build process for optimized image size and security:

Stage 1 (Frontend Build):

- Uses Node.js 18 base image
- Installs frontend dependencies from package.json
- Builds static assets with Vite
- Results in optimized production-ready frontend bundle

Stage 2 (Application Server):

- Fresh Node.js 18 base image
- Installs backend dependencies
- Copies backend source code
- Integrates frontend build output from Stage 1 into backend's public directory
- Exposes port 3200 for the application
- Runs the combined application with a single command

# Environments

The system is designed to operate seamlessly across different environments:

## Development Environment

- Frontend:
  - Runs on Vite development server (port 5173)
  - Hot module replacement for instant feedback
  - Connects to backend via configured proxy or CORS
  - Environment variables managed via `.env` files
- Backend:
  - Runs on port 3200 (configurable via `PORT` environment variable)
  - CORS explicitly configured to allow connections from frontend dev server
  - MongoDB connection string specified via `MONGO_URI` environment variable
  - JWT secrets specified via environment variables
  - Nodemon for automatic server restarts during development
- Development Workflow:
  - Run frontend and backend separately during development
  - Backend: `cd backend && npm run dev`
  - Frontend: `cd frontend && npm run dev`

## Testing Environment

- Backend Testing:
  - Jest test runner with custom configuration

    - MongoDB memory server for isolated database testing
    - Supertest for HTTP endpoint testing
    - Custom mocks for external dependencies
    - Test setup in jest.setup.js
- Frontend Testing:
    - Jest and Vitest for component and integration tests
    - React Testing Library for component testing
    - MSW (Mock Service Worker) for API mocking
    - Custom test utilities in test helpers

### Production Environment

- Deployment Strategy:
    - Single Docker container deployment
    - Backend serves static frontend files from /public directory
    - Application runs on port 3200
    - Environment variables provided via Docker Compose or direct environment configuration
    - Persistent storage mounted for logs and potential file uploads
- Runtime Configuration:
    - Environment variables loaded from .env file
    - Production-specific optimizations enabled
    - Error stack traces hidden from client responses
    - Winston logging configured for production format

## CI/CD Pipeline

The project implements a comprehensive CI/CD pipeline using GitHub Actions:

### Pipeline Triggers

- Push Events:
    - Branches: main, development
    - Triggers full pipeline execution
- Pull Request Events:
    - Target Branches: main, development
    - Runs tests to validate proposed changes

### CI/CD Jobs and Workflow

**1. Testing Job (run-tests)**

- Environment: Ubuntu latest with Node.js 18
- Backend Testing Steps:
  - Check out repository code
  - Install backend dependencies: cd backend && npm install
  - Execute test suite: npm test
  - Tests include unit tests for controllers, models, and middleware
  - Integration tests for API endpoints
- Frontend Testing Steps:
  - Install frontend dependencies: cd frontend && npm install
  - Execute test suite: npm test
  - Tests include component rendering, state management, and API interactions

**2. Docker Build and Publish Job (docker-publish)**

- Dependencies: Requires successful completion of run-tests job
- Execution Conditions: Only runs on main branch pushes
- Steps:
  - Check out repository code
  - Authenticate with Docker Hub using secrets:
    - DOCKERHUB_USERNAME
    - DOCKERHUB_PASSWORD
  - Build multi-stage Docker image
  - Push image to Docker Hub with tag: mochi21/taskmanagment:latest

# Deployment Process

**Continuous Deployment**

- Image Retrieval:
  - Latest image pulled from Docker Hub: mochi21/taskmanagment:latest
  - Specified in docker-compose.yml
- Container Configuration:
  - Container named: task-management-container
  - Port mapping: Host port 3200 to container port 3200
  - Environment variables loaded from ./backend/.env

**Deployment Command:**

```
Docker-compose up -d
```

**Environment-Specific Configurations**

- Development and staging environments can use the same Docker image with different environment variables
- Configuration for different environments managed through environment-specific .env files or Docker Compose overrides

# Security Considerations

- Authentication: JWT tokens with appropriate expiration and refresh mechanism
- Authorization: Role-based access control (admin vs. regular users)
- Data Protection: Environment variables for sensitive information
- API Security: Input validation and sanitization via middleware
- Container Security: Multi-stage build to minimize attack surface
- Network Security: Only necessary ports exposed (3200)