Coding Test (TypeScript/Node.js): Priority Queue Simulation

Goal

Create a full-stack TypeScript/Node.js application that simulates a **priority queue of tasks**:

1. **Backend (Node.js/Express)**: A REST API and WebSocket server that manages a priority queue of tasks, allowing tasks to be added, retrieved, and processed over time with progress tracking.
2. **Frontend (React/TypeScript)**: A web-based visualization that provides real-time monitoring of the queue, task management interface, and progress tracking.

The complete system should demonstrate real-time communication between backend and frontend, with tasks progressing through the queue and being visualized in real-time.

Part 1 – Backend (Node.js/Express)

Tasks

Implement a REST API and WebSocket server that supports:

1. **Add a new task to the queue**
   Each task has the following properties:

   ```
   interface Task {
     id: string;
     name: string;
     priority: number;  // higher = more important
     progress: number;  // 0-100 as percents
     createdAt: Date;
   }
   ```

2. **Retrieve the current state of the queue**
   - Tasks should be returned sorted by priority (highest first).
3. **Simulate task processing**
   - Every 5 seconds, increase the progress of the current highest priority task by some value (e.g., 10-20%).
   - Progress should be tracked as a percentage from 0-100.
   - No skipping: if the current task hasn't been finished yet (progress < 100), continue processing it.
   - Only when a task reaches 100% progress should it be considered completed and moved to a "completed" list.
   - Then the next highest priority task becomes the current task to process.
4. **Prevent starvation** (optional but recommended)
   - Ensure low-priority tasks do not wait indefinitely.

- Example: compute an effective priority using aging:

  ```
  const effectivePriority = priority + Math.floor((Date.now() - task.createdAt.getTime()) / 1000) / agingFactor;
  ```

  This ensures older tasks gradually gain priority over time.

5. **WebSocket real-time updates**
   - Broadcast queue updates to all connected clients
   - Send notifications when tasks are added, progress changes, or completed

Requirements

- Use Node.js with Express for the REST API
- Use Socket.IO or native WebSocket for real-time communication
- Use TypeScript for type safety
- Store the queue in memory (no database required)
- The API should listen on `localhost:3000`
- Implement proper error handling and validation
- Use async/await for asynchronous operations

API Endpoints

```
GET    /api/tasks          - Get all tasks in queue
POST   /api/tasks          - Add new task to queue
GET    /api/tasks/completed - Get completed tasks
DELETE /api/tasks/completed - Clear completed tasks
```

WebSocket Events

```
client -> server: 'join_queue'     - Join queue monitoring
server -> client: 'queue_update'   - Queue state changed
server -> client: 'task_progress'  - Task progress updated
server -> client: 'task_completed' - Task completed
```

Part 2 – Frontend (React/TypeScript)

Tasks

Implement a React application that displays:

1. **Queue Visualization**
   - Show all tasks in the queue sorted by priority (highest first)
   - Display task details: ID, Name, Priority, Progress (0-100%), Created At
   - Visual progress bars for each task's completion status
   - Highlight the currently processing task
2. **Real-time Updates**
   - Connect to the backend via Socket.IO or WebSocket
   - Update the queue display in real-time as tasks progress
   - Show when tasks are completed and moved to the completed list

3. **Task Management Interface**
    - Form to add new tasks to the queue
    - Input fields for: Name, Priority
    - Submit button to send new task to backend
    - Form validation and error handling
4. **Completed Tasks Section**
    - Display list of completed tasks (progress = 100%)
    - Show completion timestamp
    - Option to clear completed tasks list

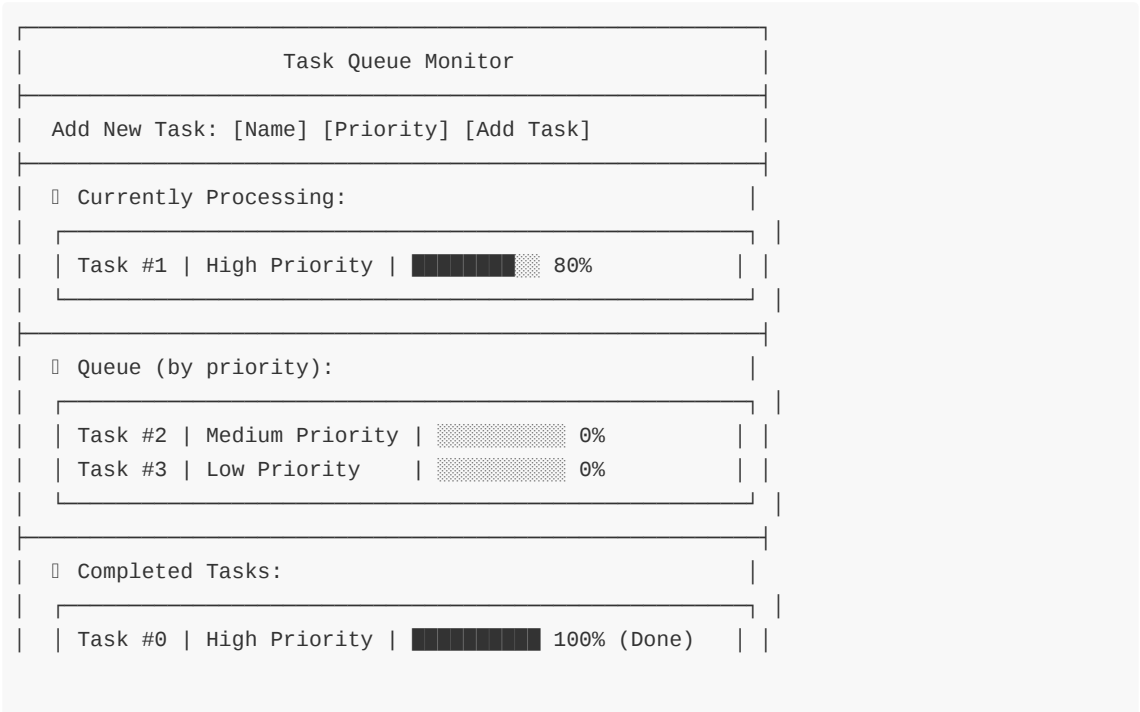
Requirements

- Use React with TypeScript
- Use modern React patterns (hooks, functional components)
- Implement responsive design that works on desktop and mobile
- Use a UI library
- Clean, intuitive user interface
- Real-time updates without page refresh
- Error handling for network issues
- Loading states and user feedback

UI/UX Suggestions

- **Queue Display**: Use cards or list items with progress bars
- **Priority Indication**: Color coding or icons for different priority levels
- **Current Task**: Special highlighting (e.g., pulsing animation, different border)
- **Progress Visualization**: Animated progress bars that update smoothly
- **Status Indicators**: Icons or badges for task states (pending, processing, completed)

Example Layout



```
│  └──────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────┘
```

Bonus Points

- Use Socket.IO for real-time queue updates
- Add comprehensive unit tests for both backend and frontend
- Implement graceful shutdown and proper error handling
- Add input validation and sanitization
- Use environment variables for configuration
- Provide a Dockerfile for easy deployment
- Add TypeScript strict mode
- Implement proper logging
- Add API rate limiting
- Use a state management library (Redux, Zustand) for frontend

---

Evaluation Criteria

Area	What will be assessed
TypeScript/Node.js	Code structure, async/await usage, type safety, API design
React/Frontend	Component design, hooks usage, state management, responsive design
Functionality	Correct priority queue behavior, starvation prevention, progress tracking
Architecture & Design	Separation of concerns, readability, maintainability, real-time communication
Documentation	README clarity, setup instructions, API documentation, deployment guide

---

Submission

Suggested repository structure:

```
/
├── backend/
│   ├── src/
│   │   ├── controllers/
│   │   ├── models/
│   │   ├── services/
│   │   ├── routes/
│   │   └── index.ts
│   ├── package.json
│   ├── tsconfig.json
│   └── README.md
├── frontend/
│   ├── src/
│   │   ├── components/
```

```
│   │   ├── hooks/
│   │   ├── services/
│   │   ├── types/
│   │   └── App.tsx
│   ├── public/
│   ├── package.json
│   ├── tsconfig.json
│   └── README.md
├── docker-compose.yml (optional)
```

Backend README should include:

- How to install dependencies and run the service (`yarn install && yarn run dev`)
- API endpoints documentation
- WebSocket events documentation
- Environment variables setup

Frontend README should include:

- How to install dependencies and run the app (`yarn install && yarn start`)
- Technology stack used
- Features implemented
- Available scripts