WAREHOUSE ROBOT DASHBOARD: DESIGN DOCUMENT

Author: Rutvik Kachhiya
Date: Dec, *2025*

1. System Architecture

The Warehouse Robot Dashboard is designed as a modular, simulation-driven system that mimics real-time warehouse robot fleet monitoring. The architecture emphasizes separation of concerns, scalability, and low latency updates.

Core Layers

1. **React Frontend (UI Layer)**
   Displays dashboards, robot cards, analytics, and task management screens. Handles user interaction and rendering logic.

2. **Zustand State Store (Application State Layer)**
   Central global state containing:
   - Robot telemetry (`bots[]`)
   - Task queues (`pendingTasks[]`, `tasks[]`)
   - UI states (theme, palette)
   - Simulation timers

3. **Mock API Simulation Layer**
   A lightweight, client-side “fake backend” used to generate:
   - Battery drain
   - Status changes (idle, busy, charging, error)
   - Speed variation
   - Random telemetry updates

4. **Simulation Engine**
   Uses timed intervals:
   - Every **10 seconds** → update robot telemetry
   - Every **3 seconds** → dequeue one pending task

5. **Optional Modules**

    - **SVG Map Viewer:** Renders user-uploaded warehouse layout and overlays robot positions.
    - **Three.js Digital Twin:** Displays a 3D simulation of robot movements.

High-Level Architecture Flow

```
User → React UI → Zustand Store → Mock API → Simulation Engine
→ Updated State → UI Re-render
```

---

2. UI/UX Decisions

✔ Card-Based Interaction Model

Bots are represented using interactive, gradient-styled cards including:

- Battery bars
- Status indicators
- Sparkline trend visualizations
- Expandable sections for details

This provides immediate at-a-glance visibility while preserving deeper insight.

✔ Minimal Cognitive Load

Pages are grouped by intent:

- **Dashboard:** high-level summary
- **Bot Status:** detailed robot-specific insights
- **Task Management:** create, assign, and track tasks
- **Analytics:** charts & statistics

✔ Theme & Palette

- Dark mode and multiple palettes (Electric, Ocean, Sunset)
- Enhances usability in control room environments with dim lighting

✔ SVG Map Viewer (Bonus)

- Users upload warehouse SVG layouts
- Robots plotted on top via coordinates
- Lightweight alternative to SLAM mapping systems

✔ Consistent Layout System

- Uses 24px grid spacing
- Tailwind utilities for uniform styling
- Responsive for tablet/desktop usage

---

3. Data Flow Explanation

The data flow is unidirectional, predictable, and optimized for real-time updates.

Step-by-Step Flow

1. **User Action**
    - Creates a task
    - Opens a page
    - Triggers refresh

2. **Zustand Store Action**
    - UI dispatches actions (`addTask`, `refreshBots`)
    - Store updates relevant slices

3. **Mock API Execution**
    - Generates new battery, speed, or status values
    - Simulates “robot behavior”

4. **Simulation Engine Timers**
    - Updates robot telemetry every 10s
    - Assigns/dequeues tasks every 3s

5. **UI Re-render**
    - Components subscribe using selectors
    - Only affected parts update → efficient performance

Data Flow Diagram

```
UI → Store → Mock API → Simulation Engine → Store (updated) → UI
```

---

4. Key Assumptions

- Telemetry refresh period is **10 seconds**, matching typical robot middleware reporting cycles.

- Task assignment simulation pops tasks every **3 seconds**.
- Authentication is **non-persistent**, following assignment rules.
- Robot movement is simulated (random walk), not controlled by real path-planning algorithms.
- SVG map does not require real-world positional calibration.
- The system is single-user for demonstration purposes.
- No backend exists; **100% front-end simulation**.

---

5. Trade-offs

Mock API Instead of Backend

Pros
✔ No backend setup required
✔ Fast iteration
✔ Deterministic demo behavior

Cons
✖ Cannot demonstrate concurrency
✖ No persistence
✖ Not reflective of production telemetry rate

Zustand Instead of Redux

Pros
✔ Minimal boilerplate
✔ Fast and lightweight
✔ Ideal for simulation loops
✔ Selectors prevent unnecessary re-renders

Cons
✖ Less structure for very large apps
✖ No native devtools unless added

Client-side Simulation

Pros
✔ Fully self-contained
✔ Works offline
✔ Easy for reviewers

Cons
✖ Not scalable for large fleets
✖ No real telemetry processing

SVG Map Instead of Real SLAM

Pros
✔ Extremely lightweight
✔ Easy to support arbitrary maps

Cons
✖ No real coordinate calibration
✖ No zone-based collision logic

---

6. State Management Design

The entire system relies on a predictable global state tree stored in **Zustand**.

State Slices

Robots (`bots[]`)

- id, name
- battery
- speed
- status
- currentTask
- lastUpdated

Tasks

- `pendingTasks[]` (queue)
- `tasks[]` (assigned)
- `addTask`, `removeTask`, `assignTask`

UI Slice

- theme mode
- palette selection

Simulation Slice

- intervals for bot updates
- intervals for task queue updates

Why Zustand Works Best

- Selective subscriptions → high performance
- Supports real-time-like updates
- Minimal mental load compared to Redux
- Works well with animation-heavy components (e.g., 60 FPS sparkline updates)

7. Improvements With More Time

1. Real Backend Integration

- WebSocket server for telemetry
- MQTT robot agent simulator
- Persisted task history

2. Detailed Telemetry Modeling

- Per-motor temperatures
- CPU load, connectivity, and fault flags

3. Collision & Path Visualization

- Draw robot paths
- Real-time traffic heatmaps

4. Advanced Analytics Module

- Productivity trends
- Battery health projections
- Robot utilization heatmaps

5. Multi-User Authentication

- JWT / OAuth
- Operator vs Supervisor roles

6. Accurate Map Calibration

- Convert SVG pixels to real warehouse meters
- Add zones (charging, loading, conveyor belts)

7. Full 3D Digital Twin

- Robot mesh animations
- Warehouse environment modeling
- Camera automation for fleet overview