Technical Design Document: Instant Marquees Melbourne Internal Operations Software (IMMIOS)

Project Name: Instant Marquees Melbourne Internal Operations Software (IMMIOS)

Version: 1.0
Date: July 11, 2025

1. Introduction & Purpose

This Technical Design Document (TDD) provides a detailed blueprint for the Instant Marquees Melbourne Internal Operations Software (IMMIOS). It translates the functional and non-functional requirements into a technical implementation plan, specifically tailored for development using **Next.js (React), Node.js, Tailwind CSS, and Firestore**, with a strong emphasis on real-time capabilities via **WebSockets**. This document is intended to guide the client's development process, particularly when leveraging Cursor AI for code generation.

2. High-Level Architecture

The IMMIOS will follow a **Serverless Full-Stack Architecture** leveraging Next.js API Routes for backend logic and a React frontend. Firestore will serve as the primary database, offering real-time data synchronization capabilities. WebSockets will be used for granular, live updates across connected clients.

```
graph TD
   UserBrowser[User Browser (React Frontend)]
   NextJSApp[Next.js Application (Frontend + API Routes)]
   Firestore[Firestore Database]
   WebSocketServer[WebSocket Server (Node.js/Socket.IO)]

   UserBrowser -- HTTP Requests --> NextJSApp
   UserBrowser -- WebSocket Connection --> WebSocketServer
   NextJSApp -- API Calls --> Firestore
   WebSocketServer -- Real-time Data Sync --> Firestore
   Firestore -- Real-time Data Changes --> WebSocketServer
   WebSocketServer -- Pushes Updates --> UserBrowser
```

Key Architectural Decisions:

- **Next.js:** Chosen for its React framework benefits (component-based UI), server-side rendering (SSR) for initial page load performance, API routes for integrated backend logic, and excellent Vercel deployment support.

- **Node.js (via Next.js API Routes):** Provides a unified JavaScript environment for both frontend and backend, leveraging client's existing expertise.
- **Firestore:** A NoSQL cloud database offering real-time data synchronization, scalability, and seamless integration with Firebase Authentication. Its document-based model is flexible for evolving data structures.
- **WebSockets (Socket.IO):** Essential for the Google Docs-like real-time updates and contextual notification badges. Socket.IO provides robust handling of connections, disconnections, and room-based messaging.
- **Tailwind CSS:** For utility-first CSS styling, enabling rapid and consistent UI development and responsive design.

3. Data Model (Firestore Schema Design)

The database schema will be designed to support the core functionalities, including job management, stock tracking (products & components), staff, vehicles, and user authentication. Firestore uses collections and documents.

Collection Naming Convention: Plural, lowercase (e.g., users, jobs).

3.1. users Collection

- **Purpose:** Stores user authentication details and links to staff profiles.
- **Document ID:** Firebase Auth uid.
- **Fields:**
    - email (string, unique)
    - passwordHash (string, generated by Firebase Auth)
    - role (string, enum: 'admin', 'staff')
    - staffId (string, optional, reference to staff document ID if linked)
    - createdAt (timestamp)
    - lastLoginAt (timestamp)

3.2. staff Collection

- **Purpose:** Stores details about each staff member, independent of their user account.
- **Document ID:** Auto-generated by Firestore.
- **Fields:**
    - name (string, required)
    - contactInfo (object, optional: phone, email)
    - notes (string, optional, e.g., "fast installer", "experienced with structures")

  - availability (map of string to array of strings, e.g., {"2025-12-01": ["08:00-12:00", "13:00-17:00"]}, for manual input)
  - linkedUserId (string, optional, reference to users document ID)
  - createdAt (timestamp)
  - updatedAt (timestamp)

3.3. products Collection

- **Purpose:** Defines the assembled items that are rented out.
- **Document ID:** Auto-generated by Firestore.
- **Fields:**
  - name (string, e.g., "3x3m Instant Pop-up Marquee - White")
  - description (string, optional)
  - currentStock (number, available assembled units)
  - minStockAlert (number, optional, threshold for low stock alert)
  - billOfMaterials (array of objects, see product_bom sub-collection or embedded)
  - createdAt (timestamp)
  - updatedAt (timestamp)

3.4. components Collection

- **Purpose:** Defines the raw materials or individual parts.
- **Document ID:** Auto-generated by Firestore.
- **Fields:**
  - name (string, e.g., "3x3m Instant Pop-up (40mm) Frame")
  - type (string, e.g., "frame", "roof", "wall", "weight")
  - size (string, e.g., "3m x 3m", "6m")
  - color (string, e.g., "white", "red", "clear")
  - condition (string, enum: "new", "festive", "damaged", etc.)
  - currentStock (number, available raw units)
  - minStockAlert (number, optional)
  - createdAt (timestamp)
  - updatedAt (timestamp)

3.5. jobs Collection

- **Purpose:** Stores all quote and confirmed job details. This will be the most complex document.
- **Document ID:** Auto-generated by Firestore.
- **Fields:**
  - clientName (string)
  - clientContact (object: phone, email)

    - status (string, enum: 'quote', 'job', 'completed', 'cancelled')
    - installDate (timestamp)
    - eventDate (timestamp)
    - removalDate (timestamp)
    - location (string)
    - notes (string, optional)
    - productsOrdered (array of objects: { productId: string, quantity: number, assembledQuantity: number }) - assembledQuantity tracks how many of the ordered products are currently assembled for this job.
    - assignedStaffIds (array of strings, references to staff document IDs)
    - assignedVehicleIds (array of strings, references to vehicles document IDs)
    - loadListStatus (string, enum: 'incomplete', 'mismatch', 'ready', 'loaded')
    - financialStatus (object: opsEmailed: boolean, invoiceSent: boolean, paid: boolean)
    - createdAt (timestamp)
    - updatedAt (timestamp)
    - **lastUpdatedSections (map):** This is crucial for granular notifications.
        - lastUpdatedSections.contactInfo (timestamp)
        - lastUpdatedSections.productsOrdered (timestamp)
        - lastUpdatedSections.dates (timestamp)
        - lastUpdatedSections.financialStatus (timestamp)
        - lastUpdatedSections.staffAssignments (timestamp)
        - lastUpdatedSections.vehicleAssignments (timestamp)
        - lastUpdatedSections.status (timestamp)
    - version (number, incremented on any update, for optimistic locking if needed)

3.6. vehicles Collection

- **Purpose:** Stores details about company vehicles.
- **Document ID:** Auto-generated by Firestore.
- **Fields:**
    - displayName (string, e.g., "Van 1", "Truck 2")
    - type (string, e.g., "van", "truck")
    - capacity (number, e.g., in cubic meters or weight units)
    - registration (string)
    - servicingDates (array of timestamps)
    - createdAt (timestamp)
    - updatedAt (timestamp)

3.7. settings Collection (Single Document)

- **Purpose:** Stores global application settings, including theme colors.
- **Document ID:** app-settings (fixed ID for a single document).
- **Fields:**
  - theme (object: mode: 'light' | 'dark', primaryColor: string (hex), accentColor: string (hex))
  - defaultView (string, e.g., 'month', 'week')
  - createdAt (timestamp)
  - updatedAt (timestamp)

3.8. user_notifications Collection

- **Purpose:** Tracks which users have seen which contextual changes.
- **Document ID:** Auto-generated by Firestore.
- **Fields:**
  - userId (string, reference to users document ID)
  - jobId (string, reference to jobs document ID)
  - section (string, enum: 'contactInfo', 'productsOrdered', 'dates', 'financialStatus', 'staffAssignments', 'vehicleAssignments', 'status')
  - seen (boolean, default: false)
  - createdAt (timestamp)

4. API Endpoints (Next.js API Routes)

Next.js API Routes will serve as the backend endpoints, interacting with Firestore and handling business logic.

- **api/auth/login (POST):** Authenticate user.
- **api/auth/logout (POST):** Log out user.
- **api/admin/users (GET, POST):** Get all users, create new user.
- **api/admin/users/[id] (PUT, DELETE):** Update/delete specific user.
- **api/admin/staff (GET, POST):** Get all staff profiles, create new staff profile.
- **api/admin/staff/[id] (PUT, DELETE):** Update/delete specific staff profile.
- **api/admin/products (GET, POST):** Get all products, create new product.
- **api/admin/products/[id] (PUT, DELETE):** Update/delete specific product.
- **api/admin/components (GET, POST):** Get all components, create new component.
- **api/admin/components/[id] (PUT, DELETE):** Update/delete specific component.
- **api/admin/vehicles (GET, POST):** Get all vehicles, create new vehicle.
- **api/admin/vehicles/[id] (PUT, DELETE):** Update/delete specific vehicle.
- **api/admin/settings (GET, PUT):** Get/update global application settings.
- **api/jobs (GET, POST):** Get all jobs, create new job (quote).
- **api/jobs/[id] (GET, PUT, DELETE):** Get/update/delete specific job.

    - **PUT endpoint logic:** This will be complex. It needs to detect *which* sections of the job were updated by comparing the old and new data, then update the lastUpdatedSections timestamps accordingly, and trigger relevant WebSocket events.
- **api/stock/check-availability (POST):** Check product/component availability for a given order.
- **api/stock/assemble (POST):** Trigger assembly of products from components, update stock.
- **api/stock/adjust (POST):** Manually adjust stock levels.
- **api/notifications/mark-seen (POST):** Mark specific job/section notifications as seen for a user.
- **api/staff/availability (GET, PUT):** Get/update staff availability.
- **api/loadlists/generate/[jobId] (GET):** Generate a loadlist for a specific job.

5. Real-time Communication (WebSockets with Socket.IO)

- **Server-Side (Node.js):**
    - A dedicated Socket.IO server will run alongside the Next.js application (or integrated within its API routes if using a custom server).
    - When a job document is updated in Firestore, a Firestore trigger (or direct backend logic) will detect the change.
    - The backend will then identify *which specific sections* of the job were modified (e.g., productsOrdered, financialStatus).
    - Based on the modified sections, the server will emit targeted WebSocket events to all connected clients.
    - **Event Structure Example:**
        - jobUpdated:contactInfo (payload: { jobId, lastUpdated: timestamp })
        - jobUpdated:productsOrdered (payload: { jobId, lastUpdated: timestamp })
        - jobUpdated:financialStatus (payload: { jobId, lastUpdated: timestamp })
        - jobUpdated:staffAssignments (payload: { jobId, lastUpdated: timestamp })
        - jobUpdated:vehicleAssignments (payload: { jobId, lastUpdated: timestamp })
        - jobUpdated:status (payload: { jobId, lastUpdated: timestamp })
- **Client-Side (React Frontend):**
    - The React application will establish a WebSocket connection to the server on user login.
    - It will listen for the specific jobUpdated events.
    - When an event is received, the client will:
        - Update the relevant user_notifications document in Firestore for the current user (setting seen: false for the specific job/section if it's a new

change).
        - Increment the corresponding badge count on the "Rostering Mode," "LoadList Mode," or "Financial Mode" buttons in the UI.
    - When a user clicks on a mode button, the client will mark all relevant notifications for that mode as seen in the user_notifications collection for the current user, causing the badge to reset.

6. Stock Management Logic (Detailed)

6.1. Product & Component Relationships

- Each product document will have a billOfMaterials array.
- Each item in billOfMaterials will reference a componentId and specify a quantity.
- **Example BOM for "3x3m Instant Pop-up Marquee - White":**
```
[
  { "componentId": "frame-40mm-3x3", "quantity": 1, "priority": 1 },
  { "componentId": "roof-3x3-white", "quantity": 1, "priority": 1 },
  { "componentId": "wall-3m-white", "quantity": 2, "priority": 2 }, // Example: 2 x 3m walls for one side
  { "componentId": "wall-6m-white", "quantity": 1, "priority": 2 }, // Example: 1 x 6m wall for another side
  { "componentId": "weight-20kg", "quantity": 20, "priority": 3 } // If weighted
]
```
    - priority can be used for assembly preferences (e.g., always use 40mm frames first, then 32mm if 40mm is exhausted).

6.2. Stock Availability Check (api/stock/check-availability)

- When a user adds products to a quote/job:
    1. The system calculates the total required components for all selected products based on their BOMs.
    2. It queries the components collection to check currentStock for each required component.
    3. If any component's currentStock is less than the required quantity, a warning is triggered.
    4. The system can also suggest alternative components if defined in the BOM and available.

6.3. Assembly Workflow (api/stock/assemble)

- **Trigger:** When a job requires X units of a product, but only Y (Y < X) are in products.currentStock.

- **Logic:**
    1. Calculate X - Y (deficit).
    2. Check components.currentStock for components needed to assemble the deficit.
    3. If components are sufficient, create an "Assembly Task" (perhaps a new document in a assembly_tasks collection or a status on the job itself).
    4. **Warehouse Manager Action:** The warehouse manager views these "Assembly Alerts" (e.g., on a dashboard or specific view). They confirm assembly completion.
    5. **Stock Update:** Upon confirmation, the backend:
        - Decrements components.currentStock for all used components.
        - Increments products.currentStock for the assembled product.
        - Updates the job.productsOrdered[].assembledQuantity field for the relevant job.
        - Triggers a WebSocket notification for stock changes.

6.4. LoadList Generation Logic (api/loadlists/generate/[jobId])

- Takes a jobId.
- Retrieves job.productsOrdered.
- For each product, retrieves its billOfMaterials from the products collection.
- Aggregates all components required for the entire job, including quantities and attributes (color, size, type).
- Applies specific rules (e.g., "each 3x3m marquee takes 20 weights, 1 stillage holds 60 weights"). This logic will reside in the backend API route.
- Generates a formatted list of components per vehicle, considering vehicle capacity if assigned.

7. User Interface (UI) / User Experience (UX) Considerations for Cursor AI

- **Component-Based Design:** Emphasize creating reusable React components for common UI elements (buttons, forms, tables, calendar cells, modals).
- **Tailwind CSS:** Use Tailwind utility classes directly in JSX for styling.
- **State Management:** For local component state, use React's useState and useReducer. For global application state (user info, current theme, notification counts), consider React Context API or a lightweight state management library like Zustand.
- **Calendar Library:** Recommend using a well-established React calendar library (e.g., react-big-calendar or fullcalendar/react) as a base, then customize its appearance with Tailwind.

- **Modals:** Implement custom modal components for confirmations, forms, and alerts instead of browser alert() or confirm().

8. Development Environment Setup (for Cursor AI)

This section will be a separate document, detailing the steps to set up the local development environment.

9. Security Considerations

- **Authentication:** Firebase Authentication will handle user sign-up/login.
- **Authorization:** Firestore Security Rules will be crucial to control data access based on user.role and user.uid. API routes will also implement server-side validation to ensure only authorized users can perform actions.
- **Data Validation:** All incoming data to API routes must be rigorously validated to prevent injection attacks and ensure data integrity.
- **Environment Variables:** Sensitive information (e.g., API keys for future integrations) will be stored as environment variables, not hardcoded.

10. Future Considerations (Out of Scope for MVP)

- **Advanced Scheduling Algorithms:** For optimal staff/vehicle suggestions based on historical data, location, traffic, and weather. This would involve more complex backend algorithms and potentially external API calls (Google Maps, Weather).
- **Deputy Integration:** Requires understanding Deputy's API for staff availability and shift pushing.
- **MYOB Integration:** Requires understanding MYOB's API for invoicing and payment status.
- **Comprehensive Audit Trails:** Beyond lastUpdatedSections, a full audit log of every change, who made it, and when.