**Factory Content Management System (FCMS) Development Architecture**

**Overview:**

The **Factory Content Management System (FCMS)** is a software solution that provides real-time updates, data management, and communication across multiple screens within a factory. The system will be locally hosted on a server, with all data and content managed and accessed through worker displays. The system allows real-time data manipulation and ensures that all screens reflect the latest changes.

**Tech Stack**

**Backend (Server-Side):**

1. **Node.js**
   * **Why Node.js?**  
     Node.js is well-suited for real-time, high-concurrency applications. It offers non-blocking, asynchronous operations, making it perfect for handling a large number of real-time communication requests, such as interactions between the server and the multiple displays.
2. **Express.js**
   * **Why Express.js?**  
     Express.js, a web framework built on Node.js, will be used to handle HTTP requests for serving content to the displays and managing API calls for fetching, updating, and controlling content on the server. It's lightweight, flexible, and can easily scale.
3. **WebSocket (Socket.io)**
   * **Why WebSockets?**  
     For real-time communication between the server and the worker displays, **WebSocket** will be used. WebSockets provide full-duplex communication, meaning data can flow in both directions in real-time without constant polling, ensuring a smooth and responsive experience.
     + **Socket.io**: A Node.js library for WebSockets that simplifies bidirectional communication.
4. **Database (MongoDB / SQL Database)**
   * **Why MongoDB?**  
     MongoDB is a NoSQL database, which is ideal for flexible data storage (such as documents, media references, training modules, etc.). It's highly scalable and efficient for managing a variety of data types without rigid schemas.
     + If relational data is needed (for structured queries or joins), a SQL-based database like **MySQL** or **PostgreSQL** could also be used.
5. **File Storage (Local Server Storage or NAS)**
   * **Why Local Storage?**  
     All media files (images, PDFs, drawings, videos, etc.) will be stored on the server itself. For redundancy and scalability, the server could be paired with a **Network Attached Storage (NAS)** system to store large amounts of media data, ensuring that it's always accessible to the displays.

**Frontend (Client-Side):**

1. **HTML5**
   * Used for structuring the content displayed on the screens (using simple layouts to ensure compatibility across various devices).
2. **CSS3**
   * CSS will be used for styling the interfaces of all screens, ensuring that content is presented in a clear, readable, and accessible way. Responsive design techniques will be used to ensure compatibility with various display sizes.
3. **JavaScript (Vanilla or React)**
   * **Vanilla JS** or **React.js** can be used for interactive and dynamic content on the worker screens. React can be used for better manageability and scalability of dynamic UI components if the system's complexity grows.
   * React’s component-based architecture will make it easier to handle the different types of data and content that each screen will display.
4. **WebSocket (Client-Side - Socket.io Client)**
   * On the client side, **Socket.io** will be used to open and maintain WebSocket connections to the server. This will allow real-time updates to be pushed to the screens without needing to reload the page.

**Communication between Screens and Server**

1. **Real-Time Communication with WebSockets:**
   * **Socket.io** will be used to establish WebSocket connections between the worker displays and the server.
   * The server can push content (e.g., documents, work instructions, training materials) to specific displays or all connected displays based on the requirements. This means that updates to content (work instructions, training updates, etc.) will appear on the displays in real-time.
2. **Request-Response Mechanism:**
   * The displays (client-side) will send requests to the server using **HTTP/REST APIs** for fetching content, submitting data (e.g., form submissions, check sheets), and receiving notifications.
   * **WebSocket channels** will be used for sending real-time updates (e.g., work instructions, alerts, data manipulation).
3. **Real-Time Data Manipulation:**
   * The backend (Node.js) will listen for data changes (e.g., content changes, updates, or worker actions) via WebSocket events. When a change occurs (such as an update to a document or check sheet), the server will broadcast the update to all relevant screens.
   * The server will handle changes, validation, and updates (e.g., new check sheets, updated training content, etc.), ensuring that the displays reflect the latest data.

**Real-Time Data Manipulation**

1. **Server-Side Logic (Node.js + MongoDB):**
   * The server will manage content dynamically by using a **centralized database** (MongoDB or SQL). When a change is made (e.g., adding a new document or modifying existing content), the server will update the database and broadcast the change using **Socket.io** to all connected displays.
   * For example, when a new **work instruction** is uploaded or updated, the server will notify the relevant screen(s) and push the updated content to the display.
2. **Event-Based Notifications:**
   * **Event-based notifications** will be sent to the relevant screens when certain triggers occur (e.g., new documents uploaded, production issues raised in Andon, training updates).
   * **WebSocket Events** will handle pushing notifications (alerts, due dates, critical issues) to the screens in real-time.
3. **Query-Based Reporting:**
   * The backend will handle querying data (e.g., check sheets, production data) and generate reports based on user requests. The server will prepare the data and send it to the display or allow workers to download the reports in formats like PDF, CSV, etc.

**System Efficiency for Local Hosting**

1. **Local Hosting (On-Premises Server):**
   * Running the system locally on a **dedicated server** within the factory ensures fast access to all content and data, minimizes network latency, and allows full control over data security.
   * The system can run on **low-cost hardware** with appropriate specifications to handle the workload of multiple worker screens and real-time updates. A **local network** will be used to connect the server and displays for optimal performance.
2. **Minimal Network Dependencies:**
   * The system is designed to work **locally** without dependency on external networks, ensuring that factory operations continue smoothly even in the event of an internet outage.
   * WebSocket connections will be maintained over the **local network**, ensuring low latency and fast real-time updates.

**Conclusion:**

The Factory Content Management System (FCMS) will use a **robust, efficient tech stack** designed to work **locally** within the factory network. The use of **Node.js** for backend logic, **WebSockets** for real-time communication, and a flexible database (MongoDB) ensures that the system can scale and perform well across multiple displays. Real-time updates will keep the workers informed, with the server managing and manipulating data dynamically to ensure smooth, uninterrupted operations.