# SYSTEM ARCHITECTURE DOCUMENT

**StockFlow: Enterprise Inventory Management System**

## 1 1. Architectural Design Overview

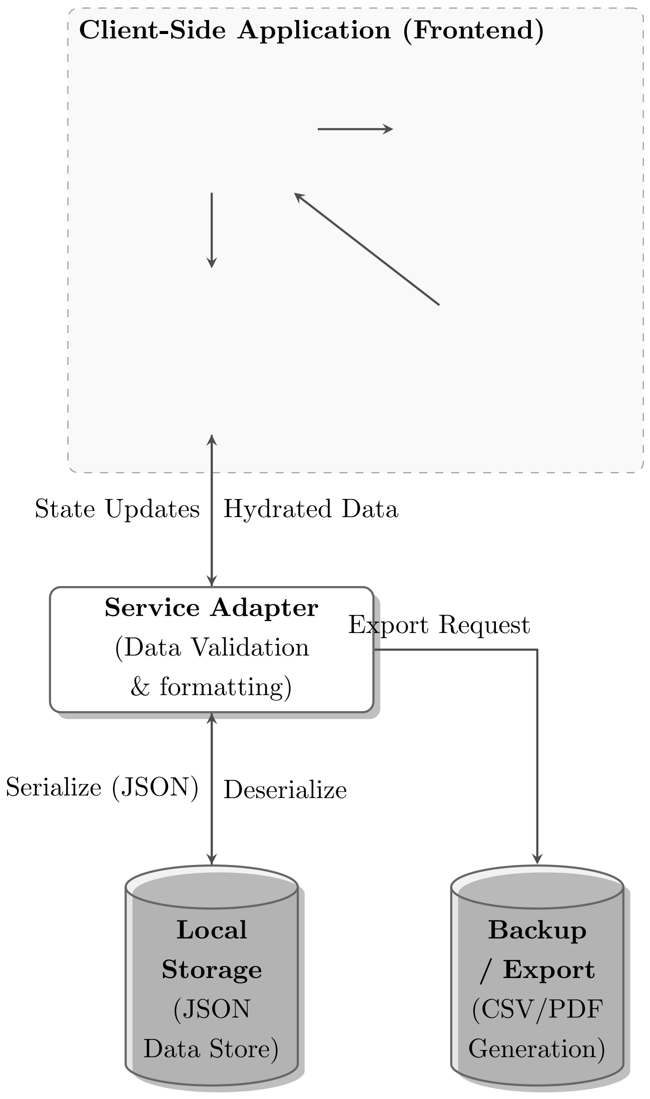
The "StockFlow" system is architected as a modern **Single Page Application (SPA)** utilizing a component-based frontend framework. The architecture follows the **Model-View-Controller (MVC)** pattern adapted for client-side rendering. This design decouples the user interface from the data logic, ensuring modularity, testability, and scalability.

The system is divided into three primary logical layers:

1. **Presentation Layer (Frontend):** Responsible for rendering the User Interface (UI) and handling user interactions.
2. **Application Logic Layer (Middle Tier):** Manages state, validation, and business rules.
3. **Data Persistence Layer (Backend/Storage):** Handles data storage, retrieval, and integrity.

## 2 2. Component Architecture Diagram

The following diagram illustrates the high-level interaction between system components, highlighting the data flow from the user interface down to the persistence layer.


**Client-Side Application (Frontend)**
State Updates
Hydrated Data
**Service Adapter**
(Data Validation & formatting)
Export Request
Serialize (JSON)
Deserialize
**Local Storage** (JSON Data Store)
**Backup / Export** (CSV/PDF Generation)

## 3 3. Detailed Component Breakdown

### 3.1 3.1. Frontend Subsystem (React.js)

The frontend is the entry point for all user interactions. It is built using React.js to create a dynamic and responsive user experience.

- **Router Module:** Manages navigation between the Dashboard, Inventory List, and Settings views without triggering full page reloads.
- **Context Providers:** Acts as a global state container, holding the current inventory list, user preferences, and theme settings. This avoids "prop drilling" and ensures data consistency across components.
- **Presentation Components:** Reusable UI elements (e.g., `<Button />`, `<Input />`, `<Modal />`) styled with Tailwind CSS utility classes.

### 3.2 3.2. Logic & Service Layer

This layer abstracts the complexity of data manipulation from the UI components.

- **Validation Engine:** Checks user input against defined schemas (e.g., ensuring Price $> 0$, Name is not empty) before processing.
- **CRUD Controller:** specific functions (`addItem`, `updateItem`, `deleteItem`) that handle business logic, such as updating total counts or triggering low-stock alerts.

### 3.3 3.3. Data Persistence (Simulation)

In the current prototype phase, the system uses the browser's `LocalStorage` API to mimic a persistent database.

- **Schema Structure:** Data is stored as a serialized JSON string under the key `stockflow_data`.
- **Entity Relationship:** The JSON structure represents an array of Item objects, where each object contains `id` (UUID), `name`, `category`, `quantity`, and `price`.

## 4 4. Data Flow Strategy

1. **Input:** The user submits a form (e.g., adding a new SKU).
2. **Validation:** The React Component captures the event and passes data to the Validation Engine.

3. **State Update:** Upon validation success, the Global State Context is updated, triggering an immediate re-render of the UI (Optimistic UI Update).
4. **Persistence:** The Service Adapter asynchronously serializes the new state and writes it to LocalStorage.

## 5 5. Security Considerations

While a client-side prototype has inherent limitations, the architecture enforces:

- **Input Sanitization:** Preventing basic XSS attacks by escaping user input before rendering.
- **Access Control:** Mocking an authentication barrier to restrict access to the Admin Dashboard.