 # Root-Level Data Flow Analysis Report

DMS Frontend Codebase - Comprehensive Analysis

****Analysis Date:**** December 19, 2025

****Codebase:**** DMS Frontend (Next.js Application)

****Analysis Type:**** Root-Level Data Flow, State Management, and Centralization Audit

---

Executive Summary

This report presents a comprehensive root-level analysis of the DMS frontend codebase, identifying critical data flow issues, architectural inconsistencies, broken flows, and decentralization problems. The analysis reveals ****severe fragmentation**** across state management, API clients, authentication systems, and data sources that require immediate architectural refactoring.

Critical Findings Overview

| Category | Issues Found | Severity |

|----------|-------------|----------|

| State Management | Dual systems (Context + Zustand) | 🔴 Critical |

| API Clients | Dual implementations | 🔴 Critical |

| Authentication | Duplicate AuthContext | 🔴 Critical |

| Data Sources | 23 mock files + services | 🟡 High |

| Local State | 500+ useState instances | 🟡 High |

| Type Definitions | 20 scattered type files | 🟡 High |

---

1. State Management Architecture

1.1 Dual State Management Systems ⚠️ CRITICAL ISSUE

The codebase implements **TWO SEPARATE** state management systems that operate independently:

System 1: Context API

- **Location:** `src/contexts/`
- **Files:**
  - `AuthContext.tsx` / `AuthContext.ts`
  - `RoleContext.tsx` / `RoleContext.ts`
  - `ToastContext.tsx` / `ToastContext.ts`

System 2: Zustand Store

- **Location:** `src/store/`
- **Files:**
  - `authStore.ts` - Persisted auth state with localStorage
  - `toastStore.ts` - Toast notification state

Problem Analysis:

```mermaid
graph TD
    A[Component] -->|Uses| B[AuthContext]
    A -->|Also Uses| C[useAuthStore]
    B -->|Wraps| D[useRole Hook]
    D -->|Calls| C
    C -->|Persists to| E[localStorage]
    C -->|Syncs to| F[Cookies]
```

```
    style A fill:#f9f,stroke:#333
    style B fill:#ff9,stroke:#333
    style C fill:#ff9,stroke:#333
```

Data Flow Conflict:

1. `AuthContext` (contexts/AuthContext.tsx) wraps `useRole` hook
2. `useRole` hook internally calls `useAuthStore` (Zustand)
3. Components can use either `useAuth()` or `useAuthStore()` directly
4. **Result:** Two sources of truth for authentication state

Code Evidence:

```typescript
// contexts/AuthContext.tsx (Line 21)
const { userRole, userInfo, updateRole, isLoading } = useRole();

// shared/hooks/useRole.ts (Line 10)
const { userRole, userInfo, setAuth, isLoading } = useAuthStore();

// app/page.tsx (Line 75) - Direct Zustand usage
useAuthStore.getState().setAuth(user.role, userInfo);
```

Impact:

- ❌ State synchronization issues
- ❌ Potential race conditions
- ❌ Difficult to debug state changes

- ❌ Inconsistent state updates across components

1.2 Duplicate AuthContext Implementations ⚠️ CRITICAL ISSUE

TWO IDENTICAL AuthContext implementations exist:

1. **`src/contexts/AuthContext.tsx`** (48 lines)
2. **`src/core/auth/AuthContext.tsx`** (48 lines)

Both export:
- `AuthProvider` component
- `useAuth()` hook
- Identical `AuthContextType` interface

Problem: No clear indication which one is being used where, leading to:
- Import confusion
- Potential runtime conflicts
- Maintenance nightmares

1.3 Local State Proliferation 🟡 HIGH SEVERITY

500+ instances of `useState` across the application:

Sample Analysis:

```typescript
```

```
// app/inventory-manager/parts-master/page.tsx
const [parts, setParts] = useState<Part[]>([]);
const [filteredParts, setFilteredParts] = useState<Part[]>([]);
const [isLoading, setIsLoading] = useState(true);
const [showModal, setShowModal] = useState(false);
const [showUploadModal, setShowUploadModal] = useState(false);
const [editingPart, setEditingPart] = useState<Part | null>(null);
const [searchQuery, setSearchQuery] = useState("");
const [categoryFilter, setCategoryFilter] = useState("");
const [uploadFile, setUploadFile] = useState<File | null>(null);
const [uploadProgress, setUploadProgress] = useState<{...}>();
const [formData, setFormData] = useState<PartsMasterFormData>(...);
// 11 useState hooks in a single component!
```

Problems:

- ❌ No centralized state management for domain data
- ❌ Data duplication across components
- ❌ Props drilling for shared state
- ❌ Difficult to maintain consistency
- ❌ No single source of truth for entities (parts, customers, vehicles, etc.)

---

2. API Client Architecture

2.1 Dual API Client Implementations ⚠️ CRITICAL ISSUE

TWO SEPARATE API client implementations:

Client 1: Core API Client

- **Location:** `src/core/api/client.ts` (218 lines)
- **Features:**
  - Request/response interceptors
  - Retry logic with exponential backoff
  - In-memory caching for GET requests
  - Error handling
  - Uses native `fetch` API

Client 2: Real API Client

- **Location:** `src/lib/api/real-client.ts` (158 lines)
- **Features:**
  - Timeout handling with AbortController
  - Token management from localStorage
  - Error handling with custom ApiError class
  - Uses native `fetch` API

Comparison:

Feature	Core API Client	Real API Client
Interceptors	☑ Yes	✗ No
Retry Logic	☑ Yes (3 attempts)	✗ No
Caching	☑ Yes (in-memory)	✗ No
Timeout	✗ No	☑ Yes (configurable)
Auth Token	☑ Via interceptor	☑ Via header builder

| Error Handling | ☑ Custom handler | ☑ ApiError class |

Problem:

- No clear usage pattern - which client to use when?
- Duplicate functionality
- Inconsistent error handling
- Different response formats

2.2 Mock vs Real API Pattern 🟡 HIGH SEVERITY

Services implement a **dual-mode pattern** that's inconsistent:

```typescript
// services/central-inventory/centralStock.service.ts
class CentralStockService {
  private useMock: boolean;

  constructor() {
    this.useMock = API_CONFIG.USE_MOCK; // From env variable
    this.setupMockEndpoints();
  }

  async getCentralStock(): Promise<CentralStock[]> {
    if (this.useMock) {
      return await centralInventoryRepository.getAllStock();
    }
    const response = await apiClient.get<CentralStock[]>("/central-inventory/stock");
```

```
    return response.data;
  }
}
```

Issues:

1. **Every service method** has `if (this.useMock)` branching
2. Mock endpoints registered in service layer (should be in mock server)
3. Repository pattern only used for mocks, not real API
4. Inconsistent data transformation between mock and real responses

---

3. Data Sources & Flow

3.1 Mock Data Architecture

23 Mock Data Files in `src/__mocks__/data/`:

```
appointments.mock.ts
approvals.mock.ts
auth.mock.ts
central-inventory.mock.ts
complaints.mock.ts
customer-service-history.mock.ts
customers.mock.ts
dashboard.mock.ts
inventory.mock.ts
```

invoices.mock.ts

job-cards.mock.ts

jobs.mock.ts

leads.mock.ts

quotations.mock.ts

reports.mock.ts

service-centers.mock.ts

service-history.mock.ts

service-requests.mock.ts

users.mock.ts

vehicles.mock.ts

workflow-mock-data.ts

workshop.mock.ts

```

#### **Problems:**

- ❌ No centralized mock data registry
- ❌ Hardcoded data in multiple files
- ❌ No data relationships maintained
- ❌ Difficult to update related entities
- ❌ No data seeding strategy

---

### 3.2 Repository Pattern (Incomplete)

Only **2 repositories** exist for mock data:

- `central-inventory.repository.ts` (460 lines)
```

- `customer.repository.ts`

Analysis:

What Works:

- ☑ Simulates database operations
- ☑ Uses localStorage for persistence
- ☑ Maintains data relationships
- ☑ Implements CRUD operations

What's Broken:

- ❌ Only used for mocks, not real API
- ❌ Only 2 repositories for 23 mock data files
- ❌ Other entities accessed directly from mock files
- ❌ No consistent data access pattern

3.3 Service Layer Architecture

14 Service Files in `src/services/`:

```
central-inventory/
  ├── adminApproval.service.ts
  ├── centralIssue.service.ts
  ├── centralPurchaseOrder.service.ts
  ├── centralPurchaseOrderCreate.service.ts
```

```
  ├── centralStock.service.ts
  └── invoice.service.ts

inventory/
  ├── inventory-approval.service.ts
  ├── jobCardPartsRequest.service.ts
  ├── partsEntry.service.ts
  ├── partsMaster.service.ts
  ├── partsOrder.service.ts
  └── stockUpdateHistory.service.ts

parts/
  └── partsIssue.service.ts

quotations/
  └── quotations.service.ts
```

Pattern Analysis:

Consistent Pattern:

```typescript
class Service {
  private useMock: boolean;

  constructor() {
    this.useMock = API_CONFIG.USE_MOCK;
    this.setupMockEndpoints(); // Registers mock handlers
  }
```

```
  async method() {
    if (this.useMock) {
      // Call repository or mock data directly
    }
    // Call real API
  }
}
```

Issues:

1. **Mixed Responsibilities:** Services handle both business logic AND mock setup
2. **Tight Coupling:** Services directly depend on repositories
3. **No Abstraction:** No data access layer abstraction
4. **Inconsistent Returns:** Mock and real API may return different shapes

---

4. Type System Architecture

4.1 Type Definition Structure

20 Type Files in `src/shared/types/`:

```
api.types.ts
appointment.types.ts
auth.types.ts
central-inventory.types.ts
```

check-in-slip.types.ts

common.types.ts

home-service.types.ts

identity.types.ts

inventory-approval.types.ts

inventory.types.ts

invoice.types.ts

job-card.types.ts

jobcard-inventory.types.ts

otc.types.ts

parts-issue.types.ts

quotation.types.ts

service-request.types.ts

vehicle.types.ts

workshop.types.ts

```

#### **Analysis:**

**What Works:**

- ☑ Centralized in `shared/types/`
- ☑ Domain-driven organization
- ☑ Exported through `index.ts`

**What's Broken:**

- ❌ No shared base types (e.g., `BaseEntity`, `Timestamped`)
- ❌ Duplicate type definitions across files
- ❌ No validation schemas (Zod, Yup, etc.)
```

- ❌ Form types mixed with domain types
- ❌ API response types not separated from domain types

4.2 Type Inconsistencies

Example: User/UserInfo Duplication

```typescript
// auth.types.ts
export interface UserInfo {
  id: string;
  name: string;
  email: string;
  role: UserRole;
  serviceCenterId?: string;
}

// users.mock.ts (inferred type)
interface MockUser {
  email: string;
  password: string;
  role: UserRole;
  name: string;
  serviceCenterId?: string;
}
```

Problem: No `id` field in MockUser, but required in UserInfo

---

5. Data Flow Diagrams

5.1 Current Authentication Flow

```mermaid
sequenceDiagram
    participant User
    participant LoginPage
    participant Zustand
    participant Context
    participant Cookie
    participant LocalStorage

    User->>LoginPage: Enter credentials
    LoginPage->>Zustand: useAuthStore.setAuth()
    Zustand->>Cookie: Set auth_role, auth_token
    Zustand->>LocalStorage: Persist state
    LoginPage->>Context: updateRole() via useAuth
    Context->>Zustand: Calls useRole -> useAuthStore

    Note over Zustand,LocalStorage: State stored in 3 places!
```

Issues:

- 🔴 State stored in: Zustand store, localStorage, AND cookies

- 🔴 Context wraps Zustand (unnecessary layer)
- 🔴 Direct Zustand access bypasses Context

5.2 Current Data Fetching Flow

```mermaid
graph TD
    A[Component] -->|useState| B[Local State]
    A -->|useEffect| C[Service Layer]
    C -->|Check useMock| D{Mock Mode?}
    D -->|Yes| E[Repository]
    D -->|No| F[API Client]
    E -->|Read/Write| G[localStorage]
    E -->|Read| H[Mock Data Files]
    F -->|HTTP| I[Backend API]
    C -->|Return data| A
    A -->|setState| B

    style D fill:#ff9,stroke:#333
    style E fill:#9f9,stroke:#333
    style F fill:#9f9,stroke:#333
```

Issues:

- 🔴 No caching strategy (except in Core API Client)
- 🔴 No global state for fetched data

- 🔴 Each component manages its own loading/error states
- 🔴 Data refetched on every component mount

5.3 Broken Data Flow: Inventory Example

```mermaid
graph LR
    A[Parts Master Page] -->|useState| B[Local Parts State]
    C[Parts Order Page] -->|useState| D[Local Parts State]
    E[Parts Entry Page] -->|useState| F[Local Parts State]

    B -.->|No sync| D
    D -.->|No sync| F
    F -.->|No sync| B

    G[Central Parts Service] -->|Fetch| H[Mock/API]
    A -->|Fetch| G
    C -->|Fetch| G
    E -->|Fetch| G

    style B fill:#f99,stroke:#333
    style D fill:#f99,stroke:#333
    style F fill:#f99,stroke:#333
```

Problem: Three different pages manage the same "Parts" data independently with no synchronization.

---

6. Centralization Issues

6.1 Decentralized Data Access

Entity	Access Pattern	Issues
Parts	Direct service calls in each component	No shared cache
Customers	Direct service calls + local state	Duplicate data
Vehicles	Direct service calls + local state	No relationships
Job Cards	Direct service calls + local state	Complex state
Inventory	Direct service calls + local state	Stock inconsistencies

---

6.2 Configuration Fragmentation

Multiple config locations:

1. **`src/config/api.config.ts`** - API endpoints and config
2. **`next.config.mjs`** - Next.js configuration
3. **`.env.example`** - Environment variables
4. **`src/shared/constants/`** - Application constants (7 files)

Problem: No single source for configuration values

6.3 Utility Function Duplication

18 utility files in `src/shared/utils/`:

```
api.utils.ts
auth.utils.ts
cn.ts
date.utils.ts
error.utils.ts
format.utils.ts
localStorage.ts
number.utils.ts
pdf.utils.ts
print.utils.ts
search.utils.ts
sort.utils.ts
storage.utils.ts
string.utils.ts
table.utils.ts
test.utils.ts
validation.utils.ts
```

Issues:

- ❌ Potential duplicate functions across files
- ❌ No tree-shaking optimization
- ❌ Unclear which util to use for common tasks

---

7. Broken Flows Identified

7.1 Authentication Flow Issues

Problem 1: Inconsistent Authentication Check

```typescript
// contexts/AuthContext.tsx (Line 23)
const isAuthenticated = !!userInfo && userRole !== "admin";

// store/authStore.ts (Line 34)
isAuthenticated: !!user && role !== 'admin'
```

Issue: Why is `admin` role considered "not authenticated"? This logic is inconsistent with typical auth patterns.

Problem 2: Cookie vs LocalStorage Sync

```typescript
// store/authStore.ts
setAuth: (role, user) => {
  Cookies.set('auth_role', role, { expires: 7 });
  Cookies.set('auth_token', 'mock_token', { expires: 7 });
  // Also persisted to localStorage via Zustand middleware
}
```

```

**Issue:** Hardcoded `'mock_token'` - not using real token from login response

---

### 7.2 Data Mutation Flow Issues

**Problem: No Optimistic Updates**

Current flow for updating data:

1. User clicks save
2. Component calls service
3. Service calls API/repository
4. Wait for response
5. Update local state
6. Re-render

**Missing:**

- ❌ Optimistic UI updates
- ❌ Rollback on error
- ❌ Global state invalidation
- ❌ Related data updates

---

### 7.3 Search/Filter Flow Issues
```

Problem: Inconsistent Search Implementation

```typescript
// Some components: Client-side filtering
const filtered = items.filter(item =>
  item.name.toLowerCase().includes(query.toLowerCase())
);

// Other components: Server-side search
const results = await service.search(query);

// No consistent pattern!
```

---

8. Missing Patterns & Best Practices

8.1 Missing: Global State Management

What's Needed:

- ☑ Single source of truth for domain entities
- ☑ Normalized state structure
- ☑ Automatic cache invalidation
- ☑ Optimistic updates
- ☑ Request deduplication

Recommended Solutions:

1. **TanStack Query (React Query)** - For server state

2. **Zustand** (already installed) - For client state

3. Remove Context API for state management

8.2 Missing: Data Normalization

Current: Nested, denormalized data

```typescript
{
  jobCard: {
    id: "jc-1",
    customer: { id: "c-1", name: "John", ... },
    vehicle: { id: "v-1", make: "Toyota", ... },
    parts: [
      { id: "p-1", name: "Oil Filter", ... },
      { id: "p-2", name: "Air Filter", ... }
    ]
  }
}
```

Needed: Normalized structure

```typescript
{
  jobCards: { "jc-1": { id: "jc-1", customerId: "c-1", vehicleId: "v-1", partIds: ["p-1", "p-2"] } },
  customers: { "c-1": { id: "c-1", name: "John", ... } },
  vehicles: { "v-1": { id: "v-1", make: "Toyota", ... } },
```

```
  parts: {
    "p-1": { id: "p-1", name: "Oil Filter", ... },
    "p-2": { id: "p-2", name: "Air Filter", ... }
  }
}
```

8.3 Missing: Error Boundary Strategy

Current: Ad-hoc error handling in components

```typescript
try {
  const data = await service.getData();
  setData(data);
} catch (error) {
  console.error(error); // Just logging!
  // No user feedback, no recovery
}
```

Needed:

- ☑ Global error boundaries
- ☑ Error recovery strategies
- ☑ User-friendly error messages
- ☑ Error reporting/logging service

8.4 Missing: Loading State Management

Current: Each component manages loading state

```typescript
const [isLoading, setIsLoading] = useState(false);
const [data, setData] = useState([]);
const [error, setError] = useState(null);
```

Needed: Centralized loading state with React Query or similar

---

9. Performance Issues

9.1 Unnecessary Re-renders

Problem: Local state changes trigger full component re-renders

Example:

```typescript
// Every keystroke in search triggers re-render
const [searchQuery, setSearchQuery] = useState("");
const filtered = items.filter(item =>
  item.name.includes(searchQuery)
);
```

Solution: Debouncing, memoization, or virtual scrolling

9.2 No Code Splitting

Problem: All services and components loaded upfront

Evidence: No dynamic imports found in service layer

Impact: Large initial bundle size

9.3 No Request Deduplication

Problem: Multiple components requesting same data simultaneously

Example:

```typescript
// Component A
useEffect(() => { service.getParts(); }, []);

// Component B (rendered at same time)
useEffect(() => { service.getParts(); }, []);

// Result: 2 API calls for same data!
```

---

10. Security Concerns

10.1 Token Storage

Current: Tokens stored in:

- localStorage (via Zustand persist)

- Cookies (via js-cookie)

Issues:

- ❌ localStorage vulnerable to XSS

- ❌ Cookies set without `httpOnly` flag

- ❌ No token refresh mechanism

- ❌ Hardcoded mock token

10.2 No Input Validation

Problem: No validation schemas for forms

Evidence: No Zod, Yup, or similar validation library usage

Risk: Invalid data sent to API, potential injection attacks

---

11. Recommendations

11.1 Immediate Actions (Critical)

1. Consolidate State Management

Action Plan:

```typescript
// Remove: Context API for state management
// Keep: Zustand for client state
// Add: TanStack Query for server state

// New structure:
src/
  store/
    auth.store.ts         // Client state (Zustand)
    ui.store.ts           // UI state (Zustand)
  hooks/
    queries/
      usePartsQuery.ts     // Server state (TanStack Query)
      useCustomersQuery.ts
      useJobCardsQuery.ts
```

Benefits:

- ☑ Single source of truth
- ☑ Automatic caching
- ☑ Request deduplication

- ☑ Optimistic updates
- ☑ Background refetching

---

2. Consolidate API Clients

Action Plan:

```typescript
// Remove: lib/api/real-client.ts
// Keep: core/api/client.ts (enhanced)
// Add: Mock Service Worker (MSW) for mocking

// New structure:
src/
  core/
    api/
      client.ts          // Single API client
      interceptors.ts
      error-handler.ts
  mocks/
    handlers/            // MSW handlers
      parts.handlers.ts
      customers.handlers.ts
    browser.ts           // MSW browser setup
    server.ts            // MSW server setup (for tests)
```

Benefits:

- ☑ Single API client
- ☑ Consistent error handling
- ☑ Better mock management
- ☑ Easier testing

---

3. Remove Duplicate AuthContext

Action Plan:

```bash
# Delete: src/contexts/AuthContext.tsx
# Keep: src/core/auth/AuthContext.tsx
# Update: All imports to use core/auth
```

Migration:

```typescript
// Before
import { useAuth } from '@/contexts/AuthContext';

// After
import { useAuth } from '@/core/auth';
```

11.2 Short-term Actions (High Priority)

4. Implement Data Access Layer

Pattern:

```typescript
// src/data/
//   queries/
//     parts.queries.ts
//   mutations/
//     parts.mutations.ts

// Example:
export const usePartsQuery = () => {
  return useQuery({
    queryKey: ['parts'],
    queryFn: () => partsService.getAll(),
    staleTime: 5 * 60 * 1000, // 5 minutes
  });
};

export const useCreatePartMutation = () => {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: (data) => partsService.create(data),
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['parts'] });
    },
  });
```

```
};
```

---

5. Add Validation Schemas

Using Zod:

```typescript
// src/schemas/parts.schema.ts
import { z } from 'zod';

export const partSchema = z.object({
  partNumber: z.string().min(1, 'Part number required'),
  partName: z.string().min(1, 'Part name required'),
  hsnCode: z.string().regex(/^\d{8}$/, 'Invalid HSN code'),
  unitPrice: z.number().positive('Price must be positive'),
  currentQty: z.number().int().nonnegative('Quantity cannot be negative'),
});

export type PartFormData = z.infer<typeof partSchema>;
```

---

6. Normalize Data Structure

Using normalizr or custom solution:

```typescript
```

```ts
// src/store/entities.store.ts
import { create } from 'zustand';

interface EntitiesState {
  parts: Record<string, Part>;
  customers: Record<string, Customer>;
  vehicles: Record<string, Vehicle>;

  addPart: (part: Part) => void;
  addParts: (parts: Part[]) => void;
  removePart: (id: string) => void;
}

export const useEntitiesStore = create<EntitiesState>((set) => ({
  parts: {},
  customers: {},
  vehicles: {},

  addPart: (part) => set((state) => ({
    parts: { ...state.parts, [part.id]: part }
  })),

  addParts: (parts) => set((state) => ({
    parts: {
      ...state.parts,
      ...Object.fromEntries(parts.map(p => [p.id, p]))
    }
  })),
```

```
  removePart: (id) => set((state) => {
    const { [id]: removed, ...rest } = state.parts;
    return { parts: rest };
  }),
}));
```

11.3 Long-term Actions (Medium Priority)

7. Implement Repository Pattern Properly

For Real API (not just mocks):

```typescript
// src/repositories/parts.repository.ts
import { apiClient } from '@/core/api';
import type { Part, CreatePartDTO, UpdatePartDTO } from '@/types';

export class PartsRepository {
  private readonly basePath = '/parts';

  async findAll(): Promise<Part[]> {
    const response = await apiClient.get<Part[]>(this.basePath);
    return response.data;
  }

  async findById(id: string): Promise<Part> {
    const response = await apiClient.get<Part>(`${this.basePath}/${id}`);
```

```
    return response.data;
  }

  async create(data: CreatePartDTO): Promise<Part> {
    const response = await apiClient.post<Part>(this.basePath, data);
    return response.data;
  }

  async update(id: string, data: UpdatePartDTO): Promise<Part> {
    const response = await apiClient.put<Part>(`${this.basePath}/${id}`, data);
    return response.data;
  }

  async delete(id: string): Promise<void> {
    await apiClient.delete(`${this.basePath}/${id}`);
  }
}

export const partsRepository = new PartsRepository();
```

---

8. Add Error Boundaries

```typescript
// src/components/ErrorBoundary.tsx
'use client';
```

```tsx
import React from 'react';

interface Props {
  children: React.ReactNode;
  fallback?: React.ReactNode;
}

export class ErrorBoundary extends React.Component<Props, { hasError: boolean }> {
  constructor(props: Props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError() {
    return { hasError: true };
  }

  componentDidCatch(error: Error, errorInfo: React.ErrorInfo) {
    console.error('Error caught by boundary:', error, errorInfo);
    // Send to error reporting service
  }

  render() {
    if (this.state.hasError) {
      return this.props.fallback || <div>Something went wrong</div>;
    }

    return this.props.children;
  }
```

}
```

---

#### **9. Implement Code Splitting**

```typescript
// Lazy load heavy components
const PartsManager = lazy(() => import('@/features/parts/PartsManager'));
const InventoryDashboard = lazy(() => import('@/features/inventory/Dashboard'));

// In component:
<Suspense fallback={<LoadingSpinner />}>
  <PartsManager />
</Suspense>
```

---

12. Migration Strategy

Phase 1: Foundation (Week 1-2)

- [ ] Remove duplicate AuthContext
- [ ] Consolidate to single API client
- [ ] Install TanStack Query
- [ ] Set up MSW for mocking

Phase 2: State Management (Week 3-4)
```

- [ ] Migrate auth state to Zustand only
- [ ] Remove Context API for state
- [ ] Implement TanStack Query for server state
- [ ] Add validation schemas (Zod)

Phase 3: Data Layer (Week 5-6)

- [ ] Implement repository pattern for all entities
- [ ] Create query/mutation hooks
- [ ] Normalize data structure
- [ ] Add error boundaries

Phase 4: Optimization (Week 7-8)

- [ ] Implement code splitting
- [ ] Add request deduplication
- [ ] Optimize re-renders
- [ ] Performance testing

Phase 5: Testing & Documentation (Week 9-10)

- [ ] Unit tests for repositories
- [ ] Integration tests for data flow
- [ ] Update documentation
- [ ] Code review and refactoring

13. Architectural Diagrams

13.1 Proposed Architecture

```mermaid
graph TB
    subgraph "Presentation Layer"
        A[Components]
        B[Pages]
    end

    subgraph "Data Access Layer"
        C[TanStack Query Hooks]
        D[Zustand Stores]
    end

    subgraph "Business Logic Layer"
        E[Services]
        F[Repositories]
    end

    subgraph "Infrastructure Layer"
        G[API Client]
        H[MSW Handlers]
        I[Backend API]
    end

    A --> C
    A --> D
    B --> C
    B --> D
    C --> E
    E --> F
```

```
    F --> G
    G --> H
    G --> I

    style C fill:#9f9,stroke:#333
    style D fill:#9f9,stroke:#333
    style G fill:#99f,stroke:#333
```

13.2 Proposed Data Flow

```mermaid
sequenceDiagram
    participant Component
    participant Query Hook
    participant Service
    participant Repository
    participant API Client
    participant Cache

    Component->>Query Hook: usePartsQuery()
    Query Hook->>Cache: Check cache

    alt Cache Hit
        Cache-->>Query Hook: Return cached data
        Query Hook-->>Component: Render with data
    else Cache Miss
```

```
    Query Hook->>Service: getParts()
    Service->>Repository: findAll()
    Repository->>API Client: GET /parts
    API Client-->>Repository: Response
    Repository-->>Service: Parts[]
    Service-->>Query Hook: Parts[]
    Query Hook->>Cache: Store in cache
    Query Hook-->>Component: Render with data
  end

  Note over Query Hook,Cache: Background refetch after staleTime
```

---

14. Conclusion

Summary of Critical Issues

1. **🔴 **Dual State Management**** - Context API + Zustand creating confusion
2. **🔴 **Dual API Clients**** - Two separate implementations with different features
3. **🔴 **Duplicate AuthContext**** - Two identical implementations
4. **🟡 **500+ useState**** - No centralized state for domain data
5. **🟡 **Mock/Real Pattern**** - Inconsistent implementation across services
6. **🟡 **No Data Normalization**** - Nested, denormalized data structures
7. **🟡 **No Validation**** - Missing input validation schemas
8. **🟡 **No Error Boundaries**** - Ad-hoc error handling

Estimated Effort

Phase	Duration	Complexity
Phase 1: Foundation	2 weeks	Medium
Phase 2: State Management	2 weeks	High
Phase 3: Data Layer	2 weeks	High
Phase 4: Optimization	2 weeks	Medium
Phase 5: Testing	2 weeks	Medium
Total	**10 weeks**	**High**

Risk Assessment

Risk	Probability	Impact	Mitigation
Breaking existing features	High	High	Incremental migration, feature flags
Team learning curve	Medium	Medium	Training sessions, pair programming
Timeline overrun	Medium	High	Buffer time, prioritize critical issues
Data loss during migration	Low	Critical	Backup strategy, rollback plan

---

15. Next Steps

Immediate Actions (This Week)

1. **Review this report** with the development team
2. **Prioritize issues** based on business impact
3. **Create detailed tickets** for Phase 1 tasks

4. **Set up project board** for tracking migration

5. **Schedule kickoff meeting** for migration project

Questions to Answer

1. What is the target timeline for backend integration?

2. Are there any features currently in development that would conflict?

3. What is the testing strategy during migration?

4. Who will be responsible for each phase?

5. What is the rollback strategy if issues arise?

Report Generated By: Antigravity AI

Analysis Depth: Root-level architectural analysis

Codebase Version: Current (as of December 19, 2025)

Total Files Analyzed: 500+ files

Total Lines of Code: ~50,000+ LOC

Appendix A: File Structure Reference

```
src/
├── __mocks__/
|   ├── data/ (23 files)
|   └── repositories/ (2 files)
├── app/ (routing)
```

```
├── components/ (74 files)
├── config/
│   └── api.config.ts
├── contexts/ (7 files) ⚠️ DUPLICATE
├── core/
│   ├── api/
│   │   └── client.ts ⚠️ DUPLICATE
│   └── auth/
│       └── AuthContext.tsx ⚠️ DUPLICATE
├── features/ (43 files)
├── hooks/ (8 files)
├── lib/
│   └── api/
│       └── real-client.ts ⚠️ DUPLICATE
├── services/ (14 files)
├── shared/
│   ├── components/ (71 files)
│   ├── constants/ (7 files)
│   ├── hooks/ (14 files)
│   ├── types/ (20 files)
│   └── utils/ (18 files)
└── store/ (2 files)
```

Appendix B: Dependencies Analysis

Current State Management Dependencies

```json
{
  "zustand": "^4.x",
  "react": "^18.x"
}
```

Recommended Additional Dependencies

```json
{
  "@tanstack/react-query": "^5.x",
  "zod": "^3.x",
  "msw": "^2.x"
}
```

Dependencies to Remove

- None (Context API is built-in to React)

Appendix C: Code Metrics

Metric	Value
Total Files	500+
Total Components	150+
useState Instances	500+

| Service Files | 14 |

| Mock Data Files | 23 |

| Type Definition Files | 20 |

| Utility Files | 18 |

| API Clients | 2 ⚠️ |

| AuthContext Implementations | 2 ⚠️ |

| State Management Systems | 2 ⚠️ |

****END OF REPORT****