# RAS Dashboard Development Patterns Guide

## Table of Contents

1. [Backend Architecture](#backend-architecture)
2. [Frontend Architecture](#frontend-architecture)
3. [API Communication](#api-communication)
4. [UI/UX Design Patterns](#uiux-design-patterns)
5. [Database Patterns](#database-patterns)
6. [Security Patterns](#security-patterns)
7. [Testing Patterns](#testing-patterns)

## Backend Architecture

### 1. Service Layer Pattern

**Location**: api/src/services/

**Structure**:

const { db } = require('../db');  
const { tableName } = require('../db/schema/schemaFile');  
const { eq, and, sql, desc, asc } = require('drizzle-orm');  
  
class ServiceName {  
 // CRUD Operations  
 async getAll(filters = {}, options = {}) {  
 // Implementation with pagination, filtering, sorting  
 }  
  
 async getById(id) {  
 // Single record retrieval  
 }  
  
 async create(data) {  
 // Create with validation and error handling  
 }  
  
 async update(id, data) {  
 // Update with validation  
 }  
  
 async delete(id) {  
 // Soft delete preferred  
 }  
  
 // Business Logic Methods  
 async customBusinessMethod(params) {  
 // Complex business operations  
 }  
}  
  
module.exports = new ServiceName();

**Key Patterns**: - Always use Drizzle ORM for database operations - Import db from ../db (not ../db/connection) - Use consistent error handling with try/catch - Return structured responses: { success: boolean, data: any, message: string } - Implement pagination with limit and offset - Use transactions for multi-table operations

### 2. Controller Layer Pattern

**Location**: api/src/controllers/

**Structure**:

const serviceName = require('../services/serviceNameService');  
const { validationResult } = require('express-validator');  
  
class ControllerName {  
 async getAll(req, res) {  
 try {  
 const filters = {  
 // Extract query parameters  
 };  
 const options = {  
 limit: parseInt(req.query.limit) || 50,  
 offset: parseInt(req.query.offset) || 0  
 };  
  
 const result = await serviceName.getAll(filters, options);  
   
 res.status(200).json({  
 success: true,  
 data: result.data,  
 pagination: result.pagination,  
 message: 'Records retrieved successfully'  
 });  
 } catch (error) {  
 console.error('Error in getAll:', error);  
 res.status(500).json({  
 success: false,  
 message: error.message || 'Failed to retrieve records'  
 });  
 }  
 }  
  
 async create(req, res) {  
 try {  
 // Validation check  
 const errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({  
 success: false,  
 message: 'Validation failed',  
 errors: errors.array()  
 });  
 }  
  
 const result = await serviceName.create(req.body);  
   
 res.status(201).json({  
 success: true,  
 data: result.data,  
 message: result.message  
 });  
 } catch (error) {  
 console.error('Error in create:', error);  
 res.status(500).json({  
 success: false,  
 message: error.message || 'Failed to create record'  
 });  
 }  
 }  
}  
  
module.exports = new ControllerName();

**Key Patterns**: - Always validate input using express-validator - Consistent error handling and logging - Standard HTTP status codes (200, 201, 400, 404, 500) - Structured JSON responses - Extract and validate query parameters

### 3. Routes Layer Pattern

**Location**: api/src/routes/

**Structure**:

const express = require('express');  
const { body, param, query } = require('express-validator');  
const controllerName = require('../controllers/controllerNameController');  
const { authenticateToken } = require('../middleware/auth');  
const { requirePermission } = require('../middleware/rbac');  
  
const router = express.Router();  
  
// Validation middleware  
const validateCreate = [  
 body('field1').notEmpty().withMessage('Field1 is required'),  
 body('field2').isEmail().withMessage('Valid email required')  
];  
  
// Apply authentication to all routes  
router.use(authenticateToken);  
router.use(requirePermission('module\_name:read'));  
  
// Routes  
router.get('/', controllerName.getAll);  
router.get('/:id', controllerName.getById);  
router.post('/',   
 requirePermission('module\_name:write'),  
 validateCreate,   
 controllerName.create  
);  
router.put('/:id',   
 requirePermission('module\_name:write'),  
 validateCreate,   
 controllerName.update  
);  
router.delete('/:id',   
 requirePermission('module\_name:delete'),  
 controllerName.delete  
);  
  
module.exports = router;

**Key Patterns**: - Use express-validator for input validation - Apply authentication with authenticateToken - Use RBAC with requirePermission('module:action') - RESTful route naming conventions - Group validation middleware logically

## Frontend Architecture

### 1. API Utilities Pattern

**Location**: client/src/utils/

**Structure**:

const API\_BASE\_URL = 'http://localhost:3001/api/v1';  
  
// Get auth token from localStorage  
const getAuthToken = () => {  
 return localStorage.getItem('accessToken');  
};  
  
// Create headers with auth token  
const createHeaders = () => {  
 const token = getAuthToken();  
 return {  
 'Authorization': `Bearer ${token}`,  
 'Content-Type': 'application/json',  
 };  
};  
  
// Handle API response  
const handleResponse = async (response) => {  
 if (!response.ok) {  
 const errorData = await response.json().catch(() => ({}));  
 throw new Error(errorData.message || `HTTP error! status: ${response.status}`);  
 }  
 return await response.json();  
};  
  
export const apiName = {  
 async getAll(filters = {}) {  
 try {  
 const params = new URLSearchParams();  
 Object.entries(filters).forEach(([key, value]) => {  
 if (value) params.append(key, value);  
 });  
  
 const response = await fetch(`${API\_BASE\_URL}/endpoint?${params}`, {  
 method: 'GET',  
 headers: createHeaders(),  
 });  
 return await handleResponse(response);  
 } catch (error) {  
 throw new Error(error.message || 'Failed to fetch records');  
 }  
 },  
  
 async create(data) {  
 try {  
 const response = await fetch(`${API\_BASE\_URL}/endpoint`, {  
 method: 'POST',  
 headers: createHeaders(),  
 body: JSON.stringify(data)  
 });  
 return await handleResponse(response);  
 } catch (error) {  
 throw new Error(error.message || 'Failed to create record');  
 }  
 }  
};

**Key Patterns**: - **NEVER use axios** - Always use native fetch API - Use accessToken from localStorage (not token) - Consistent error handling with try/catch - Use URLSearchParams for query parameters - Standard headers with Authorization and Content-Type

### 2. Component Structure Pattern

**Location**: client/src/pages/[module]/components/

**Main Page Structure**:

import React, { useState, useEffect } from "react";  
import { toast } from "react-toastify";  
import {  
 Block,  
 BlockHead,  
 BlockHeadContent,  
 BlockTitle,  
 BlockDes,  
 Icon,  
} from "@/components/Component";  
import Content from "@/layout/content/Content";  
import Head from "@/layout/head/Head";  
  
// Component imports  
import DataTable from "./components/DataTable";  
import DetailsPanel from "./components/DetailsPanel";  
import { apiName } from "@/utils/apiName";  
  
const MainComponent = () => {  
 // State management  
 const [data, setData] = useState([]);  
 const [loading, setLoading] = useState(true);  
 const [selectedItem, setSelectedItem] = useState(null);  
 const [panelOpen, setPanelOpen] = useState(false);  
  
 // API functions  
 const fetchData = async () => {  
 try {  
 setLoading(true);  
 const response = await apiName.getAll();  
 if (response.success) {  
 setData(response.data);  
 }  
 } catch (error) {  
 console.error('Error fetching data:', error);  
 toast.error('Failed to fetch data');  
 } finally {  
 setLoading(false);  
 }  
 };  
  
 // Event handlers  
 const handleView = (item) => {  
 setSelectedItem(item);  
 setPanelOpen(true);  
 };  
  
 // Effects  
 useEffect(() => {  
 fetchData();  
 }, []);  
  
 return (  
 <React.Fragment>  
 <Head title="Page Title"></Head>  
 <Content>  
 <BlockHead size="sm">  
 <BlockHeadContent>  
 <BlockTitle page>Page Title</BlockTitle>  
 <BlockDes className="text-soft">  
 <p>Page description</p>  
 </BlockDes>  
 </BlockHeadContent>  
 </BlockHead>  
  
 <Block>  
 <DataTable  
 data={data}  
 loading={loading}  
 onView={handleView}  
 />  
 </Block>  
  
 <DetailsPanel  
 isOpen={panelOpen}  
 onClose={() => setPanelOpen(false)}  
 data={selectedItem}  
 />  
 </Content>  
 </React.Fragment>  
 );  
};  
  
export default MainComponent;

**Key Patterns**: - Use functional components with hooks - Consistent state naming conventions - Error handling with toast notifications - Separate API calls into dedicated functions - Use React.Fragment for root elements

## UI/UX Design Patterns

### 1. Data Table Pattern

**IMPORTANT**: All datatables must follow this consistent structure and styling.

**Component**: DataTable or CustomDataTable

**Required Elements**: 1. **Bulk Actions** (left side) 2. **Export Buttons** (right side, colored icons) 3. **Show Entries Dropdown** (right side) 4. **Search Toggle** (right side) 5. **Sortable Column Headers** (with up/down arrows) 6. **Consistent Row Selection** (checkboxes) 7. **Actions Dropdown** (three dots menu)

**Complete DataTable Structure**:

import React, { useState, useMemo } from "react";  
import {  
 UncontrolledDropdown,  
 DropdownMenu,  
 DropdownToggle,  
 DropdownItem,  
 Button,  
 Row,  
 Col,  
} from "reactstrap";  
import {  
 Icon,  
 DataTable,  
 DataTableBody,  
 DataTableHead,  
 DataTableRow,  
 DataTableItem,  
 UserAvatar,  
 PaginationComponent,  
 RSelect,  
} from "@/components/Component";  
import "./DataTable.css";  
  
const CustomDataTable = ({ data, columns, loading, onView, onEdit, onDelete }) => {  
 // State management  
 const [selectedRows, setSelectedRows] = useState([]);  
 const [sortField, setSortField] = useState("");  
 const [sortDirection, setSortDirection] = useState("asc");  
 const [currentPage, setCurrentPage] = useState(1);  
 const [itemPerPage, setItemPerPage] = useState(50);  
 const [onSearch, setOnSearch] = useState(true);  
  
 // Sorting functionality  
 const handleSort = (field) => {  
 if (typeof field === 'function') {  
 const fieldName = field.toString().match(/row\.(\w+)/)?.[1] || '';  
 if (sortField === fieldName) {  
 setSortDirection(sortDirection === 'asc' ? 'desc' : 'asc');  
 } else {  
 setSortField(fieldName);  
 setSortDirection('asc');  
 }  
 }  
 };  
  
 // Export functionality - REQUIRED IMPLEMENTATION  
 const handleExport = (format) => {  
 // Prepare data for export - map your data fields to readable column names  
 const exportData = data.map(row => ({  
 // Map your data fields here with human-readable column names  
 'ID': row.id,  
 'Name': row.name,  
 'Status': row.status,  
 'Created Date': row.createdAt ? new Date(row.createdAt).toLocaleDateString() : 'N/A',  
 // Add all relevant fields that should be included in exports  
 }));  
  
 // Get current date for filename  
 const dateStr = new Date().toISOString().split('T')[0];  
 const fileName = `export\_${dateStr}`;  
  
 switch (format) {  
 case 'csv':  
 // CSV Export - REQUIRED  
 const csvContent = [  
 Object.keys(exportData[0]).join(','),  
 ...exportData.map(row => Object.values(row).join(','))  
 ].join('\n');  
  
 const blob = new Blob([csvContent], { type: 'text/csv' });  
 const url = window.URL.createObjectURL(blob);  
 const a = document.createElement('a');  
 a.href = url;  
 a.download = `${fileName}.csv`;  
 a.click();  
 window.URL.revokeObjectURL(url);  
 break;  
   
 case 'excel':  
 // Excel Export - REQUIRED  
 // Use export-from-json library  
 import('export-from-json').then(module => {  
 const exportFromJSON = module.default;  
 exportFromJSON({  
 data: exportData,  
 fileName,  
 exportType: 'xls'  
 });  
 }).catch(error => {  
 console.error('Error exporting to Excel:', error);  
 });  
 break;  
   
 case 'pdf':  
 // PDF Export - REQUIRED  
 // Create a printable HTML table for PDF export  
 const printWindow = window.open('', '\_blank');  
   
 if (!printWindow) {  
 alert('Please allow pop-ups to export as PDF');  
 return;  
 }  
   
 // Create a styled HTML table  
 const tableHTML = `  
 <!DOCTYPE html>  
 <html>  
 <head>  
 <title>${fileName}</title>  
 <style>  
 body { font-family: Arial, sans-serif; }  
 table { width: 100%; border-collapse: collapse; margin-bottom: 20px; }  
 th, td { padding: 8px; text-align: left; border-bottom: 1px solid #ddd; }  
 th { background-color: #f2f2f2; font-weight: bold; }  
 .header { margin-bottom: 20px; }  
 .header h1 { margin-bottom: 5px; }  
 .header p { color: #666; margin-top: 0; }  
 @media print {  
 button { display: none; }  
 }  
 </style>  
 </head>  
 <body>  
 <div class="header">  
 <h1>Data Export</h1>  
 <p>Generated on ${new Date().toLocaleString()}</p>  
 </div>  
 <button onclick="window.print();window.close();" style="padding: 10px; margin-bottom: 20px; cursor: pointer;">  
 Print as PDF  
 </button>  
 <table>  
 <thead>  
 <tr>  
 ${Object.keys(exportData[0]).map(key => `<th>${key}</th>`).join('')}  
 </tr>  
 </thead>  
 <tbody>  
 ${exportData.map(row => `  
 <tr>  
 ${Object.values(row).map(value => `<td>${value}</td>`).join('')}  
 </tr>  
 `).join('')}  
 </tbody>  
 </table>  
 <script>  
 // Auto-trigger print dialog after a short delay  
 setTimeout(() => {  
 window.print();  
 }, 500);  
 </script>  
 </body>  
 </html>  
 `;  
   
 printWindow.document.write(tableHTML);  
 printWindow.document.close();  
 break;  
 }  
 };  
  
 // Apply sorting  
 const sortedData = useMemo(() => {  
 if (!sortField) return data;  
 return [...data].sort((a, b) => {  
 let aValue = a[sortField];  
 let bValue = b[sortField];  
  
 if (typeof aValue === 'string') {  
 aValue = aValue.toLowerCase();  
 bValue = bValue?.toLowerCase() || '';  
 }  
  
 if (aValue < bValue) return sortDirection === 'asc' ? -1 : 1;  
 if (aValue > bValue) return sortDirection === 'asc' ? 1 : -1;  
 return 0;  
 });  
 }, [data, sortField, sortDirection]);  
  
 // Pagination  
 const indexOfLastItem = currentPage \* itemPerPage;  
 const indexOfFirstItem = indexOfLastItem - itemPerPage;  
 const currentItems = sortedData.slice(indexOfFirstItem, indexOfLastItem);  
  
 return (  
 <DataTable className="card-stretch">  
 <div className="card-inner position-relative card-tools-toggle">  
 <div className="card-title-group">  
 {/\* Left side - Bulk Actions \*/}  
 <div className="card-tools">  
 <div className="form-inline flex-nowrap gx-3">  
 <div className="form-wrap">  
 <RSelect  
 options={[  
 { value: "export", label: "Export Selected" },  
 { value: "delete", label: "Delete Selected" },  
 ]}  
 className="w-130px"  
 placeholder="Bulk Action"  
 />  
 </div>  
 <div className="btn-wrap">  
 <Button  
 disabled={selectedRows.length === 0}  
 color="light"  
 outline  
 className="btn-dim"  
 >  
 Apply  
 </Button>  
 </div>  
 </div>  
 </div>  
  
 {/\* Right side - Export, Show, Search, Filter \*/}  
 <div className="card-tools me-n1">  
 <ul className="btn-toolbar gx-1">  
 {/\* Export Section \*/}  
 <li className="export-section">  
 <span className="export-label">Export</span>  
 <Button  
 color="light"  
 size="sm"  
 className="btn-icon export-excel"  
 onClick={() => handleExport('excel')}  
 title="Export to Excel"  
 >  
 <Icon name="file-xls" className="export-excel"></Icon>  
 </Button>  
 <Button  
 color="light"  
 size="sm"  
 className="btn-icon export-pdf"  
 onClick={() => handleExport('pdf')}  
 title="Export to PDF"  
 >  
 <Icon name="file-pdf" className="export-pdf"></Icon>  
 </Button>  
 <Button  
 color="light"  
 size="sm"  
 className="btn-icon export-csv"  
 onClick={() => handleExport('csv')}  
 title="Export to CSV"  
 >  
 <Icon name="file-text" className="export-csv"></Icon>  
 </Button>  
 </li>  
 <li className="btn-toolbar-sep"></li>  
  
 {/\* Show entries \*/}  
 <li className="show-entries">  
 <span className="show-label">Show</span>  
 <RSelect  
 options={[  
 { value: 10, label: "10" },  
 { value: 25, label: "25" },  
 { value: 50, label: "50" },  
 { value: 100, label: "100" },  
 ]}  
 value={{ value: itemPerPage, label: itemPerPage.toString() }}  
 onChange={(selected) => setItemPerPage(selected.value)}  
 className="w-80px"  
 />  
 </li>  
 <li className="btn-toolbar-sep"></li>  
  
 {/\* Search toggle \*/}  
 <li>  
 <a  
 href="#search"  
 onClick={(ev) => {  
 ev.preventDefault();  
 setOnSearch(!onSearch);  
 }}  
 className="btn btn-icon search-toggle toggle-search"  
 >  
 <Icon name="search"></Icon>  
 </a>  
 </li>  
 </ul>  
 </div>  
 </div>  
 </div>  
  
 {/\* Table Body \*/}  
 <DataTableBody compact>  
 <DataTableHead>  
 {/\* Select All Checkbox \*/}  
 <DataTableRow className="nk-tb-col-check">  
 <div className="custom-control custom-control-sm custom-checkbox notext">  
 <input  
 type="checkbox"  
 className="custom-control-input"  
 onChange={(e) => handleSelectAll(e.target.checked)}  
 id="uid\_all"  
 />  
 <label className="custom-control-label" htmlFor="uid\_all"></label>  
 </div>  
 </DataTableRow>  
  
 {/\* Column Headers \*/}  
 {columns.map((column, index) => (  
 <DataTableRow key={index}>  
 {column.sortable ? (  
 <div  
 className="d-flex align-items-center cursor-pointer sortable-header"  
 onClick={() => handleSort(column.selector)}  
 >  
 {column.name}  
 </div>  
 ) : (  
 <span className="sub-text">{column.name}</span>  
 )}  
 </DataTableRow>  
 ))}  
 </DataTableHead>  
  
 {/\* Table Rows \*/}  
 {currentItems.map((item) => (  
 <DataTableItem key={item.id}>  
 {/\* Row Checkbox \*/}  
 <DataTableRow className="nk-tb-col-check">  
 <div className="custom-control custom-control-sm custom-checkbox notext">  
 <input  
 type="checkbox"  
 className="custom-control-input"  
 checked={selectedRows.some(row => row.id === item.id)}  
 onChange={(e) => handleRowSelect(item, e.target.checked)}  
 id={`uid\_${item.id}`}  
 />  
 <label className="custom-control-label" htmlFor={`uid\_${item.id}`}></label>  
 </div>  
 </DataTableRow>  
  
 {/\* Data Cells \*/}  
 {columns.map((column, index) => (  
 <DataTableRow key={index}>  
 {column.cell ? column.cell(item) : item[column.selector(item)]}  
 </DataTableRow>  
 ))}  
 </DataTableItem>  
 ))}  
 </DataTableBody>  
  
 {/\* Pagination \*/}  
 <div className="card-inner">  
 <PaginationComponent  
 itemPerPage={itemPerPage}  
 totalItems={sortedData.length}  
 paginate={setCurrentPage}  
 currentPage={currentPage}  
 />  
 </div>  
 </DataTable>  
 );  
};

**Standard Column Definitions**:

// Column structure with sorting  
const columns = [  
 {  
 name: (  
 <div className="d-flex align-items-center">  
 <span style={{ fontSize: '0.875rem', fontWeight: '500' }}>Name</span>  
 <div className="ms-1 d-flex flex-column">  
 <Icon name="chevron-up" style={{ fontSize: '0.6rem', color: '#8094ae', cursor: 'pointer' }}></Icon>  
 <Icon name="chevron-down" style={{ fontSize: '0.6rem', color: '#8094ae', cursor: 'pointer' }}></Icon>  
 </div>  
 </div>  
 ),  
 selector: (row) => row.name,  
 sortable: true,  
 grow: 2,  
 cell: (row) => (  
 <div className="user-card mt-2 mb-2">  
 <UserAvatar  
 theme="primary"  
 text={row.name?.substring(0, 2).toUpperCase()}  
 />  
 <div className="user-info">  
 <span className="tb-lead" style={{ fontSize: '0.875rem', fontWeight: '500' }}>  
 {row.name} {/\* Bold label \*/}  
 </span>  
 <div className="text-soft" style={{ fontSize: '0.75rem', fontWeight: 'normal' }}>  
 {row.id} {/\* Unbolded data \*/}  
 </div>  
 </div>  
 </div>  
 ),  
 },  
 {  
 name: (  
 <div className="d-flex align-items-center">  
 <span style={{ fontSize: '0.875rem', fontWeight: '500' }}>Status</span>  
 <div className="ms-1 d-flex flex-column">  
 <Icon name="chevron-up" style={{ fontSize: '0.6rem', color: '#8094ae', cursor: 'pointer' }}></Icon>  
 <Icon name="chevron-down" style={{ fontSize: '0.6rem', color: '#8094ae', cursor: 'pointer' }}></Icon>  
 </div>  
 </div>  
 ),  
 selector: (row) => row.status,  
 sortable: true,  
 cell: (row) => (  
 <span className={`badge badge-dim ${getStatusBadgeColor(row.status)}`}>  
 {row.status}  
 </span>  
 ),  
 },  
 {  
 name: "Actions",  
 cell: (row) => (  
 <UncontrolledDropdown>  
 <DropdownToggle tag="a" className="dropdown-toggle btn btn-icon btn-trigger">  
 <Icon name="more-h"></Icon>  
 </DropdownToggle>  
 <DropdownMenu end>  
 <ul className="link-list-opt no-bdr">  
 <li>  
 <DropdownItem onClick={() => onView(row)}>  
 <Icon name="eye"></Icon>  
 <span>View Details</span>  
 </DropdownItem>  
 </li>  
 <li>  
 <DropdownItem onClick={() => onEdit(row)}>  
 <Icon name="edit"></Icon>  
 <span>Edit</span>  
 </DropdownItem>  
 </li>  
 </ul>  
 </DropdownMenu>  
 </UncontrolledDropdown>  
 ),  
 allowOverflow: true,  
 button: true,  
 },  
];

**Required CSS File** (DataTable.css):

/\* DataTable Consistency Styles \*/  
  
/\* Export button styling \*/  
.export-section {  
 display: flex;  
 align-items: center;  
 gap: 0.5rem;  
}  
  
.export-section .export-label {  
 font-size: 0.875rem;  
 font-weight: 500;  
 color: #526484;  
 margin-right: 0.25rem;  
}  
  
.export-section .btn-icon {  
 margin-right: 0.25rem;  
 border: none;  
 background: transparent;  
 padding: 0.375rem;  
 border-radius: 0.25rem;  
 transition: all 0.2s ease;  
}  
  
/\* Export button colors - REQUIRED \*/  
.export-excel { color: #28a745 !important; }  
.export-pdf { color: #dc3545 !important; }  
.export-csv { color: #007bff !important; }  
  
/\* Hover effects \*/  
.btn-icon.export-excel:hover { background-color: rgba(40, 167, 69, 0.1) !important; }  
.btn-icon.export-pdf:hover { background-color: rgba(220, 53, 69, 0.1) !important; }  
.btn-icon.export-csv:hover { background-color: rgba(0, 123, 255, 0.1) !important; }  
  
/\* Show entries styling \*/  
.show-entries {  
 display: flex;  
 align-items: center;  
 gap: 0.5rem;  
}  
  
.show-entries .show-label {  
 font-size: 0.875rem;  
 font-weight: 500;  
 color: #526484;  
}  
  
/\* Sortable column headers \*/  
.sortable-header {  
 cursor: pointer;  
 user-select: none;  
 transition: color 0.2s ease;  
}  
  
.sortable-header:hover {  
 color: #007bff;  
}  
  
/\* Responsive design \*/  
@media (max-width: 768px) {  
 .export-section .export-label,  
 .show-entries .show-label {  
 display: none;  
 }  
}

**Key Patterns**: - **ALWAYS** include export buttons (Excel=green, PDF=red, CSV=blue) - **ALWAYS** include Show entries dropdown (10, 25, 50, 100) - **ALWAYS** make columns sortable with up/down arrows - **ALWAYS** include bulk actions on the left - **ALWAYS** include search toggle - **ALWAYS** use consistent font sizes and weights - **ALWAYS** include row selection checkboxes - **ALWAYS** use Actions dropdown with three dots

### 2. Slide-Out Panel Pattern

**Component**: SlideOutPanel from @/components/partials/SlideOutPanel

**🎨 IMPORTANT: Use Unified CSS Architecture** - **ALWAYS** import the shared CSS file: import "./AssetSlideOutPanels.css" - **NEVER** create individual CSS files for panels - **USE** the standardized CSS classes provided in the shared file

**Structure**:

import React, { useState, useEffect } from 'react';  
import {  
 Button,  
 Form,  
 FormGroup,  
 Label,  
 Input,  
 Row,  
 Col,  
 Badge,  
 Alert,  
 Card,  
 CardBody,  
 CardHeader  
} from 'reactstrap';  
import { Icon } from '@/components/Component';  
import SlideOutPanel from '@/components/partials/SlideOutPanel';  
import { toast } from 'react-toastify';  
import "./AssetSlideOutPanels.css"; // 🎨 REQUIRED: Shared CSS file  
  
const CustomPanel = ({ isOpen, onClose, data, onUpdate }) => {  
 const [loading, setLoading] = useState(false);  
 const [formData, setFormData] = useState({});  
 const [isEditing, setIsEditing] = useState(false);  
  
 useEffect(() => {  
 if (isOpen && data) {  
 setFormData(data);  
 }  
 }, [isOpen, data]);  
  
 const handleSave = async () => {  
 try {  
 setLoading(true);  
 // API call  
 toast.success('Updated successfully');  
 setIsEditing(false);  
 if (onUpdate) onUpdate();  
 } catch (error) {  
 toast.error('Failed to update');  
 } finally {  
 setLoading(false);  
 }  
 };  
  
 return (  
 <SlideOutPanel  
 isOpen={isOpen}  
 onClose={onClose}  
 title="Panel Title"  
 subtitle={data?.name || 'Item'}  
 size="md"  
 >  
 <div className="asset-panel-container"> {/\* 🎨 Use shared container class \*/}  
 {loading && (  
 <div className="panel-loading"> {/\* 🎨 Use shared loading class \*/}  
 <Spinner color="primary" />  
 <p className="panel-text-muted">Loading...</p> {/\* 🎨 Use shared text class \*/}  
 </div>  
 )}  
  
 {!loading && (  
 <div>  
 {/\* Header with action button \*/}  
 <div className="d-flex justify-content-between align-items-center mb-4">  
 <h5 className="mb-0">  
 <Icon name="icon-name" className="me-2"></Icon>  
 Section Title  
 </h5>  
 <Button  
 color="primary"  
 size="sm"  
 onClick={() => setIsEditing(!isEditing)}  
 >  
 <Icon name="edit" className="me-1"></Icon>  
 {isEditing ? 'Cancel' : 'Edit'}  
 </Button>  
 </div>  
  
 {/\* Form Section \*/}  
 <Form className="panel-form"> {/\* 🎨 Use shared form class \*/}  
 <Row>  
 <Col md={6}>  
 <FormGroup>  
 <Label for="field1">Field Label</Label>  
 <Input  
 type="text"  
 name="field1"  
 id="field1"  
 value={formData.field1 || ''}  
 onChange={(e) => setFormData({...formData, field1: e.target.value})}  
 disabled={!isEditing}  
 />  
 </FormGroup>  
 </Col>  
 </Row>  
  
 {isEditing && (  
 <div className="d-flex gap-2 mt-3">  
 <Button  
 color="primary"  
 onClick={handleSave}  
 disabled={loading}  
 >  
 <Icon name="check" className="me-1"></Icon>  
 Save Changes  
 </Button>  
 <Button  
 color="secondary"  
 onClick={() => setIsEditing(false)}  
 >  
 <Icon name="cross" className="me-1"></Icon>  
 Cancel  
 </Button>  
 </div>  
 )}  
 </Form>  
 </div>  
 )}  
 </div>  
 </SlideOutPanel>  
 );  
};

**Key Patterns**: - Import from @/components/partials/SlideOutPanel - **ALWAYS** import shared CSS: import "./AssetSlideOutPanels.css" - Use size="md" for standard panels, size="lg" for complex forms - **NEVER** create individual CSS files for panels - Use shared CSS classes for consistency

**🎨 Required CSS Classes**: - .asset-panel-container - Main panel wrapper (replaces p-4) - .panel-loading - Loading state container - .panel-form - Form wrapper with consistent styling - .panel-actions - Button container with proper spacing - .panel-card-list - Scrollable list container - .panel-card - Individual record cards - .panel-empty-state - No data state display - .panel-badge - Consistent badge styling - .panel-text-muted - Muted text color

### 2.1. Shared CSS Architecture for Slide-Out Panels

**🎨 CRITICAL: Use Unified CSS System**

All slide-out panels **MUST** use the shared CSS file to ensure consistency and maintainability.

**File Location**: client/src/pages/assets/components/AssetSlideOutPanels.css

**Import Pattern**:

import "./AssetSlideOutPanels.css"; // Always use this exact import

**❌ DO NOT**: - Create individual CSS files for panels (e.g., MyPanel.css) - Use inline styles for panel structure - Override shared classes with custom CSS - Use p-4 or other padding classes on panel containers

**✅ DO**: - Use shared CSS classes for all panel elements - Follow the established class naming conventions - Extend shared classes only when absolutely necessary - Maintain consistency across all panels

**Available CSS Class Categories**:

1. **Panel Structure**: .asset-panel-container, .panel-card-list, .panel-card
2. **Form Elements**: .panel-form, .panel-actions, .panel-data-row
3. **Visual Elements**: .panel-badge, .panel-alert, .panel-loading
4. **Utility Classes**: .panel-text-muted, .panel-gap-1, .panel-fw-bold

**Example Panel Structure**:

<SlideOutPanel>  
 <div className="asset-panel-container">  
 <div className="panel-card-list">  
 <Card className="panel-card">  
 <CardHeader className="panel-card-header">  
 <h6>Record Title</h6>  
 </CardHeader>  
 <CardBody className="panel-card-body">  
 <div className="panel-data-section">  
 <div className="panel-data-row">  
 <span className="label">Field:</span>  
 <span className="value">Value</span>  
 </div>  
 </div>  
 </CardBody>  
 </Card>  
 </div>  
 </div>  
</SlideOutPanel>

### 3. Form Controls Pattern

**Standard Form Structure**:

import { Form, FormGroup, Label, Input, Row, Col, Button } from 'reactstrap';  
  
<Form>  
 <Row>  
 <Col md={6}>  
 <FormGroup>  
 <Label  
 for="fieldName"  
 style={{ fontSize: '0.875rem', fontWeight: '500' }}  
 >  
 Field Label \* {/\* Bold label \*/}  
 </Label>  
 <Input  
 type="text"  
 name="fieldName"  
 id="fieldName"  
 value={formData.fieldName || ''}  
 onChange={handleInputChange}  
 placeholder="Enter field value..."  
 invalid={!!errors.fieldName}  
 style={{ fontSize: '0.875rem' }}  
 />  
 {errors.fieldName && (  
 <div className="invalid-feedback" style={{ fontSize: '0.75rem' }}>  
 {errors.fieldName}  
 </div>  
 )}  
 </FormGroup>  
 </Col>  
 </Row>  
  
 {/\* Form action buttons - always use Bootstrap Button \*/}  
 <div className="d-flex gap-2 mt-3">  
 <Button color="primary" onClick={handleSave}>  
 <Icon name="check" className="me-1"></Icon>  
 Save Changes  
 </Button>  
 <Button color="secondary" onClick={handleCancel}>  
 <Icon name="cross" className="me-1"></Icon>  
 Cancel  
 </Button>  
 </div>  
</Form>

**Select Dropdown Pattern**:

<FormGroup>  
 <Label  
 for="selectField"  
 style={{ fontSize: '0.875rem', fontWeight: '500' }}  
 >  
 Select Field {/\* Bold label \*/}  
 </Label>  
 <Input  
 type="select"  
 name="selectField"  
 id="selectField"  
 value={formData.selectField || ''}  
 onChange={handleInputChange}  
 style={{ fontSize: '0.875rem' }}  
 >  
 <option value="">Select option...</option>  
 {options.map(option => (  
 <option key={option.value} value={option.value}>  
 {option.label}  
 </option>  
 ))}  
 </Input>  
</FormGroup>

### 4. Button Patterns

**IMPORTANT**: All buttons must use Bootstrap Button components from reactstrap, never native HTML buttons.

**Primary Actions**:

import { Button } from 'reactstrap';  
  
<Button color="primary" size="sm" onClick={handleAction}>  
 <Icon name="plus" className="me-1"></Icon>  
 Add New  
</Button>

**Secondary Actions**:

<Button color="secondary" size="sm" onClick={handleCancel}>  
 <Icon name="cross" className="me-1"></Icon>  
 Cancel  
</Button>

**Danger Actions**:

<Button color="danger" size="sm" onClick={handleDelete}>  
 <Icon name="trash" className="me-1"></Icon>  
 Delete  
</Button>

**Button Sizes**: - size="sm" - For panel actions, table actions - size="md" (default) - For main page actions - size="lg" - For primary call-to-action buttons

**Button Colors**: - color="primary" - Main actions (Add, Save, Submit) - color="secondary" - Cancel, Close actions - color="success" - Confirm, Approve actions - color="warning" - Edit, Modify actions - color="danger" - Delete, Remove actions - color="info" - View, Details actions

### 5. Badge and Status Patterns

**Status Badges**:

const getStatusBadgeColor = (status) => {  
 switch (status?.toLowerCase()) {  
 case 'active': return 'success';  
 case 'inactive': return 'secondary';  
 case 'pending': return 'warning';  
 case 'error': return 'danger';  
 default: return 'light';  
 }  
};  
  
<span className={`badge badge-dim bg-${getStatusBadgeColor(status)}`}>  
 {status}  
</span>

**Tag Badges**:

import { Badge, Button } from 'reactstrap';  
  
<Badge  
 color="primary"  
 className="d-flex align-items-center gap-1"  
 style={{ fontSize: '0.8rem' }}  
>  
 Tag Value  
 <Button  
 color="link"  
 size="sm"  
 className="p-0 ms-1"  
 style={{ fontSize: '0.7rem', color: 'inherit' }}  
 onClick={handleRemove}  
 >  
 <Icon name="cross"></Icon>  
 </Button>  
</Badge>

### 6. Typography and Font Patterns

**IMPORTANT**: Maintain consistent font sizes and weights throughout the application.

**Font Size Standards**:

// Main content text  
style={{ fontSize: '0.875rem' }} // 14px - Standard body text  
  
// Secondary/helper text  
style={{ fontSize: '0.75rem' }} // 12px - Captions, helper text  
  
// Small text (badges, tags)  
style={{ fontSize: '0.7rem' }} // 11.2px - Small badges, compact text  
  
// Large text (headings in panels)  
style={{ fontSize: '1rem' }} // 16px - Panel headings, important text

**Font Weight Standards**:

// Labels (always bold)  
style={{ fontWeight: '500' }} // Medium weight for labels  
style={{ fontWeight: '600' }} // Semi-bold for important labels  
  
// Data/Content (never bold)  
style={{ fontWeight: '400' }} // Normal weight for data values  
style={{ fontWeight: 'normal' }} // Alternative normal weight

**Label and Data Display Pattern**:

// In tables  
<div className="user-info">  
 <span className="tb-lead" style={{ fontSize: '0.875rem', fontWeight: '500' }}>  
 {labelText} {/\* Bold label \*/}  
 </span>  
 <div className="text-soft" style={{ fontSize: '0.75rem', fontWeight: 'normal' }}>  
 {dataValue} {/\* Unbolded data \*/}  
 </div>  
</div>  
  
// In forms and panels  
<div className="mb-2">  
 <span style={{ fontSize: '0.875rem', fontWeight: '500' }}>  
 Label: {/\* Bold label \*/}  
 </span>  
 <span style={{ fontSize: '0.875rem', fontWeight: 'normal' }} className="ms-2">  
 {dataValue} {/\* Unbolded data \*/}  
 </span>  
</div>  
  
// In detail views  
<Row className="mb-3">  
 <Col md={4}>  
 <span style={{ fontSize: '0.875rem', fontWeight: '500' }}>  
 Field Label: {/\* Bold label \*/}  
 </span>  
 </Col>  
 <Col md={8}>  
 <span style={{ fontSize: '0.875rem', fontWeight: 'normal' }}>  
 {fieldValue} {/\* Unbolded data \*/}  
 </span>  
 </Col>  
</Row>

**Text Color Standards**:

// Primary text (labels, important content)  
className="text-dark"  
  
// Secondary text (data values, descriptions)  
className="text-soft"  
  
// Muted text (helper text, captions)  
className="text-muted"

## Database Patterns

### 1. Schema Definition Pattern

**Location**: api/src/db/schema/

**Structure**:

const { pgTable, uuid, varchar, text, timestamp, boolean, integer, decimal, pgEnum } = require('drizzle-orm/pg-core');  
  
// Enums  
const statusEnum = pgEnum('status', ['active', 'inactive', 'pending']);  
  
// Table definition  
const tableName = pgTable('table\_name', {  
 id: uuid('id').primaryKey().defaultRandom(),  
 name: varchar('name', { length: 255 }).notNull(),  
 description: text('description'),  
 status: statusEnum('status').default('active'),  
 isActive: boolean('is\_active').default(true),  
 createdAt: timestamp('created\_at').defaultNow(),  
 updatedAt: timestamp('updated\_at').defaultNow(),  
});  
  
module.exports = {  
 tableName,  
 statusEnum,  
};

### 2. Migration Pattern

**Location**: api/scripts/

**Structure**:

const { db } = require('../src/db');  
const { sql } = require('drizzle-orm');  
  
async function createTable() {  
 try {  
 await db.execute(sql`  
 CREATE TABLE IF NOT EXISTS table\_name (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 name VARCHAR(255) NOT NULL,  
 created\_at TIMESTAMP DEFAULT NOW(),  
 updated\_at TIMESTAMP DEFAULT NOW()  
 );  
 `);  
 console.log('✅ Table created successfully');  
 } catch (error) {  
 console.error('❌ Error creating table:', error);  
 throw error;  
 }  
}  
  
createTable().catch(console.error);

## Security Patterns

### 1. Authentication Pattern

**Middleware**: authenticateToken

// Always use in routes  
router.use(authenticateToken);

### 2. Authorization Pattern

**RBAC Middleware**: requirePermission

// Module-based permissions  
router.use(requirePermission('module\_name:read'));  
router.post('/', requirePermission('module\_name:write'), controller.create);  
router.delete('/:id', requirePermission('module\_name:delete'), controller.delete);

### 3. Input Validation Pattern

**Express Validator**:

const validateInput = [  
 body('email').isEmail().withMessage('Valid email required'),  
 body('name').notEmpty().withMessage('Name is required'),  
 body('age').isInt({ min: 0 }).withMessage('Age must be a positive integer')  
];

## Testing Patterns

### 1. API Testing Pattern

**Structure**:

describe('API Endpoint', () => {  
 beforeEach(async () => {  
 // Setup test data  
 });  
  
 afterEach(async () => {  
 // Cleanup test data  
 });  
  
 it('should return success response', async () => {  
 const response = await request(app)  
 .get('/api/v1/endpoint')  
 .set('Authorization', `Bearer ${testToken}`)  
 .expect(200);  
  
 expect(response.body.success).toBe(true);  
 expect(response.body.data).toBeDefined();  
 });  
});

### 2. Component Testing Pattern

**Structure**:

import { render, screen, fireEvent } from '@testing-library/react';  
import Component from './Component';  
  
describe('Component', () => {  
 it('should render correctly', () => {  
 render(<Component />);  
 expect(screen.getByText('Expected Text')).toBeInTheDocument();  
 });  
  
 it('should handle user interaction', () => {  
 const mockHandler = jest.fn();  
 render(<Component onAction={mockHandler} />);  
  
 fireEvent.click(screen.getByRole('button'));  
 expect(mockHandler).toHaveBeenCalled();  
 });  
});

## Common Pitfalls to Avoid

### ❌ Don’t Do This:

* Use axios instead of fetch
* Import from wrong paths (../db/connection vs ../db)
* Use token instead of accessToken in localStorage
* Mix different UI component libraries
* Skip authentication/authorization middleware
* Use inconsistent error handling
* Forget to validate inputs
* Use different badge/button styling
* **Use native HTML buttons instead of Bootstrap Button components**
* **Mix font sizes randomly - stick to the standard sizes**
* **Make data values bold - only labels should be bold**
* **Use inconsistent font weights**

### ✅ Always Do This:

* Use native fetch API with consistent patterns
* Follow established import paths
* Use accessToken from localStorage
* Follow UI component patterns consistently
* Apply authentication and RBAC to all routes
* Use structured error handling with try/catch
* Validate all inputs with express-validator
* Follow established styling patterns
* **Always use Bootstrap Button components from reactstrap**
* **Use consistent font sizes: 0.875rem (body), 0.75rem (secondary), 0.7rem (small)**
* **Bold labels (fontWeight: ‘500’), unbold data (fontWeight: ‘normal’)**
* **Import Button from reactstrap, never use HTML <button> tags**

## Quick Reference Checklist

### Backend Checklist:

* Service uses const { db } = require('../db');
* Controller has proper error handling
* Routes use authenticateToken and requirePermission
* Input validation with express-validator
* Structured JSON responses

### Frontend Checklist:

* API utility uses fetch (not axios)
* Uses accessToken from localStorage
* Consistent error handling with toast
* Follows UI component patterns
* Proper loading states
* Consistent styling classes

### UI Checklist:

* SlideOutPanel from correct import path
* **REQUIRED: Import shared CSS** import "./AssetSlideOutPanels.css"
* **NEVER create individual panel CSS files**
* Use .asset-panel-container instead of p-4
* Use shared CSS classes (.panel-form, .panel-actions, etc.)
* **All buttons use Bootstrap Button from reactstrap (never HTML button)**
* Consistent button styling with icons and proper colors
* Badge styling with .panel-badge class
* Form validation and error display
* Responsive layout with Row/Col
* **Font sizes follow standards: 0.875rem (body), 0.75rem (secondary), 0.7rem (small)**
* **Labels are bold (fontWeight: ‘500’), data is unbolded (fontWeight: ‘normal’)**
* **All form Labels have consistent styling**
* **Button imports from reactstrap, not native HTML**

### DataTable Checklist:

* **Export buttons present: Excel (green), PDF (red), CSV (blue)**
* **Show entries dropdown with options: 10, 25, 50, 100**
* **All columns have sortable headers with up/down arrows**
* **Bulk actions dropdown on left side**
* **Search toggle button present**
* **Row selection checkboxes implemented**
* **Actions dropdown with three dots menu**
* **DataTable.css file imported and styled**
* **Consistent column header styling with proper font weights**
* **Export functionality implemented for ALL formats (CSV, Excel, PDF)**
* **Export buttons have correct colors (Excel=green, PDF=red, CSV=blue)**
* **Sorting functionality works on all sortable columns**
* **Pagination component included**
* **Responsive design for mobile devices**

### 1.1 DataTable Export Pattern

**CRITICAL: All datatables MUST implement export functionality for all three formats**

The export functionality is a key feature of all datatables in the application. Users expect to be able to export data in various formats for reporting, analysis, and sharing. All datatables must implement export functionality for CSV, Excel, and PDF formats.

**Required Export Formats:** 1. **CSV** - Simple comma-separated values format for universal compatibility 2. **Excel** - XLS format for spreadsheet applications 3. **PDF** - Formatted document for printing and sharing

**Export Button Implementation:**

// Export Section - REQUIRED  
<li className="export-section">  
 <span className="export-label">Export</span>  
 <Button  
 color="light"  
 size="sm"  
 className="btn-icon export-excel"  
 onClick={() => handleExport('excel')}  
 title="Export to Excel"  
 >  
 <Icon name="file-xls" className="export-excel"></Icon>  
 </Button>  
 <Button  
 color="light"  
 size="sm"  
 className="btn-icon export-pdf"  
 onClick={() => handleExport('pdf')}  
 title="Export to PDF"  
 >  
 <Icon name="file-pdf" className="export-pdf"></Icon>  
 </Button>  
 <Button  
 color="light"  
 size="sm"  
 className="btn-icon export-csv"  
 onClick={() => handleExport('csv')}  
 title="Export to CSV"  
 >  
 <Icon name="file-text" className="export-csv"></Icon>  
 </Button>  
</li>

**Export Button Colors - REQUIRED:** - Excel: Green (export-excel class, file-xls icon) - PDF: Red (export-pdf class, file-pdf icon) - CSV: Blue (export-csv class, file-text icon)

**Export Data Preparation:** - Always map raw data to human-readable column names - Format dates, numbers, and special values for readability - Include all relevant fields that would be useful in exports - Handle null/undefined values gracefully

**Export Filename Convention:** - Use descriptive names: [entity-type]\_export\_[date].extension - Include current date in ISO format (YYYY-MM-DD) - Examples: assets\_export\_2025-07-25.csv, users\_export\_2025-07-25.xlsx

**CSV Export Implementation:**

// CSV Export - REQUIRED  
const csvContent = [  
 Object.keys(exportData[0]).join(','),  
 ...exportData.map(row => Object.values(row).join(','))  
].join('\n');  
  
const blob = new Blob([csvContent], { type: 'text/csv' });  
const url = window.URL.createObjectURL(blob);  
const a = document.createElement('a');  
a.href = url;  
a.download = `${fileName}.csv`;  
a.click();  
window.URL.revokeObjectURL(url);

**Excel Export Implementation:**

// Excel Export - REQUIRED  
// Use export-from-json library  
import('export-from-json').then(module => {  
 const exportFromJSON = module.default;  
 exportFromJSON({  
 data: exportData,  
 fileName,  
 exportType: 'xls'  
 });  
}).catch(error => {  
 console.error('Error exporting to Excel:', error);  
});

**PDF Export Implementation:**

// PDF Export - REQUIRED  
// Create a printable HTML table for PDF export  
const printWindow = window.open('', '\_blank');  
  
if (!printWindow) {  
 alert('Please allow pop-ups to export as PDF');  
 return;  
}  
  
// Create a styled HTML table with standard styling  
const tableHTML = `  
 <!DOCTYPE html>  
 <html>  
 <head>  
 <title>${fileName}</title>  
 <style>  
 body { font-family: Arial, sans-serif; }  
 table { width: 100%; border-collapse: collapse; margin-bottom: 20px; }  
 th, td { padding: 8px; text-align: left; border-bottom: 1px solid #ddd; }  
 th { background-color: #f2f2f2; font-weight: bold; }  
 .header { margin-bottom: 20px; }  
 .header h1 { margin-bottom: 5px; }  
 .header p { color: #666; margin-top: 0; }  
 @media print {  
 button { display: none; }  
 }  
 </style>  
 </head>  
 <body>  
 <div class="header">  
 <h1>Data Export</h1>  
 <p>Generated on ${new Date().toLocaleString()}</p>  
 </div>  
 <button onclick="window.print();window.close();" style="padding: 10px; margin-bottom: 20px; cursor: pointer;">  
 Print as PDF  
 </button>  
 <table>  
 <thead>  
 <tr>  
 ${Object.keys(exportData[0]).map(key => `<th>${key}</th>`).join('')}  
 </tr>  
 </thead>  
 <tbody>  
 ${exportData.map(row => `  
 <tr>  
 ${Object.values(row).map(value => `<td>${value}</td>`).join('')}  
 </tr>  
 `).join('')}  
 </tbody>  
 </table>  
 <script>  
 // Auto-trigger print dialog after a short delay  
 setTimeout(() => {  
 window.print();  
 }, 500);  
 </script>  
 </body>  
 </html>  
`;  
  
printWindow.document.write(tableHTML);  
printWindow.document.close();

**Best Practices:** - Always include all three export formats - Use the standard button colors and icons - Format data appropriately for each export format - Include headers and metadata in exports - Handle large datasets efficiently - Provide user feedback during export operations

### 1.2 DataTable Export Utility

**IMPORTANT: Always use the shared export utility for all datatables**

To maintain consistency and avoid code duplication, all datatables must use the shared export utility located at client/src/utils/exportUtils.js. This utility provides standardized export functionality for CSV, Excel, and PDF formats.

**Import Pattern**:

import { handleExport as exportData } from "@/utils/exportUtils";

**Usage in DataTable Components**:

// Handle export functionality using the global export utility  
const handleExport = (format) => {  
 // Prepare data for export with custom formatting  
 const formattedData = tableData.map(row => ({  
 'Column 1': row.field1,  
 'Column 2': row.field2,  
 // Add all fields that should be included in exports  
 }));  
  
 // Use the global export utility  
 exportData(format, formattedData, 'entityName', null, 'Export Title');  
};

**Key Benefits**: - Consistent export functionality across all datatables - Centralized maintenance and updates - Standardized file naming and formatting - Reduced code duplication - Easier to add new export formats in the future

**Implementation in Export Buttons**:

<Button  
 color="light"  
 size="sm"  
 className="btn-icon export-excel"  
 onClick={() => handleExport('excel')}  
 title="Export to Excel"  
>  
 <Icon name="file-xls" className="export-excel"></Icon>  
</Button>

**Available Export Formats**: - CSV: handleExport('csv') - Excel: handleExport('excel') - PDF: handleExport('pdf')

**Utility Functions**: - exportToCsv(data, fileName) - Export data to CSV format - exportToExcel(data, fileName) - Export data to Excel format - exportToPdf(data, fileName, title) - Export data to PDF format - prepareDataForExport(rawData, fieldMappings) - Prepare data for export - handleExport(format, data, entityName, fieldMappings, title) - Main export function

## 🔍 Vulnerability Analytics Patterns

### Database Schema Structure

**Core Analytics Tables:** - vulnerability\_cost\_analysis - AI-powered cost impact analysis - vulnerability\_cost\_factors - Configurable cost calculation factors - vulnerability\_cost\_history - Historical cost tracking and trends - vulnerability\_cost\_models - AI models for cost prediction - vulnerability\_patches - Patch management and tracking - vulnerability\_poams - POAM relationships

**Integration with Existing Data:** - Links to vulnerabilities table (56K records from Tenable) - Connects to assets table for asset-specific cost analysis - Integrates with poams table for remediation tracking

### API Endpoint Patterns

**Base URL:** /api/v1/vulnerability-analytics/

**Standard CRUD Endpoints:**

GET /cost-analysis # List with filtering  
POST /cost-analysis # Create new analysis  
PUT /cost-analysis/:id # Update existing  
DELETE /cost-analysis/:id # Delete analysis  
  
GET /cost-factors # List factors  
POST /cost-factors # Create factor  
PUT /cost-factors/:id # Update factor  
DELETE /cost-factors/:id # Delete factor  
  
GET /cost-models # List models  
POST /cost-models # Create model  
PUT /cost-models/:id # Update model  
DELETE /cost-models/:id # Delete model

**Analytics Endpoints:**

GET /cost-summary # Aggregate cost metrics  
GET /vulnerability/:id/cost-analysis # Vulnerability-specific analysis

### Service Layer Patterns

**VulnerabilityAnalyticsService Methods:** - getCostAnalysis(filters) - Retrieve with filtering/sorting - createCostAnalysis(data) - Create with validation - updateCostAnalysis(id, data) - Update existing record - getCostSummary() - Aggregate analytics - getVulnerabilityWithCostAnalysis(id) - Join queries

### Permission Requirements

**Required Permissions:** - vulnerability\_management:read - View analytics data - vulnerability\_management:write - Create/update analyses - vulnerability\_management:delete - Delete records

### Frontend Integration Patterns

**API Client Structure:**

// utils/vulnerabilityAnalyticsApi.js  
const getCostAnalysis = (filters) => apiRequest('/vulnerability-analytics/cost-analysis', { params: filters });  
const createCostAnalysis = (data) => apiRequest('/vulnerability-analytics/cost-analysis', { method: 'POST', data });

**Component Patterns:** - Cost analysis dashboard components - Vulnerability cost breakdown panels - AI-powered cost prediction interfaces - Historical cost trend visualizations

This guide should be referenced for all new development to ensure consistency across the entire application stack.

## 🎨 Slide-Out Panel CSS Class Reference

### Shared CSS File: AssetSlideOutPanels.css

**CRITICAL**: All slide-out panels must use this shared CSS file. Never create individual CSS files for panels.

**Panel Structure Classes**: - .asset-panel-container - Main panel wrapper (replaces p-4) - .panel-card-list - Scrollable content container - .panel-card - Individual record cards with hover effects - .panel-card-header - Card header styling - .panel-card-body - Card body content area

**Form Classes**: - .panel-form - Form wrapper with consistent styling - .panel-actions - Button container with proper spacing - .panel-data-row - Key-value pair display - .panel-data-section - Grouped data display

**State Classes**: - .panel-loading - Loading state container - .panel-empty-state - No data state display - .panel-alert - Alert message styling

**Visual Classes**: - .panel-badge - Consistent badge styling - .panel-text-muted - Muted text color - .panel-fw-bold - Bold font weight

**Utility Classes**: - .panel-gap-1, .panel-gap-2 - Spacing utilities - .panel-border-light - Light border color

**Usage Pattern**:

import "./AssetSlideOutPanels.css"; // REQUIRED  
  
<SlideOutPanel>  
 <div className="asset-panel-container">  
 <div className="panel-card-list">  
 {/\* Content \*/}  
 </div>  
 </div>  
</SlideOutPanel>