Frontend Developer Assessment: Dynamic Filter Component System

Overview

Build a reusable, type-safe dynamic filter component system that can be integrated with any data table. This assessment tests your ability to architect complex React applications with TypeScript, focusing on component modularity, state management, and creating intuitive user interfaces.

Project Requirements

🎯 Core Functionality

1. Dynamic Filter Builder

Create a filter interface that allows users to:

1. **Add multiple filter conditions** with different field types
2. **Select fields/columns** to filter on
3. **Choose appropriate operators** based on the selected field type
4. **Input filter values** using context-appropriate input methods
5. **Remove individual filters** or **clear all filters**
6. **Apply filters** to update displayed data

2. Multi-Type Filter Support

Your system must handle these data types with appropriate operators and inputs:

Text Fields:

1. Operators: Equals, Contains, Starts With, Ends With, Does Not Contain
2. Input: Text input field

Number Fields:

1. Operators: Equals, Greater Than, Less Than, Greater Than or Equal, Less Than or Equal
2. Input: Number input field with validation

Date Fields:

1. Operators: Between (date range)
2. Input: Date range picker with calendar interface

Amount/Currency Fields:

1. Operators: Between (amount range)
2. Input: Number inputs for min/max amounts with proper formatting

Single Select Fields:

1. Operators: Is, Is Not
2. Input: Dropdown with predefined options

Multi-Select Fields:

1. Operators: In, Not In
2. Input: Multi-select dropdown with checkboxes

Boolean Fields:

1. Operators: Is
2. Input: Toggle switch or checkbox

3. Component Architecture

Reusable Component System: Design a modular architecture that supports:

1. **Dynamic input rendering** based on field type
2. **Type-safe prop interfaces** with TypeScript
3. **Configurable field definitions** and operator mappings
4. **Validation logic** for filter conditions
5. **State management** for filter operations

Component Organization: Your architecture should demonstrate:

1. **Separation of concerns** between UI, logic, and data
2. **Reusable components** that can work independently
3. **Clear naming conventions** and logical organization
4. **Type safety** throughout the application
5. **Extensible design** for adding new field types

4. Data Structure & Client-Side Filtering

Sample Data Structure: You will work with a complex local JSON dataset. Here's a sample structure you should create and use:

```
[
{
"id": 1,
"name": "John Smith",
"email": "john.smith@company.com",
"department": "Engineering",
"role": "Senior Developer",
"salary": 95000,
"joinDate": "2021-03-15",
"isActive": true,
"skills": ["React", "TypeScript", "Node.js", "GraphQL"],
"address": {
"city": "San Francisco",
"state": "CA",
"country": "USA"
},
"projects": 3,
"lastReview": "2024-01-15",
"performanceRating": 4.5
},
// ... more employee records
]
```

Data Requirements:

1. Create at least 50 sample records with varied data
2. Include multiple data types: text, numbers, dates, booleans, arrays, nested objects
3. Ensure data diversity for meaningful filtering demonstrations
4. Add realistic variations in values to test different filter scenarios
5. Use **mock-json-api** package mock the api from json data

Client-Side Filtering Implementation:

1. **Implement filtering algorithms** that process the local JSON data
2. **Real-time table updates** as filters are applied/modified
3. **Multiple filter logic**: AND between different fields, OR within same field
4. **Case-insensitive text matching** for string fields
5. **Date range comparisons** with proper date parsing
6. **Numeric range filtering** with min/max values
7. **Array filtering** for multi-select fields (IN/NOT IN operations)
8. **Nested object filtering** (e.g., filter by address.city)

Table Component Requirements:

1. Display the local JSON data in a sortable table
2. Show "No results" message when filters return empty dataset

3. **Display record counts**: Total records and filtered records count
4. **Performance optimization**: Efficient filtering for 50+ records
5. **Data transformation**: Handle nested object display in table cells
6. **Real-time updates**: Table should update immediately as filters change

Technical Requirements

🛠️ Technology Stack

1. **React 18** with TypeScript
2. **Vite** for project setup and development
3. **Material UI** preferred for styling; other CSS solutions are acceptable. Tailwind is not expected.
4. **Lucide React** for icons (trash, calendar, etc.)

Assessment Criteria

🏆 Scoring Overview (100 points)

1. **Architecture & Code Quality (40)**: Separation of concerns, modular components, clear organization, strong TypeScript usage.
2. **Filtering & Data Handling (40)**: Required operators working across types, correct client-side algorithms, real-time table updates, dataset handling including arrays and nested objects.
3. **Technical Excellence (20)**: Performance (efficient filtering, minimal re-renders) and robust error handling/validation.

🌟 Bonus (Up to 10 points)

1. Filter persistence, export to CSV/JSON, advanced operators (regex/custom ranges), helpful UX touches (e.g., debounced updates, accessibility).

Deliverables

📦 Required Submissions

1. **Source Code**: Complete React application with all components
2. **Live Demo**: Deployed version (Vercel, Netlify, or similar)
3. **Documentation**: README with setup instructions and component usage examples
4. **Code Comments**: Inline documentation for complex logic

📋 Submission Checklist

1. Required filter types implemented and operators working
2. Dynamic operator selection based on field type
3. Local JSON dataset with at least 50 records
4. Client-side filtering producing real-time table updates
5. Validation and error handling in place
6. TypeScript types defined for data and core components
7. Working demo link (table filtering shown)

💡 Implementation Approach

Focus on these key architectural decisions:

1. **Data Modeling**: Define your TypeScript interfaces and types first
2. **State Management**: Plan how filter state will be managed and updated
3. **Component Hierarchy**: Design the relationship between parent and child components
4. **Input Strategy**: Determine how to render appropriate inputs for different data types
5. **Validation Logic**: Implement proper validation for filter conditions
6. **Filtering Algorithms**: Design efficient client-side filtering logic for different data types
7. **Data Transformation**: Plan how to handle nested objects and array filtering

🔧 Data Filtering Requirements

Implement filtering algorithms for:

1. **Text fields**: Case-insensitive contains, starts with, ends with, exact match
2. **Number fields**: Equal, not equal, greater than, less than, between ranges
3. **Date fields**: Before, after, between ranges, relative dates (last 30 days)
4. **Boolean fields**: True/false matching
5. **Array fields**: Contains any, contains all, does not contain
6. **Nested objects**: Dot notation filtering (e.g., address.city)

Performance Considerations:

1. Optimize filtering for datasets with 50+ records
2. Implement efficient data structures for quick lookups
3. Consider memoization for expensive filter operations
4. Handle edge cases (null values, empty arrays, invalid dates)

Note: The specific implementation details, component names, and architectural patterns are intentionally left for you to design. This assessment evaluates your ability to architect a complex system from functional requirements.

Evaluation Process

🔍 Code Review

Your submission will be evaluated based on:

1. **Functionality**: Does it meet all requirements?
2. **Code Quality**: Is the code clean, maintainable, and well-structured?
3. **User Experience**: Is the interface intuitive and professional?
4. **Technical Skills**: Does it demonstrate strong React and TypeScript knowledge?
5. **Problem Solving**: How are edge cases and complex scenarios handled?

📞 Questions & Support

If you have questions during implementation:

1. Review the requirements carefully
2. Focus on core functionality first, then add enhancements
3. Document any assumptions or design decisions
4. Test thoroughly across different scenarios

⏰ **Time Allocation**: You have **3 days** to complete this assessment. Focus on delivering a working solution that demonstrates your skills rather than perfecting every detail. Quality and thoughtful implementation are valued over quantity of features.

🎯 **Success Criteria**: A successful submission will be a functional, well-architected dynamic filter system that could be used in a real-world application, demonstrating your ability to build complex, reusable React components with TypeScript.