Integration Guide: GEE Explorer with React and MapLibre

This document outlines an approach to integrate the Python-based GEE Dataset Explorer with a React application using MapLibre.

Architecture Overview

The integration will follow a client-server architecture:

1. **Backend**: Python Flask API (existing GEE Explorer)
2. **Frontend**: React with MapLibre (your existing application)

Integration Strategy

1. API-First Approach

Convert the current Flask app into a proper REST API:

```
React/MapLibre App <-----> Flask API <-----> Google Earth Engine
```

2. Key Components to Integrate

Backend (Python Flask)

1. **Create RESTful Endpoints**:
   - `/api/search_datasets` - Search for datasets (already implemented)
   - `/api/get_tile` - Get tile URLs for map display (already implemented)
   - `/api/get_value_at_location` - Get pixel values (already implemented)
   - Add CORS support for cross-domain requests
2. **Decouple UI from Server Logic**:
   - Remove HTML templates and static file serving
   - Focus only on data processing and JSON responses

Frontend (React)

1. **API Integration**:
   - Create services to call the Flask API endpoints
   - Use React hooks or state management (Redux/Context API) to store dataset state
2. **MapLibre Integration**:

- Replace Leaflet with MapLibre (or keep both if needed for different visualizations)
- Implement custom layer handling for GEE tiles in MapLibre
- Port the visualization controls (opacity, basemap selection) to React

3. **UI Components to Port**:
   - Dataset search and results panel
   - Left panel for dataset information
   - Band and visualization controls
   - Temporal selection controls for time-series data

Implementation Steps

Step 1: Convert Flask App to API Server

1. Modify your Flask routes to return only JSON data
2. Add CORS headers:

```python
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
```

3. Update `app.py` to not serve HTML content

Step 2: Create React Components for GEE Functionality

1. **Dataset Search Component**:

jsx

```jsx
function DatasetSearch() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);

  const searchDatasets = async () => {
    const response = await fetch('http://your-flask-api/api/search_datasets', {
      method: 'POST',
      body: JSON.stringify({ query }),
      headers: { 'Content-Type': 'application/json' }
    });
    const data = await response.json();
    setResults(data.results);
  };

  return (
    <div>
      <input value={query} onChange={e => setQuery(e.target.value)} />
      <button onClick={searchDatasets}>Search</button>
      <ResultsList results={results} />
    </div>
  );
}
```

2. **MapLibre GEE Layer Component**:

jsx

```jsx
function GeeMapLayer({ tileUrl, opacity }) {
  const map = useMap();

  useEffect(() => {
    if (!map || !tileUrl) return;

    // Add GEE tile layer to MapLibre
    map.addSource('gee-source', {
      type: 'raster',
      tiles: [tileUrl],
      tileSize: 256
    });

    map.addLayer({
      id: 'gee-layer',
      type: 'raster',
      source: 'gee-source',
      paint: {
        'raster-opacity': opacity
      }
    });

    return () => {
      // Cleanup on unmount
      if (map.getSource('gee-source')) {
        map.removeLayer('gee-layer');
        map.removeSource('gee-source');
      }
    };
  }, [map, tileUrl, opacity]);

  return null; // This is a utility component with no visual rendering
}
```

3. **Dataset Info Panel Component**:

jsx

```jsx
function DatasetInfoPanel({ dataset }) {
  // Port existing left panel functionality from dataset.js
  return (
    <div className="dataset-panel">
      <h3>{dataset.title}</h3>
      <div className="dataset-description">
        <h4>Description</h4>
        <p>{dataset.description}</p>
      </div>
      {/* Add band selection, temporal controls, etc. */}
    </div>
  );
}
```

Step 3: Integrate MapLibre with GEE Visualization

1. Update MapLibre to handle GEE tile URLs
2. Implement click handlers for value retrieval:

jsx

```
function MapContainer() {
  const [mapInstance, setMapInstance] = useState(null);
  const [geeLayer, setGeeLayer] = useState(null);

  useEffect(() => {
    const map = new maplibregl.Map({
      container: 'map',
      style: 'mapbox://styles/mapbox/dark-v10',
      center: [0, 0],
      zoom: 2
    });

    map.on('load', () => {
      setMapInstance(map);
    });

    // Handle right-click for value retrieval
    map.on('contextmenu', async (e) => {
      // Get coordinates
      const { lng, lat } = e.lngLat;

      // Call API to get value at location
      const response = await fetch('http://your-flask-api/api/get_value_at_location', {
        method: 'POST',
        body: JSON.stringify({
          dataset_id: currentDataset,
          coordinates: { lon: lng, lat: lat }
        }),
        headers: { 'Content-Type': 'application/json' }
      });

      const data = await response.json();
      // Display value popup
      showValuePopup(data, lng, lat);
    });

    return () => map.remove();
  }, []);

  return (
    <div id="map" style={{ width: '100%', height: '100vh' }}>
      {/* Controls and other components */}
    </div>
```

```
    );
  }
```

Step 4: Implement State Management

Use React Context API or Redux to manage:

- Current dataset
- Search results
- Map settings
- Visualization parameters
- Temporal selection

Advanced Considerations

1. Server-Side vs. Client-Side Rendering

For complex GEE operations, keep processing on the server-side (Python). For UI operations, move to client-side (React).

2. Performance Optimizations

- Implement dataset caching both on backend and frontend
- Use React memoization for expensive computations
- Consider using WebSockets for real-time updates during long GEE operations

3. Authentication & Authorization

- Implement authentication for GEE access using API keys or OAuth
- Use JWT tokens between React and Flask

Technical Challenges and Solutions

Challenge 1: Cross-Origin Resource Sharing (CORS)

Ensure your Flask API includes proper CORS headers or use a proxy in development:

```python
# In Flask
from flask_cors import CORS
app = Flask(__name__)
CORS(app, resources={r"/api/*": {"origins": "*"}})
```

Challenge 2: MapLibre vs. Leaflet Differences

MapLibre and Leaflet handle maps differently. Key differences to address:

- Tile layer syntax and options are different
- Event handling (e.g., context menu vs. right-click)
- Custom controls implementation
- Popup handling

Challenge 3: Stateful Visualization Controls

Port the visualization logic from vanilla JS to React state:

jsx

```jsx
function VisualizationControls({ dataset, onApply }) {
  const [band, setBand] = useState('');
  const [colormap, setColormap] = useState('terrain');
  const [min, setMin] = useState(0);
  const [max, setMax] = useState(255);

  const applyVisualization = () => {
    onApply({ band, colormap, min, max });
  };

  return (
    <div className="visualization-controls">
      {/* Band selection */}
      <select value={band} onChange={e => setBand(e.target.value)}>
        {dataset.bands.map(b => (
          <option key={b.name} value={b.name}>{b.name}</option>
        ))}
      </select>

      {/* Other controls */}
      <button onClick={applyVisualization}>Apply</button>
    </div>
  );
}
```

Conclusion

This integration approach keeps the powerful backend processing of the GEE Explorer while modernizing the frontend with React and MapLibre. The API-first approach allows for clean separation of concerns and flexible UI implementation.

For successful integration:

1. Separate backend logic from UI rendering
2. Create a clean API layer
3. Port UI components to React systematically
4. Adapt map handling from Leaflet to MapLibre
5. Use proper state management for application data

With this approach, you'll be able to leverage the existing GEE Explorer functionality while gaining the benefits of a modern React frontend with MapLibre's powerful mapping capabilities.