Complete Guide to REST API, Axios, Event Handling & Conditional Rendering in React

Table of Contents

REST API Fundamentals

What is REST?

REST stands for **Representational State Transfer**. Think of it as a set of rules for how different software applications should communicate with each other over the internet.

Real-world analogy: Imagine REST API as a waiter in a restaurant:

- You (the client) place an order (request)
- The waiter takes your order to the kitchen (server)
- The kitchen prepares your food (processes the request)
- The waiter brings back your food (response)

Key Concepts of REST:

1. **Resources**

- Everything in REST is treated as a "resource" (users, products, posts, etc.)
- Each resource has a unique URL (like a home address)
- Example: `https://api.example.com/users/123` represents user with ID 123

2. **HTTP Methods (Verbs)**

REST uses standard HTTP methods to perform different actions:

Method	Purpose	Example
GET	Retrieve data	Get list of users
POST	Create new data	Create a new user
PUT	Update existing data	Update user information
DELETE	Remove data	Delete a user

3. Status Codes

APIs return status codes to tell you what happened:

- **200**: Success
- **201**: Created successfully
- **400**: Bad request (you made an error)
- **404**: Not found
- **500**: Server error

REST API Structure Example:

```
Base URL: https://api.mystore.com

GET /products           → Get all products
GET /products/5         → Get product with ID 5
POST /products          → Create a new product
PUT /products/5         → Update product with ID 5
DELETE /products/5      → Delete product with ID 5
```

Axios - HTTP Client Library

What is Axios?

Axios is a JavaScript library that makes it easy to send HTTP requests to APIs. It's like having a smart messenger that can talk to servers for you.

Why use Axios instead of built-in fetch()?

- Easier to use and understand
- Better error handling
- Automatic JSON parsing
- Request/response interceptors
- Works in both browser and Node.js

Installing Axios

```bash
npm install axios
```

Basic Axios Usage

1. Importing Axios

```javascript
import axios from 'axios';
```

2. Making GET Requests

```javascript
// Get all users
const getUsers = async () => {
  try {
    const response = await axios.get('https://api.example.com/users');
    console.log(response.data); // The actual data
  } catch (error) {
    console.error('Error fetching users:', error);
  }
};
```

3. Making POST Requests

```javascript
// Create a new user
const createUser = async (userData) => {
  try {
    const response = await axios.post('https://api.example.com/users', userData);
    console.log('User created:', response.data);
  } catch (error) {
    console.error('Error creating user:', error);
  }
};
```

4. Making PUT Requests

```javascript
// Update existing user
const updateUser = async (userId, userData) => {
  try {
    const response = await axios.put(`https://api.example.com/users/${userId}`, userData);
    console.log('User updated:', response.data);
  } catch (error) {
    console.error('Error updating user:', error);
  }
};
```

5. Making DELETE Requests

```javascript
// Delete a user
const deleteUser = async (userId) => {
  try {
    await axios.delete(`https://api.example.com/users/${userId}`);
    console.log('User deleted successfully');
  } catch (error) {
    console.error('Error deleting user:', error);
  }
};
```

Using Axios with React

Complete Example - User Management Component:

```
javascript
```

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const UserManager = () => {
  const [users, setUsers] = useState([]);
  const [newUser, setNewUser] = useState({ name: '', email: '' });
  const [loading, setLoading] = useState(false);

  // Fetch users when component loads
  useEffect(() => {
    fetchUsers();
  }, []);

  // GET - Fetch all users
  const fetchUsers = async () => {
    setLoading(true);
    try {
      const response = await axios.get('https://jsonplaceholder.typicode.com/users');
      setUsers(response.data);
    } catch (error) {
      console.error('Error fetching users:', error);
    } finally {
      setLoading(false);
    }
  };

  // POST - Create new user
  const handleCreateUser = async (e) => {
    e.preventDefault();
    try {
      const response = await axios.post('https://jsonplaceholder.typicode.com/users', newUser);
      setUsers([...users, response.data]);
      setNewUser({ name: '', email: '' }); // Clear form
    } catch (error) {
      console.error('Error creating user:', error);
    }
  };

  // DELETE - Remove user
  const handleDeleteUser = async (userId) => {
    try {
      await axios.delete(`https://jsonplaceholder.typicode.com/users/${userId}`);
      setUsers(users.filter(user => user.id !== userId));
    } catch (error) {
      console.error('Error deleting user:', error);
    }
  };

  return (
    <div>
      <h2>User Management</h2>

      {/* Create User Form */}
      <form onSubmit={handleCreateUser}>
        <input
          type="text"
          placeholder="Name"
          value={newUser.name}
          onChange={(e) => setNewUser({...newUser, name: e.target.value})}
        />
        <input
          type="email"
          placeholder="Email"
          value={newUser.email}
          onChange={(e) => setNewUser({...newUser, email: e.target.value})}
        />
        <button type="submit">Add User</button>
      </form>

      {/* Users List */}
      {loading ? (
        <p>Loading users...</p>
      ) : (
        <ul>
          {users.map(user => (
            <li key={user.id}>
              {user.name} - {user.email}
              <button onClick={() => handleDeleteUser(user.id)}>Delete</button>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
};

export default UserManager;
```

Event Handling in React

What are Events?

Events are actions that happen in your application - like clicking a button, typing in a text box, or moving the mouse. Event handling is how you respond to these actions.

Real-world analogy: Think of events like doorbells - when someone presses the doorbell (event), you respond by opening the door (event handler).

Common Events in React:

1. **onClick** - When something is clicked

```javascript
const handleClick = () => {
  alert('Button was clicked!');
};

<button onClick={handleClick}>Click Me</button>
```

2. **onChange** - When input value changes

```javascript
const [text, setText] = useState('');

const handleChange = (e) => {
  setText(e.target.value); // e.target.value is the current input value
};

<input
  type="text"
  value={text}
  onChange={handleChange}
  placeholder="Type something..."
/>
```

3. **onSubmit** - When a form is submitted

```javascript
const handleSubmit = (e) => {
  e.preventDefault(); // Prevents page refresh
  console.log('Form submitted!');
};

<form onSubmit={handleSubmit}>
  <input type="text" />
  <button type="submit">Submit</button>
</form>
```

Event Handling in Functional Components

Basic Example:

```javascript
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  // Event handler functions
  const handleIncrement = () => {
    setCount(count + 1);
  };

  const handleDecrement = () => {
    setCount(count - 1);
  };

  const handleReset = () => {
    setCount(0);
  };

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={handleIncrement}>+</button>
      <button onClick={handleDecrement}>-</button>
      <button onClick={handleReset}>Reset</button>
    </div>
  );
};
```

Handling Input Events:

```javascript
const LoginForm = () => {
  const [formData, setFormData] = useState({
    username: '',
    password: ''
  });

  const handleInputChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Login data:', formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        name="username"
        value={formData.username}
        onChange={handleInputChange}
        placeholder="Username"
      />
      <input
        type="password"
        name="password"
        value={formData.password}
        onChange={handleInputChange}
        placeholder="Password"
      />
      <button type="submit">Login</button>
    </form>
  );
};
```

Event Handling in Class Components

```javascript
import React, { Component } from 'react';

class ClassCounter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };

    // Binding methods (important for 'this' to work)
    this.handleIncrement = this.handleIncrement.bind(this);
    this.handleDecrement = this.handleDecrement.bind(this);
  }

  handleIncrement() {
    this.setState({ count: this.state.count + 1 });
  }

  handleDecrement() {
    this.setState({ count: this.state.count - 1 });
  }

  // Alternative: Arrow function (automatically binds 'this')
  handleReset = () => {
    this.setState({ count: 0 });
  };

  render() {
    return (
      <div>
        <h2>Count: {this.state.count}</h2>
        <button onClick={this.handleIncrement}>+</button>
        <button onClick={this.handleDecrement}>-</button>
        <button onClick={this.handleReset}>Reset</button>
      </div>
    );
  }
}
```

Understanding 'this' and 'bind' in Class Components

Why do we need to bind?

In JavaScript classes, `this` doesn't automatically refer to the class instance inside methods. We need to bind it.

```javascript
// Problem: 'this' is undefined
handleClick() {
  console.log(this); // undefined!
  this.setState(...); // Error!
}

// Solution 1: Bind in constructor
constructor(props) {
  super(props);
  this.handleClick = this.handleClick.bind(this);
}

// Solution 2: Use arrow functions
handleClick = () => {
  console.log(this); // Works! Points to component instance
  this.setState(...); // Works!
};

// Solution 3: Bind inline (not recommended - creates new function each render)
<button onClick={this.handleClick.bind(this)}>Click</button>
```

Conditional Rendering

What is Conditional Rendering?

Conditional rendering means showing different content based on certain conditions. It's like having different conversations depending on who you're talking to.

Real-world analogy: Like a traffic light - it shows different colors (red, yellow, green) based on traffic conditions.

Methods of Conditional Rendering:

1. If-Else Statements

```javascript
const Greeting = ({ isLoggedIn, userName }) => {
  if (isLoggedIn) {
    return <h1>Welcome back, {userName}!</h1>;
  } else {
    return <h1>Please log in to continue.</h1>;
  }
};
```

2. Ternary Operator (? :)

```javascript
const UserStatus = ({ isOnline }) => {
  return (
    <div>
      <h2>User Status</h2>
      <p>You are {isOnline ? 'online' : 'offline'}</p>
      <div className={isOnline ? 'status-online' : 'status-offline'}>
        {isOnline ? '🟢' : '🔴'}
      </div>
    </div>
  );
};
```

3. Logical AND (&&) Operator

The `&&` operator is perfect for showing something only when a condition is true:

javascript

```javascript
const Notifications = ({ notifications }) => {
  return (
    <div>
      <h2>Dashboard</h2>
      {notifications.length > 0 && (
        <div className="notification-badge">
          You have {notifications.length} new notifications
        </div>
      )}

      {notifications.length === 0 && (
        <p>No new notifications</p>
      )}
    </div>
  );
};
```

4. Switch Case for Multiple Conditions

javascript

```javascript
const UserRole = ({ role }) => {
  const renderContent = () => {
    switch (role) {
      case 'admin':
        return (
          <div>
            <h2>Admin Dashboard</h2>
            <button>Manage Users</button>
            <button>View Reports</button>
            <button>Settings</button>
          </div>
        );

      case 'moderator':
        return (
          <div>
            <h2>Moderator Panel</h2>
            <button>Review Posts</button>
            <button>Ban Users</button>
          </div>
        );

      case 'user':
        return (
          <div>
            <h2>User Dashboard</h2>
            <button>View Profile</button>
            <button>Edit Settings</button>
          </div>
        );

      default:
        return (
          <div>
            <h2>Access Denied</h2>
            <p>You don't have permission to view this page.</p>
          </div>
        );
    }
  };

  return <div>{renderContent()}</div>;
};
```

Complex Conditional Rendering Examples:

Loading States and Error Handling:

```
javascript
```

```
const DataDisplay = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetchData();
  }, []);

  const fetchData = async () => {
    try {
      setLoading(true);
      const response = await axios.get('https://api.example.com/data');
      setData(response.data);
    } catch (err) {
      setError('Failed to load data');
    } finally {
      setLoading(false);
    }
  };

  // Conditional rendering based on state
  if (loading) {
    return <div>Loading... ⏳</div>;
  }

  if (error) {
    return (
      <div>
        <h2>Oops! Something went wrong 😞</h2>
        <p>{error}</p>
        <button onClick={fetchData}>Try Again</button>
      </div>
    );
  }

  if (!data || data.length === 0) {
    return <div>No data available 📭</div>;
  }

  return (
    <div>
      <h2>Data loaded successfully! ✅</h2>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
};
```

Authentication-based Rendering:

javascript

```
const App = () => {
  const [user, setUser] = useState(null);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    // Check if user is logged in
    checkAuthStatus();
  }, []);

  const checkAuthStatus = () => {
    // Simulate checking authentication
    setTimeout(() => {
      const savedUser = localStorage.getItem('user');
      setUser(savedUser ? JSON.parse(savedUser) : null);
      setIsLoading(false);
    }, 1000);
  };

  // Show loading spinner while checking auth
  if (isLoading) {
    return <div>Checking authentication... 🔄</div>;
  }

  // Conditional rendering based on authentication status
  return (
    <div>
      {user ? (
        // User is logged in - show main app
        <div>
          <header>
            <h1>Welcome, {user.name}!</h1>
            <button onClick={() => setUser(null)}>Logout</button>
          </header>
          <main>
            <h2>Dashboard</h2>
            <p>You have access to all features!</p>
          </main>
        </div>
      ) : (
        // User is not logged in - show login form
        <div>
          <h1>Please Log In</h1>
          <LoginForm onLogin={setUser} />
        </div>
      )}
    </div>
  );
};
```

Best Practices for Conditional Rendering:

1. **Keep conditions simple and readable**
2. **Use descriptive variable names for boolean conditions**
3. **Extract complex conditions into separate functions**
4. **Handle loading and error states properly**
5. **Provide fallback content when data is empty**

```javascript
// Good: Clear and readable
const isUserLoggedIn = user !== null;
const hasNotifications = notifications.length > 0;
const isLoadingComplete = !loading && !error;

return (
  <div>
    {isUserLoggedIn && <UserDashboard />}
    {hasNotifications && <NotificationBell count={notifications.length} />}
    {isLoadingComplete && <MainContent />}
  </div>
);
```

Summary

Key Takeaways:

1. **REST APIs** provide a standardized way for applications to communicate using HTTP methods (GET, POST, PUT, DELETE)
2. **Axios** simplifies making HTTP requests in React applications with better error handling and cleaner syntax
3. **Event Handling** allows your React components to respond to user interactions like clicks, form submissions, and input changes
4. **Conditional Rendering** lets you show different content based on application state, user permissions, or data availability
5. **Best Practices**:
   - Always handle loading and error states
   - Use meaningful variable names

- Keep your code readable and well-organized
- Test your API calls thoroughly
- Provide good user feedback

These concepts work together to create dynamic, interactive React applications that can fetch data from servers and respond to user actions effectively.