Lab 6: React and REST

Step 1: Project Setup

REST and `json-server`

- **`json-server`**: Provides a full fake REST API for testing and prototyping. It watches the db.json file and provides endpoints for CRUD operations.
- **Endpoints**:
  - `GET /products`: Fetch all products.
  - `POST /products`: Add a new product.
  - `PUT /products/:id`: Update an existing product.
  - `DELETE /products/:id`: Delete a product.

React Components

- **App Component**: Manages the state of the application (selected product, adding state) and conditionally renders `ProductList`, `ProductForm`, or `Product` based on the current state.
- **`ProductList` Component**: Fetches and displays the list of products, handles product selection for editing, and deletion.
- **`ProductForm` Component**: Handles the form for adding a new product and posts the new product to the REST API.
- **`Product Component`**: Handles the form for editing an existing product and updates the product via the REST API.

CSS (App.css)

- Provides styling for the application to ensure a clean, consistent look across all components.
- Styles include layout, typography, form elements, buttons, and hover effects.

This setup creates a full-featured, responsive, and visually appealing product management application that interacts seamlessly with a REST API provided by json-server.

1. Create a React Application

First, create a new React application if you haven't already:

2. Set Up `json-server`

Install `json-server` globally, might have to use `sudo`:

```
npm install -g json-server
```

Create a `db.json` file in the root of your project to serve as the database for `json-server`:

```
{
  "products": [
    {
      "id": 1,
      "name": "React Book",
      "description": "Great book about React.",
      "price": "$120",
      "stock": 10
    },
    {
      "id": 2,
      "name": "ES6 Book",
      "description": "The basis of React.",
      "price": "$80",
      "stock": 20
    }
  ]
}
```

Start `json-server` to serve the database:`json-server --watch db.json --port 5000`

This starts a RESTful API server at http://localhost:5000.

Step 2: CSS Styling

src/App.css

Create a CSS file to style your application. You can copy the following:

```
/* src/App.css */

.App {
  font-family: 'Arial', sans-serif;
  padding: 20px;
  max-width: 800px;
  margin: 0 auto;
  background-color:
  #f7f7f7; border-radius:
  10px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}
h1 {
  font-size: 2em;
  margin-bottom:
  20px; color: #333;
}

h2 {
  font-size: 1.5em;
  margin-bottom:
  10px; color: #555;
}

ul {
  list-style-type: none;
  padding: 0;
}

ul li
  { padding:
  10px; margin:
  5px 0;
  border: 1px solid #ccc;
  background-color:
  white; cursor: pointer;
  display: flex;
```

```
  justify-content: space-
  between; align-items: center;
  border-radius: 5px;
}

ul li:hover {
  background-color: #f0f0f0;
}

button {
  padding: 10px
  20px; font-size:
  16px; cursor:
  pointer;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 4px;
  transition: background-color 0.3s ease;
}

button:hover {
  background-color: #0056b3;
}

button:disabled
  { background-color:
  #cccccc; cursor: not-
  allowed;
}

form div {
  margin-bottom:
  15px; display:
  flex; align-items:
  center;
}

form label { flex:
  0 0 120px;
  margin-right:
```

```
  10px; font-weight:
  bold; color: #555;
}

form input[type="text"], form input[type="number"]
  { flex: 1;
  padding: 10px;
  font-size:
  16px;
  border: 1px solid
  #ccc; border-radius:
  4px;
}

.form-container
  { background-color:
  #fff; padding: 20px;
  border: 1px solid
  #ccc; border-radius:
  8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
  margin-bottom: 20px;
}
```

Step 4: Components `src/components/`

`ProductForm.js` Handles the form for

adding new products:

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

function ProductForm({ onAddProduct, onCancel })
  { const [newProduct, setNewProduct] = useState({
    name: '',
    description: '',
    price: '',
    stock: ''
  });

  const handleChange = (e) =>
    { const { name, value } =
```

```
    e.target;
    setNewProduct(prevState => ({
      ...prevState
      , [name]:
      value
    }));
  };

  const handleSubmit = (e) =>
    { e.preventDefault(); fetch('http:// localhost:5000/
    products', {
      method:
      'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(newProduct)
    })
      .then(response => response.json())
      .then(data => {
        console.log('Product added:', data);
        onAddProduct(data);
      })
      .catch(error => console.error('Error adding
product:', error));
  };

  return (
    <div className="form-container">
      <h2>Add New Product</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Name:</label>
          <input type="text" name="name"
value={newProduct.name} onChange={handleChange}
required />
        </div>
        <div>
          <label>Description:</label>
          <input type="text" name="description"
value={newProduct.description} onChange={handleChange}
```

```
required />
        </div>
        <div>
          <label>Price:</label>
          <input type="text" name="price"
value={newProduct.price} onChange={handleChange}
required />
        </div>
        <div>
          <label>Stock:</label>
          <input type="number" name="stock"
value={newProduct.stock} onChange={handleChange}
required />
        </div>
        <button type="submit">Add Product</button>
        <button type="button" onClick={onCancel}
>Cancel</button>
      </form>
    </div>
  );
}
export default ProductForm;
```

src/components/Product.js

Handles the form for editing an existing product:

```
import React, { useState } from

'react'; function Product({ product,

onBack }) {

  const [productData, setProductData] =
useState(product);
  const handleSave = () => {
  fetch(`http://localhost:5000/products/$
{productData.id}`)
  }, { method: 'PUT', headers: {
        'Content-Type': 'application/json'
      },
```

```
      body: JSON.stringify(productData)
    })
      .then(response => response.json())
      .then(data => {
        console.log('Product updated:', data);
        onBack();
      })
      .catch(error => console.error('Error updating
product:', error));
  };

  const handleChange = (e) =>
    { const { name, value } =
    e.target;
    setProductData(prevState => ({
      ...prevState
      , [name]:
      value
    }));
  };
  return (
    <div className="form-container">
      <h2>Edit Product</h2>
      <form>
        <div>
          <label>Name:</label>
          <input type="text" name="name"
value={productData.name} onChange={handleChange} />
        </div>
        <div>
          <label>Description:</label>
          <input type="text" name="description"
value={productData.description} onChange={handleChange}
/>
        </div>
        <div>
          <label>Price:</label>
          <input type="text" name="price"
value={productData.price} onChange={handleChange} />
        </div>
        <div>
```

```
          <label>Stock:</label>
          <input type="number" name="stock"
value={productData.stock} onChange={handleChange} />
        </div>
        <button type="button" onClick={handleSave}
>Save</button>
        <button type="button" onClick={onBack}>Back to
list</button>
      </form>
    </div>
  );
}

export default Product;
```

src/ components/ProductList.js

Displays the list of products:

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

function ProductList({ onSelectProduct })
  { const [products, setProducts] =
  useState([]);

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

  const fetchProducts = () => { fetch('http://
    localhost:5000/products')
      .then(response => response.json())
      .then(data => setProducts(data))
      .catch(error => console.error('Error fetching
products:', error));
  };

  const handleDelete = (productId) => { fetch(`http://
    localhost:5000/products/${productId}`, {})
method: 'DELETE',
```

```
.then(() =>
  { fetchProducts(
  );


        })
        .catch(error => console.error('Error deleting
  product:', error));
    };

    return (
      <div>
        <h1>Product List</h1>
        <ul>
          {products.map((product) => (
```

```
          <li key={product.id}>
            <div className="product-details"
onClick={() => onSelectProduct(product)}>
              {product.name}
            </div>
            <div className="product-actions">
              <button onClick={() =>
onSelectProduct(product)}>Edit</button>
              <button onClick={() =>
handleDelete(product.id)}>Delete</button>
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default ProductList;
```

//src/App.js (Main application component):

```
import React, { useState } from 'react';
import ProductList from './components/ProductList';
import Product from './components/Product';
import ProductForm from './components/ProductForm';
import './App.css';

function App() {
  const [selectedProduct, setSelectedProduct] =
useState(null);
  const [isAdding, setIsAdding] = useState(false);

  const handleSelectProduct = (product) =>
    { setSelectedProduct(product);
  };

  const handleBack = () =>
    { setSelectedProduct(null);
    setIsAdding(false);
  };
```

```
  const handleAddProduct = (product) =>
    { setIsAdding(false);
    setSelectedProduct(null); // Clear selected product
  };

  const handleStartAdding = () =>
    { setIsAdding(true);
  };

  return (
    <div className="App">
      {isAdding ? (
        <ProductForm onAddProduct={handleAddProduct}
onCancel={handleBack} />
      ) : selectedProduct ? (
        <Product product={selectedProduct}
onBack={handleBack} />
      ) : (
        <>
          <ProductList
onSelectProduct={handleSelectProduct} />
          <button onClick={handleStartAdding}>Add
Product</button>
        </>
      )}
    </div>
  );
}

export default App;
```

Running the Application

1. **Start `json-server` and leave it running. You might have to run this from the `/src` directory or have the `db.json` file in the root directory for this command to find it**:

   ```
   json-server --watch db.json --port 5000
   ```

2. ***In another terminal window*, start the development server after**

after navigating to your application:

```
npm run dev
```

Lab 7: Use REST and `axios`

Benefits of Using Axios:

- Axios uses a more concise and intuitive syntax compared to the native fetch API.
- Axios automatically parses JSON responses.
- Axios provides a clear way to handle errors using try...catch blocks, improving the robustness of your application.
- Axios allows you to use interceptors to modify requests or responses globally, which can be helpful for tasks like authentication or logging.
- Axios works seamlessly across different browsers, ensuring consistent behavior.
- Axios has a large and active community, making it easy to find help and resources

Step 1: Install Axios in your application

```
1pm install axios
```

Step 2: We will replace Fetch API calls with axis methods

Axios methods are `axios.get`, `axios.post`, `axios.put`, and `axios.delete`. Axios methods directly return the response data.

The `handleSubmit`, `handleSave`, `fetchProducts`, and `handleDelete` functions are declared as `async` to enable the use of `await` for cleaner asynchronous code.

`Product.js`

1. Import the axis library: `import axios from 'axios';`
2. `handleSave` function:
    - Now declared as `async` to enable the use of `await`.

- Inside a `try...catch` block:
  - Send a PUT request with `const response = await axios.put(...)` to the specified URL (`http://localhost:5000/products/${productData.id}`) using Axios. Include the updated `productData` in the request body. Use `await` keyword to pause execution until the promise returned by `axios.put` is resolved, either successfully or with an error.
  - If the request is successful, the updated product data from the server's response is logged to the console: `console.log('Product updated:', response.data);`
  - The `onBack` prop function is called to navigate back to the product list: `onBack();`
  - If there's an error during the API call, the error is caught and logged to the console: `catch (error)`

3. The rest of the component should be the same as without Axios, handling the form input changes and rendering the edit product form.

```
import React, { useState } from

'react'; import axios from 'axios';

function Product({ product, onBack }) {

  // ... code from previous Lab

  const handleSave = async () => {

    try {

const response = await axios.put(`http://
localhost:5000/products/${productData.id}`,
productData);

      console.log('Product updated:', response.data);

      onBack();

    } catch (error) {
```

```
      console.error('Error updating product:', error);
    }
  };
  // ... code from previous Lab
}
```

ProductList.js

1. `import axios from 'axios';`
2. `fetchProducts` function:
    - It's also declared as `async`.
    - Inside a `try...catch` block:
        - Send a GET request to fetch all products from the server. Use `await` to wait for the response.
          `const response = await axios.get(...)`
        - Set the fetched product data in the component's state using the `setProducts` function:
          `setProducts(response.data);`
        - Handle potential errors during the API call.
        - `catch (error)`
3. `handleDelete` function:
    - Declare as `async`.
    - Inside a `try...catch` block:
        - Send a `DELETE` request to remove the specified product.
          `await axios.delete(...)`
        - After successful deletion, the product list is refreshed by calling `fetchProducts` again.
          `fetchProducts();`
        - Handle any errors:
          `catch (error)`
4. The rest of the component is similar to one from the previous lab.

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

import axios from 'axios';

function ProductList({ onSelectProduct }) {
```

```
  // ... code from previous lab

  const fetchProducts = async () =>
    { try {
      const response = await axios.get('http://
localhost:5000/products');
      setProducts(response.data);
    } catch (error) {
      console.error('Error fetching products:', error);
    }
  };

  const handleDelete = async (productId) =>
    { try {
      await axios.delete(`http://localhost:5000/
products/${productId}`);
      fetchProducts();
    } catch (error) {
      console.error('Error deleting product:', error);
    }
  };
  // ... code from previous lab
}
```

ProductForm.js

1. `import axios from 'axios';`
2. `handleSubmit` function:

- Declare as `async`.
- Inside a `try...catch` block:
  - Send a `POST` request to create a new product, including the `newProduct` data in the request body.

```
const response = await axios.post(...)
```

  - Log the newly created product data.

```
console.log('Product added:', response.data);
```

  - Call the `onAddProduct` prop function to update the parent component's state with the new product.

```
onAddProduct(response.data);
```

- Handle the errors: `catch (error)`

3. The rest of the component handles form input changes and rendering the new product form.

```
import React, { useState } from

'react'; import axios from 'axios';

function ProductForm({ onAddProduct, onCancel }) {

  // ... (code from previous lab)

  const handleSubmit = async (e) => {

    e.preventDefault();

    try {

      const response = await axios.post('http://
localhost:5000/products', newProduct);

      console.log('Product added:', response.data);
      onAddProduct(response.data);

    } catch (error) {

      console.error('Error adding product:', error);

    }

  };

  // ... (code from previous lab)

}
```

Remember to keep `json-server` running `json-server --watch db.json --port 5000` - you might have to reinstall it and re-create the `db.json` file if you started a new application. Start your React application *in a di*ff*erent terminal* with `npm start`

Lab 8: Unit Testing

Step 1: `vite.config.js`

This file configures the **Vite** build tool and **Vitest** testing framework. It tells Vite to use the React plugin for building React apps and sets up Vitest for testing.

- **`plugins: [react()]`**: This enables Vite to handle React-specific optimizations like JSX syntax.
- **`test`**:
  - **`globals: true`**: Automatically makes global functions like `describe`, `it`, and `expect` available in your test files without needing to import them explicitly.
  - **`environment: 'jsdom'`**: Specifies that **Vitest** should use a browser-like environment (jsdom) for testing React components, since React interacts with the DOM.
  - **`setupFiles: './vitest.setup.js'`**: Tells Vitest to run `vitest.setup.js` before running the tests. This file can be used to import global test utilities and configure mocks.

`vite.config.js`

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

// Vite configuration with Vitest setup
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true, // Allows global test functions like
describe, it, expect, etc.
    environment: 'jsdom', // Use jsdom for browser-like
environment
    setupFiles: './vitest.setup.js', // Specify the setup
file for global configurations
  },
});
```

Step 2: `vitest.setup.js`

This setup file is used to configure global test settings and imports before tests run. Here, we import **`@testing-library/jest-dom`** to extend the `expect` API with useful matchers for DOM assertions. Additionally, you can use **`vi`** from **Vitest** to mock global functions or modules.

- **`@testing-library/jest-dom`**: This package provides custom DOM matchers such as `toBeInTheDocument()`, `toHaveTextContent()`, `toHaveClass()`, etc. These matchers make it easier to test React components by working directly with the DOM.
- **`vi` from Vitest**: Vitest's mock function, useful for mocking network requests (`axios`), global objects, and other utility functions. You can mock any functions that are called in your component (like `axios.get` or `console.log`).

`vitest.setup.js`

```
import '@testing-library/jest-dom';
// Extends jest with useful matchers for DOM elements like
`toBeInTheDocument`
import { vi } from 'vitest';
// Optional: for mocking global methods
```

Step 3: Test File for `Product.test.js`

This file contains tests for your `Product` component, checking the following functionality:

- Rendering the product details.
- Editing product data and saving it.
- Handling the back button functionality.
- Handling errors when saving the product.

1. Mocking Axios

At the top of the file, you import **`axios`** and use **Vitest's `vi.mock()`** to mock the **`axios.put()`** method.

```
import axios from 'axios';
import { vi } from 'vitest';
// Mock axios PUT request
```

```
vi.mock('axios');
```

- **`vi.mock('axios')`**: This ensures that any calls to **`axios`** inside the tests are mocked, so no real API requests are made. Instead, you simulate responses for testing.

2. Test Setup (beforeEach)

In the **`beforeEach`** function, you clear the mock function `mockOnBack` before each test to ensure no leftover state from previous tests.

```
beforeEach(() => {
  mockOnBack.mockClear(); // Clear previous calls before
each test
});
```

`mockOnBack` is used to simulate a callback function passed to the `Product` component, and you want to reset its state before each test to prevent any interactions from affecting subsequent tests.

3. Test 1: Rendering Product Details

This test checks that the product details are rendered correctly when the component is first loaded.

- **`render(<Product product={product} onBack={mockOnBack} />)`**: This renders the `Product` component with the test `product` data.
- **`expect(screen.getByLabelText(...))`**: These assertions check that the values inside the input fields match the expected initial values (`name`, `description`, `price`, and `stock`).

```
test('renders product details', () => {
  render(<Product product={product} onBack={mockOnBack} />
);

  // Check if the input fields are rendered with the
correct initial values
  expect(screen.getByLabelText(/
Name:/)).toHaveValue('Product 1');
  expect(screen.getByLabelText(/
Description:/)).toHaveValue('Product description');
expect(screen.getByLabelText(/Price:/)).toHaveValue('100');
```

```
  expect(screen.getByLabelText(/Stock:/)).toHaveValue(10);
});
```

4. Test 2: Editing Product and Saving

This test simulates a user editing the product details and clicking the **Save** button. It then checks if the **`axios.put()`** method was called with the correct data and if the `onBack` function is called.

- **`fireEvent.change()`** simulates the user entering new data into the input fields.
- **`fireEvent.click()`** simulates clicking the **Save** button.
- **`axios.put.mockResolvedValue()`**: This mocks a successful `PUT` request, returning the updated `product` object as the response.
- **`await waitFor()`**: This ensures that the assertions happen after the asynchronous operations (e.g., the network request) have completed.

Assertions:

- **`expect(axios.put).toHaveBeenCalledWith()`**: Verifies that **`axios.put()`** was called with the correct URL and product data.
- **`expect(mockOnBack).toHaveBeenCalled()`**: Verifies that the **`onBack`** function was called after the product was saved.

```
test('allows user to edit product and save', async () => {
  render(<Product product={product} onBack={mockOnBack} /
>);

  // Change product data in the input fields
  fireEvent.change(screen.getByLabelText(/Name:/),
{ target: { value: 'Updated Product' } });
  fireEvent.change(screen.getByLabelText(/Description:/), {
target: { value: 'Updated description' } });
  fireEvent.change(screen.getByLabelText(/Price:/),
{ target: { value: '200' } });
  fireEvent.change(screen.getByLabelText(/Stock:/),
{ target: { value: 5 } });

  // Simulate save action
  fireEvent.click(screen.getByText(/Save/));
```

```
  // Mock axios PUT request response
  axios.put.mockResolvedValue({ data: product });

  await waitFor(() => {
    // Verify that the save function was called
    expect(axios.put).toHaveBeenCalledWith(
      'http://localhost:5000/products/1',
      expect.objectContaining({
        id: 1,
        name: 'Updated Product',
        description: 'Updated description',
        price: '200',
        stock: 5,
      })
    );
    expect(mockOnBack).toHaveBeenCalled();
  });
});
```

5. Test 3: Clicking the Back Button

This test checks if the **Back to list** button calls the `onBack` function.

- **`fireEvent.click()`**: Simulates clicking the **Back to list** button.
- **`expect(mockOnBack).toHaveBeenCalled()`**: Verifies that the **`onBack`** function is called when the button is clicked.

```
test('calls onBack when Back to list button is clicked', ()
=> {
  render(<Product product={product} onBack={mockOnBack} /
>);

  // Simulate the back button click
  fireEvent.click(screen.getByText(/Back to list/));

  // Verify that the onBack function is called
  expect(mockOnBack).toHaveBeenCalled();
});
```

6. Test 4: Handling Errors During Product Save

This test simulates a failed **`PUT`** request and checks if the error is handled properly by displaying an error message.

- **`axios.put.mockRejectedValue()`**: This mocks a failed **`PUT`** request by rejecting it with an error (`Network Error`).
- **`await waitFor()`**: Ensures the error message appears after the async operation completes.
- **`expect(screen.getByText(/Error updating product/)).toBeInTheDocument()`**: Verifies that the error message is displayed on the screen when the save operation fails.

```
test('handles errors during product save', async () => {
  render(<Product product={product} onBack={mockOnBack} /
>);

  // Simulate saving the product
  fireEvent.click(screen.getByText(/Save/));

  // Simulate a failed request
  axios.put.mockRejectedValue(new Error('Network Error'));

  await waitFor(() => {
    // Verify error handling behavior
    expect(screen.getByText(/Error updating
product/)).toBeInTheDocument();
  });
});
```

Review of Test Structure

1. **Mocking Dependencies**:
    - **`vi.mock('axios')`**: Mocks `axios` to prevent real network requests.
    - **`axios.put.mockResolvedValue()`**: Simulates a successful network response.
    - **`axios.put.mockRejectedValue()`**: Simulates a failed network response.
2. **Rendering and User Interaction**:

- **`render(<Product />)`**: Renders the `Product` component.
- **`fireEvent.change()`**: Simulates user input in the form fields.
- **`fireEvent.click()`**: Simulates clicking the save button and the back button.

3. **Assertions**:
    - Verifies that **axios** is called with the correct data.
    - Verifies that the `onBack` function is called after the product is saved.
    - Verifies that the error message appears when the save fails.

Step 4: `ProductForm.test.jsx` tests the core functionality of the **`ProductForm`** component.

1. Mocking Axios:

At the top of the file, you mock the **`axios`** library using **Vitest's `vi.mock()`** method. This ensures that **`axios`** requests do not actually hit the server during the tests.

- **`vi.mock('axios')`**: This prevents the actual HTTP requests from being made and allows you to simulate the response for your tests. This is crucial for unit tests because it avoids making real network calls and ensures tests are repeatable and fast.

```
import axios from 'axios';
import { vi } from 'vitest';

// Mocking axios POST request
vi.mock('axios');
```

2. Test 1: Rendering Form with Empty Fields

This test checks whether the form fields are rendered with empty initial values when the form is first loaded.

- **`render(<ProductForm />)`**: This renders the `ProductForm` component and allows you to interact with it in the test.

- **`expect(screen.getByLabelText(...)).toHaveValue('')`**: These assertions check if the form fields (`Name`, `Description`, `Price`, `Stock`) are empty when the form is first rendered.

```
test('renders form with empty fields initially', () => {
  render(<ProductForm onAddProduct={mockOnAddProduct}
onCancel={mockOnCancel} />);

  // Check if the form fields are rendered with empty
values
  expect(screen.getByLabelText(/Name:/)).toHaveValue('');
  expect(screen.getByLabelText(/
Description:/)).toHaveValue('');
  expect(screen.getByLabelText(/Price:/)).toHaveValue('');
  expect(screen.getByLabelText(/Stock:/)).toHaveValue('');
});
```

3. Test 2: Allowing User to Fill in the Form

This test simulates user input by filling in the form fields and verifies that the form values have been updated accordingly.

- **`fireEvent.change()`**: This simulates the user typing into the form fields (such as `Name`, `Description`, `Price`, `Stock`).
- **`expect(...).toHaveValue()`**: These assertions check if the values of the form fields are updated as expected after the user input.

```
test('allows user to fill in the form', () => {
  render(<ProductForm onAddProduct={mockOnAddProduct}
onCancel={mockOnCancel} />);

  // Simulate user filling in the form
  fireEvent.change(screen.getByLabelText(/Name:/),
{ target: { value: 'New Product' } });
  fireEvent.change(screen.getByLabelText(/Description:/), {
target: { value: 'This is a great product' } });
  fireEvent.change(screen.getByLabelText(/Price:/),
{ target: { value: '99.99' } });
  fireEvent.change(screen.getByLabelText(/Stock:/),
{ target: { value: 50 } });
```

```
  // Verify that the form fields have been updated
  expect(screen.getByLabelText(/Name:/)).toHaveValue('New
Product');
  expect(screen.getByLabelText(/
Description:/)).toHaveValue('This is a great product');
  expect(screen.getByLabelText(/
Price:/)).toHaveValue('99.99');
  expect(screen.getByLabelText(/Stock:/)).toHaveValue(50);
});
```

4. Test 3: Submitting the Form and Calling `onAddProduct`

This test checks the behavior when the user submits the form. It verifies that the correct API request is made and that the `onAddProduct` callback is called.

- **`fireEvent.submit()`**: This simulates submitting the form.
- **`axios.post.mockResolvedValue()`**: Mocks the **POST** request to the server, simulating a successful API call with the provided product data.
- **`expect(axios.post).toHaveBeenCalledWith()`**: Verifies that the **POST** request was made with the correct product data.
- **`expect(mockOnAddProduct).toHaveBeenCalledWith()`**: Verifies that the `onAddProduct` callback was called with the correct data after the form submission.

```
test('submits the form and calls onAddProduct', async () =>
{
  render(<ProductForm onAddProduct={mockOnAddProduct}
onCancel={mockOnCancel} />);

  // Fill in the form
  fireEvent.change(screen.getByLabelText(/Name:/),
{ target: { value: 'New Product' } });
  fireEvent.change(screen.getByLabelText(/Description:/), {
target: { value: 'This is a great product' } });
  fireEvent.change(screen.getByLabelText(/Price:/),
{ target: { value: '99.99' } });
  fireEvent.change(screen.getByLabelText(/Stock:/),
{ target: { value: 50 } });

  // Mock the axios POST request
```

```
  axios.post.mockResolvedValue({ data: { id:
1, ...screen.getByLabelText('Name:').value } });

  // Simulate form submission
  fireEvent.submit(screen.getByRole('form'));

  await waitFor(() => {
    // Verify the axios POST request
    expect(axios.post).toHaveBeenCalledWith('http://
localhost:5000/products', {
      name: 'New Product',
      description: 'This is a great product',
      price: '99.99',
      stock: 50,
    });

    // Verify that the onAddProduct callback was called
with the correct data
    expect(mockOnAddProduct).toHaveBeenCalledWith({
      id: 1,
      name: 'New Product',
      description: 'This is a great product',
      price: '99.99',
      stock: 50,
    });
  });
});
```

5. Test 4: Calling `onCancel` When Cancel Button Is Clicked

This test verifies that the `onCancel` function is called when the **Cancel** button is clicked.

- **`fireEvent.click()`**: This simulates the click event on the **Cancel** button.
- **`expect(mockOnCancel).toHaveBeenCalled()`**: This verifies that the `onCancel` callback was triggered when the user clicked the cancel button.

```
test('calls onCancel when Cancel button is clicked', () =>
{
```

```
  render(<ProductForm onAddProduct={mockOnAddProduct}
onCancel={mockOnCancel} />);

  // Simulate clicking the Cancel button
  fireEvent.click(screen.getByText(/Cancel/));

  // Verify that the onCancel callback was called
  expect(mockOnCancel).toHaveBeenCalled();
});
```

6. Test 5: Handling Errors During Form Submission

This test simulates a failure when saving the product (such as a network error). It checks if the error message is displayed correctly.

- **`axios.post.mockRejectedValue()`**: Mocks a failed **POST** request (e.g., a network error).
- **`await waitFor()`**: Ensures that the error message appears after the failed request.
- **`expect(screen.getByText(/Error updating product/)).toBeInTheDocument()`**: Verifies that an error message is displayed when the request fails.

```
test('handles errors during product save', async () => {
  render(<ProductForm onAddProduct={mockOnAddProduct}
onCancel={mockOnCancel} />);

  // Simulate saving the product
  fireEvent.click(screen.getByText(/Save/));

  // Simulate a failed request
  axios.post.mockRejectedValue(new Error('Network Error'));

  await waitFor(() => {
    // Verify error handling behavior
    expect(screen.getByText(/Error updating
product/)).toBeInTheDocument();
  });
});
```

Review of the Tests

1. **Test 1: `renders form with empty fields initially`**:
    - Ensures that the form fields are empty when the form is first rendered.
2. **Test 2: `allows user to fill in the form`**:
    - Simulates user input in the form and checks that the form fields are updated with the new values.
3. **Test 3: `submits the form and calls onAddProduct`**:
    - Verifies that submitting the form triggers the correct **axios** request and calls the `onAddProduct` function with the correct data.
4. **Test 4: `calls onCancel when Cancel button is clicked`**:
    - Verifies that the `onCancel` callback is called when the **Cancel** button is clicked.
5. **Test 5: `handles errors during product save`**:
    - Simulates a failed **POST** request and checks that an error message is shown.

Step 5: The **`ProductList.test.jsx`** file will be a test suite for your **`ProductList`** component. It checks multiple scenarios, including rendering the product list, selecting a product, deleting a product, and handling errors.

1. Mocking Axios

The **`axios`** module is mocked using **Vitest's `vi.mock()`** method to avoid making real network requests.

```
import axios from 'axios';
import { vi } from 'vitest';

// Mock axios GET and DELETE requests
vi.mock('axios');
```

- **`vi.mock('axios')`**: This ensures that **`axios.get()`** and **`axios.delete()`** calls are intercepted and replaced with mock implementations, allowing you to simulate various responses without making actual network requests.

2. Setup for Tests (`beforeEach`)

In the **`beforeEach()`** block, the mock function `mockOnSelectProduct` is cleared before each test to ensure there's no leftover state between tests.

```
beforeEach(() => {
  mockOnSelectProduct.mockClear();
});
```

- This prevents interactions from one test affecting others, ensuring each test runs in a clean state.

3. Test 1: Rendering the List of Products

This test checks whether the list of products is correctly rendered when the component is loaded.

- **`axios.get.mockResolvedValue({ data: products })`**: This mocks a successful response from the `axios.get()` call with the `products` data.
- **`render(<ProductList />)`**: This renders the `ProductList` component and allows us to interact with it in the test.
- **`await waitFor()`**: This ensures the assertions are run after the products are rendered asynchronously. This is important because data is being fetched (mocked by `axios.get`), and you need to wait for the DOM updates.
- **`expect(screen.getByText(...))`**: Verifies that the product names are present in the document.

```
test('renders the list of products', async () => {
  axios.get.mockResolvedValue({ data: products });

  render(<ProductList onSelectProduct={mockOnSelectProduct}
/>);

  // Wait for the products to be rendered
  await waitFor(() => {
    expect(screen.getByText('Product
1')).toBeInTheDocument();
```

```
    expect(screen.getByText('Product
2')).toBeInTheDocument();
  });
});
```

4. Test 2: Selecting a Product

This test ensures that when a product is clicked, the `onSelectProduct` callback is called with the correct product.

- **`fireEvent.click()`**: This simulates a click on **Product 1**.
- **`expect(mockOnSelectProduct).toHaveBeenCalledWith()`**: Verifies that the `onSelectProduct` function is called with the correct product data when a product is clicked.

```
test('calls onSelectProduct when a product is clicked',
async () => {
  axios.get.mockResolvedValue({ data: products });

  render(<ProductList onSelectProduct={mockOnSelectProduct}
/>);

  // Wait for the products to be rendered
  await waitFor(() => {
    fireEvent.click(screen.getByText('Product 1'));
  });

  // Verify that onSelectProduct was called with the
correct product
  expect(mockOnSelectProduct).toHaveBeenCalledWith({ id: 1,
name: 'Product 1' });
});
```

5. Test 3: Handling Product Deletion

This test verifies that when a product is deleted, the **`DELETE`** request is made to the correct endpoint, and the product list is refreshed.

- **`axios.delete.mockResolvedValue({})`**: Mocks a successful response for the **`DELETE`** request.

- **`fireEvent.click()`**: Simulates a click on the **Delete** button for **Product 1**.
- **`expect(axios.delete).toHaveBeenCalledWith()`**: Verifies that the **`DELETE`** request is sent to the correct URL with the correct product ID (`1`).
- **`expect(axios.get).toHaveBeenCalledTimes(4)`**: Verifies that **`axios.get`** was called 4 times: once on mount to fetch the initial products and three more times due to the state updates (or product list refresh) after deletion.

```
test('handles product deletion and refreshes the list',
async () => {
  axios.get.mockResolvedValue({ data: products });
  axios.delete.mockResolvedValue({});

  render(<ProductList onSelectProduct={mockOnSelectProduct}
/>);

  // Wait for the products to be rendered
  await waitFor(() => {
    expect(screen.getByText('Product
1')).toBeInTheDocument();
    expect(screen.getByText('Product
2')).toBeInTheDocument();
  });

  // Simulate the delete button click for Product 1
  fireEvent.click(screen.getAllByText('Delete')[0]);

  // Verify that the axios DELETE request was called with
the correct product ID
  await waitFor(() => {
    expect(axios.delete).toHaveBeenCalledWith('http://
localhost:5000/products/1');
  });

  // Verify that after deleting, the list of products is
refreshed
  expect(axios.get).toHaveBeenCalledTimes(4); // Once on
mount, once after delete
```

```
});
```

6. Test 4: Handling Errors When Fetching Products

This test simulates a failed request when fetching products and ensures that the UI handles it gracefully.

- **`axios.get.mockRejectedValue()`**: This mocks a failed **`GET`** request by rejecting it with an error (`Network Error`).
- **`await waitFor()`**: Ensures that the error handling is executed after the asynchronous operation (the failed request).
- **`expect(screen.queryByText(...)).toBeNull()`**: Verifies that the product names are not present in the document when the request fails. This suggests that the product list isn't rendered when there's an error fetching the products.

```
test('handles error when fetching products', async () => {
  axios.get.mockRejectedValue(new Error('Network Error'));

  render(<ProductList onSelectProduct={mockOnSelectProduct}
/>);

  // Verify that the error is handled gracefully (in this
case, by logging an error)
  await waitFor(() => {
    expect(screen.queryByText('Product 1')).toBeNull();
    expect(screen.queryByText('Product 2')).toBeNull();
  });
});
```

Review of the Tests

- **Test 1: `renders the list of products`**:
    - Verifies that the products are rendered correctly after being fetched from the server.
- **Test 2: `calls onSelectProduct when a product is clicked`**:
    - Ensures that clicking a product calls the `onSelectProduct` callback with the correct product.

- **Test 3: `handles product deletion and refreshes the list`**:
    - Verifies that the **DELETE** request is made when a product is deleted, and the list is refreshed.
- **Test 4: `handles error when fetching products`**:
    - Ensures that the component gracefully handles errors during the data fetching process.

Lab 9 (Optional): Using ES6

Step 1: Create the Project Structure for an ES6-based task list

1. **Create a Project Folder**
    - Create a new folder on your computer. Name it `todo-app`.
2. **Create HTML, CSS, and JS Files**
    - Inside the `todo-app` folder, create three new files: `index.html`, `styles.css`, and `app.js`.

Step 2: Add the HTML Code

1. **Open index.html**: Open the index.html file in a text editor

Copy the HTML Code: Copy and paste this HTML code into the index.html file:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>To-Do List App</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h1>To-Do List</h1>
        <div class="input-container">
            <input type="text" id="todo-input"
placeholder="Add a new task...">
            <button id="add-btn">Add</button>
        </div>
        <ul id="todo-list"></ul>
    </div>
    <script src="app.js"></script>
</body>
</html>
```

Step 3: Add the CSS Code

1. **Open styles.css**: Open the styles.css file in your text editor.

CSS Code: You can just copy and paste the following CSS code into the styles.css file to make this look more user-friendly:

```
body {

    font-family: Arial, sans-serif;
    display: flex;
    justify-content:
    center; align-items:
    center; height: 100vh;
    margin: 0;
    background-color: #f0f0f0;
}

.container {
    background-color: white;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);
    width: 300px;
    text-align: center;
}

.input-container
    { display: flex;
    margin-bottom:
    20px;
}

#todo-input {
    flex: 1;
```

```
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 4px 0 0
    4px;
}

#add-btn {
    padding: 10px;
    border: none;
    background-color: #28a745;
    color: white;
    cursor: pointer;
    border-radius: 0 4px 4px 0;
}

#add-btn:hover {
    background-color: #218838;
}

#todo-list {
    list-style:
    none; padding:
    0;
}

.todo-item {
    display: flex;
    justify-content: space-
    between; align-items: center;
    padding: 10px;
    border: 1px solid
    #ccc; border-radius:
    4px; margin-bottom:
    10px;
}

.todo-item button
    { background-color:
    #dc3545; color: white;
    border: none;
```

```
    padding: 5px 10px;
    cursor: pointer;
    border-radius:
    4px;
    }

.todo-item button:hover
    { background-color:
    #c82333;
}
```

Step 4: Add the JavaScript Code

1. **Open `app.js`**: Open the `app.js` file in your text editor.

Copy the JavaScript Code: Copy and paste the following JavaScript code into the `app.js` file:

```
class TodoApp

    { constructor()

        { this.todos = [];
        this.todoInput = document.getElementById('todo-

input')
;

        this.todoList =
list');

        document.getElementById('todo- this.addButton
btn');

        = document.getElementById('add-



        this.addButton.addEventListener('click', () =>
this.addTodo());
        this.todoInput.addEventListener('keypress',
(event) => {
            if (event.key === 'Enter') this.addTodo();
```

```
        });
    }

    addTodo() {
        const todoText = this.todoInput.value.trim();
        if (todoText) {
            const todo = {
                id: Date.now(),
                text: todoText
            };
            this.todos.push(todo);
            this.renderTodoList();
            this.todoInput.value = '';
        }
    }

    removeTodo(id) {
        this.todos = this.todos.filter(todo =>
todo.id !== id);
        this.renderTodoList();
    }

    renderTodoList()
        { this.todoList.innerHTML = '';
        this.todos.forEach(todo => {
            const li =
            document.createElement('li');
            li.className = 'todo-item';
            li.innerHTML = `
                <span>${todo.text}</span>
                <button onclick="app.removeTodo($
{todo.id})">Delete</button>
            `;
            this.todoList.appendChild(li);
        });
    }
}

const app = new TodoApp();
```

Step 5: Run the Application

1. **Open the Project in a Browser**:
    - Navigate to the todo-app folder.
    - Double-click the index.html file to open it in your default web browser.
    - Alternatively, you can right-click on the index.html file and select "Open with" and choose your preferred web browser.
2. **Test the Application**:
    - In the browser, you should see a simple to-do list interface.
    - Enter a task in the input field and click the "Add" button or press "Enter" to add the task to the list.
    - Each task will appear below the input field with a "Delete" button next to it.
    - Click the "Delete" button to remove a task from the list.