Full-Stack Development Handbook

Java Spring Boot + React + MySQL

📚 Core Concepts (Quick Read)

Client-Server Architecture: Your React app (client) talks to your Spring Boot app (server) which talks to MySQL database.

REST API: A way for apps to talk using HTTP methods:

- GET = Read data
- POST = Create data
- PUT = Update data
- DELETE = Remove data

Request/Response Cycle: React sends request → Spring Boot processes → MySQL stores/retrieves → Spring Boot sends response → React displays

State Management: How React remembers and shares data between components

---

Chapter 1: Project Setup

🎯 **Objective**: Create a organized project folder structure

Step 1 → Create main project folder

```bash
mkdir task-manager-app
cd task-manager-app
```

Creates your main project container

Step 2 → Create backend folder

```bash
mkdir backend
```

Holds your Java Spring Boot API

Step 3 → Create frontend folder

```bash
mkdir frontend
```

Holds your React application

Step 4 → Generate Spring Boot project

1. Go to https://start.spring.io/
2. Choose: Maven, Java 17, Spring Boot 3.2.x
3. Add dependencies: Spring Web, Spring Data JPA, MySQL Driver
4. Generate and extract to `backend` folder

Gets you a working Spring Boot template

Step 5 → Create React app

```bash
cd frontend
npx create-react-app .
npm install axios react-router-dom
```

Gets you a working React template with HTTP client

✅ **Chapter 1 Complete**: You now have organized project structure with backend and frontend ready to develop.

Chapter 2: Database & Spring Boot API

🎯 **Objective**: Connect Spring Boot to MySQL and create your first API endpoint

Step 1 → Configure database connection

Create `backend/src/main/resources/application.properties`:

```properties
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/taskdb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Tells Spring Boot how to connect to MySQL

Step 2 → Create Task entity

Create `backend/src/main/java/com/example/demo/Task.java`:

```java
java
```

```
package com.example.demo;

import jakarta.persistence.*;

@Entity
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    private boolean completed;

    // Constructors
    public Task() {}

    public Task(String title, String description) {
        this.title = title;
        this.description = description;
        this.completed = false;
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }

    public boolean isCompleted() { return completed; }
    public void setCompleted(boolean completed) { this.completed = completed; }
}
```

Creates your database table structure

Step 3 → Create repository

Create `backend/src/main/java/com/example/demo/TaskRepository.java`:

```
java
```

```java
package com.example.demo;

import org.springframework.data.jpa.repository.JpaRepository;

public interface TaskRepository extends JpaRepository<Task, Long> {
}
```

Handles database operations automatically

Step 4 → Create REST controller

Create `backend/src/main/java/com/example/demo/TaskController.java`:

```
java
```

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/tasks")
public class TaskController {

    @Autowired
    private TaskRepository taskRepository;

    @GetMapping
    public List<Task> getAllTasks() {
        return taskRepository.findAll();
    }

    @PostMapping
    public Task createTask(@RequestBody Task task) {
        return taskRepository.save(task);
    }

    @PutMapping("/{id}")
    public Task updateTask(@PathVariable Long id, @RequestBody Task task) {
        task.setId(id);
        return taskRepository.save(task);
    }

    @DeleteMapping("/{id}")
    public void deleteTask(@PathVariable Long id) {
        taskRepository.deleteById(id);
    }
}
```

Creates API endpoints that React can call

Step 5 → Test your API

```bash
cd backend
./mvnw spring-boot:run
```

Starts your API server on http://localhost:8080

✅ **Chapter 2 Complete**: Your Spring Boot API is connected to MySQL and ready to handle CRUD operations.

Chapter 3: CORS Configuration

🎯 **Objective**: Allow React (port 3000) to talk to Spring Boot (port 8080)

Step 1 → Global CORS configuration

Create `backend/src/main/java/com/example/demo/WebConfig.java`:

```java
package com.example.demo;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*");
    }
}
```

Allows all API routes to accept requests from React

Step 2 → Controller-level CORS (alternative)

Add to your TaskController:

```java
```

```
@CrossOrigin(origins = "http://localhost:3000")
@RestController
@RequestMapping("/api/tasks")
public class TaskController {
    // ... rest of your controller
}
```

Allows only this controller to accept requests from React

✅ **Chapter 3 Complete**: React can now successfully communicate with your Spring Boot API.

Chapter 4: React API Integration

🎯 **Objective**: Make React talk to your Spring Boot API using Axios

Step 1 → Create API service

Create `frontend/src/api/taskService.js`:

```javascript
import axios from 'axios';

const API_URL = 'http://localhost:8080/api/tasks';

export const taskService = {
    getAllTasks: () => axios.get(API_URL),
    createTask: (task) => axios.post(API_URL, task),
    updateTask: (id, task) => axios.put(`${API_URL}/${id}`, task),
    deleteTask: (id) => axios.delete(`${API_URL}/${id}`)
};
```

Centralizes all your API calls

Step 2 → Create basic task component

Create `frontend/src/components/TaskList.js`:

```javascript
```

```
import React, { useState, useEffect } from 'react';
import { taskService } from '../api/taskService';

function TaskList() {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

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

  const fetchTasks = async () => {
    try {
      setLoading(true);
      const response = await taskService.getAllTasks();
      setTasks(response.data);
    } catch (err) {
      setError('Failed to fetch tasks');
    } finally {
      setLoading(false);
    }
  };

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      <h2>Tasks</h2>
      {tasks.map(task => (
        <div key={task.id}>
          <h3>{task.title}</h3>
          <p>{task.description}</p>
          <p>Status: {task.completed ? 'Done' : 'Pending'}</p>
        </div>
      ))}
    </div>
  );
}

export default TaskList;
```

Displays tasks from your API with proper loading and error handling

Step 3 → Update App.js

Replace `frontend/src/App.js`:

```javascript
import React from 'react';
import TaskList from './components/TaskList';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>Task Manager</h1>
        <TaskList />
      </header>
    </div>
  );
}

export default App;
```

Displays your task list component

Step 4 → Test the connection

```bash
cd frontend
npm start
```

Starts React on http://localhost:3000 and should show tasks from your API

✅ **Chapter 4 Complete**: React is now successfully fetching and displaying data from your Spring Boot API.

Chapter 5: HTTP Methods & Async Handling

🎯 **Objective**: Master different HTTP methods and handle async operations properly

Step 1 → Understand HTTP methods

javascript

```javascript
// GET - Read data (no body needed)
const getTasks = async () => {
  const response = await axios.get('/api/tasks');
  return response.data;
};

// POST - Create data (body contains new data)
const createTask = async (newTask) => {
  const response = await axios.post('/api/tasks', newTask);
  return response.data;
};

// PUT - Update data (body contains updated data)
const updateTask = async (id, updatedTask) => {
  const response = await axios.put(`/api/tasks/${id}`, updatedTask);
  return response.data;
};

// DELETE - Remove data (no body needed)
const deleteTask = async (id) => {
  await axios.delete(`/api/tasks/${id}`);
};
```

Each HTTP method has a specific purpose and structure

Step 2 → Promise vs Async/Await

javascript

```javascript
// Promise style (older way)
taskService.getAllTasks()
  .then(response => setTasks(response.data))
  .catch(error => setError(error.message));

// Async/Await style (modern way)
const fetchTasks = async () => {
  try {
    const response = await taskService.getAllTasks();
    setTasks(response.data);
  } catch (error) {
    setError(error.message);
  }
};
```

Async/await is cleaner and easier to read

Step 3 → Proper error handling

```javascript
const handleApiCall = async (apiFunction, successMessage) => {
  try {
    setLoading(true);
    setError(null);
    await apiFunction();
    setSuccessMessage(successMessage);
  } catch (error) {
    if (error.response) {
      // Server responded with error status
      setError(`Server Error: ${error.response.status}`);
    } else if (error.request) {
      // Network error
      setError('Network Error: Unable to reach server');
    } else {
      // Other error
      setError('An unexpected error occurred');
    }
  } finally {
    setLoading(false);
  }
};
```

Handles different types of errors gracefully

Step 4 → Loading states for better UX

```javascript
function TaskManager() {
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(false);
  const [actionLoading, setActionLoading] = useState(false);

  const deleteTaskWithLoading = async (id) => {
    setActionLoading(true);
    try {
      await taskService.deleteTask(id);
      setTasks(tasks.filter(task => task.id !== id));
    } catch (error) {
      setError('Failed to delete task');
    } finally {
      setActionLoading(false);
    }
  };

  return (
    <div>
      {loading && <div>Loading tasks...</div>}
      {actionLoading && <div>Processing...</div>}
      {/* Your task components */}
    </div>
  );
}
```

Shows users what's happening during API calls

✅ **Chapter 5 Complete**: You understand HTTP methods and can handle async operations with proper error handling and loading states.

Chapter 6: React Hooks Deep Dive

🎯 **Objective**: Master useEffect and useContext for API calls and state management

Step 1 → useEffect for API calls

```javascript
```

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

function TaskList() {
  const [tasks, setTasks] = useState([]);

  // Run once when component mounts
  useEffect(() => {
    fetchTasks();
  }, []); // Empty dependency array = run once

  // Run when tasks change
  useEffect(() => {
    console.log('Tasks updated:', tasks);
  }, [tasks]); // Runs when tasks state changes

  // Cleanup effect (optional)
  useEffect(() => {
    const timer = setInterval(() => fetchTasks(), 30000);
    return () => clearInterval(timer); // Cleanup when unmounting
  }, []);

  const fetchTasks = async () => {
    // Your API call here
  };
}
```

useEffect controls when your API calls happen

Step 2 → Create Context for global state

Create `frontend/src/context/TaskContext.js`:

```
javascript
```

```
import React, { createContext, useContext, useReducer } from 'react';

const TaskContext = createContext();

const taskReducer = (state, action) => {
  switch (action.type) {
    case 'SET_TASKS':
      return { ...state, tasks: action.payload };
    case 'ADD_TASK':
      return { ...state, tasks: [...state.tasks, action.payload] };
    case 'DELETE_TASK':
      return {
        ...state,
        tasks: state.tasks.filter(task => task.id !== action.payload)
      };
    case 'UPDATE_TASK':
      return {
        ...state,
        tasks: state.tasks.map(task =>
          task.id === action.payload.id ? action.payload : task
        )
      };
    case 'SET_LOADING':
      return { ...state, loading: action.payload };
    case 'SET_ERROR':
      return { ...state, error: action.payload };
    default:
      return state;
  }
};

export function TaskProvider({ children }) {
  const [state, dispatch] = useReducer(taskReducer, {
    tasks: [],
    loading: false,
    error: null
  });

  return (
    <TaskContext.Provider value={{ state, dispatch }}>
      {children}
    </TaskContext.Provider>
  );
}

export const useTaskContext = () => {
  const context = useContext(TaskContext);
  if (!context) {
    throw new Error('useTaskContext must be used within TaskProvider');
  }
  return context;
};
```

Creates global state that any component can access

Step 3 → Use Context in components

Update `frontend/src/App.js`:

```javascript
import React from 'react';
import { TaskProvider } from './context/TaskContext';
import TaskManager from './components/TaskManager';

function App() {
  return (
    <TaskProvider>
      <div className="App">
        <h1>Task Manager</h1>
        <TaskManager />
      </div>
    </TaskProvider>
  );
}

export default App;
```

Wraps your app with the context provider

Step 4 → Access context in any component

Create `frontend/src/components/TaskManager.js`:

```javascript
```

```
import React, { useEffect } from 'react';
import { useTaskContext } from '../context/TaskContext';
import { taskService } from '../api/taskService';

function TaskManager() {
  const { state, dispatch } = useTaskContext();
  const { tasks, loading, error } = state;

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

  const fetchTasks = async () => {
    dispatch({ type: 'SET_LOADING', payload: true });
    try {
      const response = await taskService.getAllTasks();
      dispatch({ type: 'SET_TASKS', payload: response.data });
    } catch (err) {
      dispatch({ type: 'SET_ERROR', payload: 'Failed to fetch tasks' });
    } finally {
      dispatch({ type: 'SET_LOADING', payload: false });
    }
  };

  const deleteTask = async (id) => {
    try {
      await taskService.deleteTask(id);
      dispatch({ type: 'DELETE_TASK', payload: id });
    } catch (err) {
      dispatch({ type: 'SET_ERROR', payload: 'Failed to delete task' });
    }
  };

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div>
      {tasks.map(task => (
        <div key={task.id}>
          <h3>{task.title}</h3>
          <button onClick={() => deleteTask(task.id)}>Delete</button>
        </div>
        ))}
      </div>
    );
}

export default TaskManager;
```

Any component can now access and modify global task state

✅ **Chapter 6 Complete**: You can manage global state with Context API and control when API calls happen with useEffect.

Chapter 7: React Router & Navigation

🎯 **Objective**: Add navigation between different pages in your React app

Step 1 → Setup Router

Update `frontend/src/App.js`:

```
javascript
```

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import { TaskProvider } from './context/TaskContext';
import Navigation from './components/Navigation';
import TaskList from './components/TaskList';
import CreateTask from './components/CreateTask';
import EditTask from './components/EditTask';

function App() {
  return (
    <TaskProvider>
      <Router>
        <div className="App">
          <Navigation />
          <Routes>
            <Route path="/" element={<TaskList />} />
            <Route path="/create" element={<CreateTask />} />
            <Route path="/edit/:id" element={<EditTask />} />
          </Routes>
        </div>
      </Router>
    </TaskProvider>
  );
}

export default App;
```

Sets up routing for different pages

Step 2 → Create Navigation component

Create `frontend/src/components/Navigation.js`:

```javascript
```

```javascript
import React from 'react';
import { Link, useLocation } from 'react-router-dom';

function Navigation() {
  const location = useLocation();

  return (
    <nav style={{ padding: '20px', borderBottom: '1px solid #ccc' }}>
      <Link
        to="/"
        style={{
          marginRight: '20px',
          fontWeight: location.pathname === '/' ? 'bold' : 'normal'
        }}
      >
        All Tasks
      </Link>
      <Link
        to="/create"
        style={{
          fontWeight: location.pathname === '/create' ? 'bold' : 'normal'
        }}
      >
        Create Task
      </Link>
    </nav>
  );
}

export default Navigation;
```

Creates navigation links between pages

Step 3 → Create task form component

Create `frontend/src/components/CreateTask.js`:

```
javascript
```

```
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { useTaskContext } from '../context/TaskContext';
import { taskService } from '../api/taskService';

function CreateTask() {
  const [title, setTitle] = useState('');
  const [description, setDescription] = useState('');
  const [submitting, setSubmitting] = useState(false);
  const navigate = useNavigate();
  const { dispatch } = useTaskContext();

  const handleSubmit = async (e) => {
    e.preventDefault();
    setSubmitting(true);

    try {
      const newTask = { title, description };
      const response = await taskService.createTask(newTask);
      dispatch({ type: 'ADD_TASK', payload: response.data });
      navigate('/'); // Redirect to task list
    } catch (error) {
      alert('Failed to create task');
    } finally {
      setSubmitting(false);
    }
  };

  return (
    <div style={{ padding: '20px' }}>
      <h2>Create New Task</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Title:</label>
          <input
            type="text"
            value={title}
            onChange={(e) => setTitle(e.target.value)}
            required
            style={{ width: '100%', padding: '8px', margin: '8px 0' }}
          />
        </div>
        <div>
```

```
            <label>Description:</label>
            <textarea
              value={description}
              onChange={(e) => setDescription(e.target.value)}
              style={{ width: '100%', padding: '8px', margin: '8px 0' }}
            />
          </div>
          <button
            type="submit"
            disabled={submitting}
            style={{ padding: '10px 20px' }}
          >
            {submitting ? 'Creating...' : 'Create Task'}
          </button>
        </form>
      </div>
  );
}

export default CreateTask;
```

Form to create new tasks with navigation after submission

Step 4 → Conditional rendering

Update `frontend/src/components/TaskList.js`:

```javascript
```

```
import React, { useEffect } from 'react';
import { Link } from 'react-router-dom';
import { useTaskContext } from '../context/TaskContext';
import { taskService } from '../api/taskService';

function TaskList() {
  const { state, dispatch } = useTaskContext();
  const { tasks, loading, error } = state;

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

  const fetchTasks = async () => {
    dispatch({ type: 'SET_LOADING', payload: true });
    try {
      const response = await taskService.getAllTasks();
      dispatch({ type: 'SET_TASKS', payload: response.data });
    } catch (err) {
      dispatch({ type: 'SET_ERROR', payload: 'Failed to fetch tasks' });
    } finally {
      dispatch({ type: 'SET_LOADING', payload: false });
    }
  };

  if (loading) return <div>Loading tasks...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div style={{ padding: '20px' }}>
      <h2>Task List</h2>
      {tasks.length === 0 ? (
        <div>
          <p>No tasks found.</p>
          <Link to="/create">Create your first task</Link>
        </div>
      ) : (
        tasks.map(task => (
          <div key={task.id} style={{ border: '1px solid #ccc', padding: '10px', margin: '10px 0' }}>
            <h3>{task.title}</h3>
            <p>{task.description}</p>
            <p>Status: {task.completed ? '✅ Completed' : '⏳ Pending'}</p>
            <Link to={`/edit/${task.id}`}>Edit</Link>
```

```
                </div>
            ))
        )}
      </div>
    );
}

export default TaskList;
```

Shows different content based on conditions (no tasks vs has tasks)

✅ **Chapter 7 Complete**: Your app now has navigation between different pages and conditional rendering based on data.

Chapter 8: Complete CRUD Application

🎯 **Objective**: Build a fully functional task management app with Create, Read, Update, Delete operations

Step 1 → Create EditTask component

Create `frontend/src/components/EditTask.js`:

```javascript
```

```
import React, { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';
import { useTaskContext } from '../context/TaskContext';
import { taskService } from '../api/taskService';

function EditTask() {
  const { id } = useParams();
  const navigate = useNavigate();
  const { state, dispatch } = useTaskContext();
  const [title, setTitle] = useState('');
  const [description, setDescription] = useState('');
  const [completed, setCompleted] = useState(false);
  const [loading, setLoading] = useState(true);
  const [submitting, setSubmitting] = useState(false);

  useEffect(() => {
    const task = state.tasks.find(t => t.id === parseInt(id));
    if (task) {
      setTitle(task.title);
      setDescription(task.description);
      setCompleted(task.completed);
      setLoading(false);
    } else {
      // Task not found in state, fetch from API
      fetchTask();
    }
  }, [id, state.tasks]);

  const fetchTask = async () => {
    try {
      const response = await taskService.getAllTasks();
      const task = response.data.find(t => t.id === parseInt(id));
      if (task) {
        setTitle(task.title);
        setDescription(task.description);
        setCompleted(task.completed);
      } else {
        alert('Task not found');
        navigate('/');
      }
    } catch (error) {
      alert('Failed to fetch task');
      navigate('/');
    } finally {
      setLoading(false);
    }
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setSubmitting(true);

    try {
      const updatedTask = { title, description, completed };
      const response = await taskService.updateTask(id, updatedTask);
      dispatch({ type: 'UPDATE_TASK', payload: response.data });
      navigate('/');
    } catch (error) {
      alert('Failed to update task');
    } finally {
      setSubmitting(false);
    }
  };

  if (loading) return <div>Loading task...</div>;

  return (
    <div style={{ padding: '20px' }}>
      <h2>Edit Task</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Title:</label>
          <input
            type="text"
            value={title}
            onChange={(e) => setTitle(e.target.value)}
            required
            style={{ width: '100%', padding: '8px', margin: '8px 0' }}
          />
        </div>
        <div>
          <label>Description:</label>
          <textarea
            value={description}
            onChange={(e) => setDescription(e.target.value)}
            style={{ width: '100%', padding: '8px', margin: '8px 0' }}
          />
          </div>
          <div>
            <label>
              <input
                type="checkbox"
                checked={completed}
                onChange={(e) => setCompleted(e.target.checked)}
              />
              Completed
            </label>
          </div>
          <button
            type="submit"
            disabled={submitting}
            style={{ padding: '10px 20px', marginRight: '10px' }}
          >
            {submitting ? 'Updating...' : 'Update Task'}
          </button>
          <button
            type="button"
            onClick={() => navigate('/')}
            style={{ padding: '10px 20px' }}
          >
            Cancel
          </button>
        </form>
      </div>
  );
}

export default EditTask;
```

Allows editing existing tasks with pre-filled form

Step 2 → Add delete functionality to TaskList

Update `frontend/src/components/TaskList.js`:

```javascript
```

```
import React, { useEffect } from 'react';
import { Link } from 'react-router-dom';
import { useTaskContext } from '../context/TaskContext';
import { taskService } from '../api/taskService';

function TaskList() {
  const { state, dispatch } = useTaskContext();
  const { tasks, loading, error } = state;

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

  const fetchTasks = async () => {
    dispatch({ type: 'SET_LOADING', payload: true });
    try {
      const response = await taskService.getAllTasks();
      dispatch({ type: 'SET_TASKS', payload: response.data });
    } catch (err) {
      dispatch({ type: 'SET_ERROR', payload: 'Failed to fetch tasks' });
    } finally {
      dispatch({ type: 'SET_LOADING', payload: false });
    }
  };

  const deleteTask = async (id, title) => {
    if (window.confirm(`Are you sure you want to delete "${title}"?`)) {
      try {
        await taskService.deleteTask(id);
        dispatch({ type: 'DELETE_TASK', payload: id });
      } catch (error) {
        alert('Failed to delete task');
      }
    }
  };

  const toggleComplete = async (task) => {
    try {
      const updatedTask = { ...task, completed: !task.completed };
      const response = await taskService.updateTask(task.id, updatedTask);
      dispatch({ type: 'UPDATE_TASK', payload: response.data });
    } catch (error) {
      alert('Failed to update task');
    }
  };

  if (loading) return <div>Loading tasks...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <div style={{ padding: '20px' }}>
      <h2>Task List ({tasks.length} tasks)</h2>
      {tasks.length === 0 ? (
        <div>
          <p>No tasks found.</p>
          <Link to="/create" style={{ padding: '10px 20px', backgroundColor: '#007bff', color: 'white', textDecor
            Create your first task
          </Link>
        </div>
      ) : (
        <div>
          {tasks.map(task => (
            <div
              key={task.id}
              style={{
                border: '1px solid #ccc',
                padding: '15px',
                margin: '10px 0',
                backgroundColor: task.completed ? '#f8f9fa' : 'white'
              }}
            >
              <h3 style={{ textDecoration: task.completed ? 'line-through' : 'none' }}>
                {task.title}
              </h3>
              <p style={{ color: task.completed ? '#6c757d' : 'black' }}>
                {task.description}
              </p>
              <p>
                Status: {task.completed ? '✅ Completed' : '⏳ Pending'}
              </p>
              <div style={{ marginTop: '10px' }}>
                <button
                  onClick={() => toggleComplete(task)}
                  style={{
                    padding: '5px 10px',
                    marginRight: '10px',
                    backgroundColor: task.completed ? '#28a745' : '#ffc107',
                color: 'white',
                border: 'none'
              }}
            >
              {task.completed ? 'Mark Incomplete' : 'Mark Complete'}
            </button>
            <Link
              to={`/edit/${task.id}`}
              style={{
                padding: '5px 10px',
                marginRight: '10px',
                backgroundColor: '#007bff',
                color: 'white',
                textDecoration: 'none'
              }}
            >
              Edit
            </Link>
            <button
              onClick={() => deleteTask(task.id, task.title)}
              style={{
                padding: '5px 10px',
                backgroundColor: '#dc3545',
                color: 'white',
                border: 'none'
              }}
            >
              Delete
            </button>
          </div>
        </div>
      ))}
      <div style={{ marginTop: '20px' }}>
        <Link
          to="/create"
          style={{
            padding: '10px 20px',
            backgroundColor: '#28a745',
            color: 'white',
            textDecoration: 'none'
          }}
        >
          Add New Task
        </Link>
```

```
                </div>
              </div>
            )}
        </div>
    );
}

export default TaskList;
```

Complete task list with all CRUD operations and better styling

Step 3 → Create database setup script

Create `backend/setup.sql`:

```sql
CREATE DATABASE IF NOT EXISTS taskdb;
USE taskdb;

-- Test data (optional)
INSERT INTO task (title, description, completed) VALUES
('Learn Spring Boot', 'Complete the Spring Boot tutorial', false),
('Build React App', 'Create a React frontend application', false),
('Connect Frontend to Backend', 'Integrate React with Spring Boot API', false);
```

Sets up your MySQL database with test data

Step 4 → Add error boundary

Create `frontend/src/components/ErrorBoundary.js`:

```javascript
```

```
import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.error('Error caught by boundary:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return (
        <div style={{ padding: '20px', textAlign: 'center' }}>
          <h2>Something went wrong.</h2>
          <button onClick={() => window.location.reload()}>
            Reload Page
          </button>
        </div>
      );
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

Catches and handles React component errors gracefully

Step 5 → Final App.js with error boundary

Update `frontend/src/App.js`:

```javascript
```

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import { TaskProvider } from './context/TaskContext';
import ErrorBoundary from './components/ErrorBoundary';
import Navigation from './components/Navigation';
import TaskList from './components/TaskList';
import CreateTask from './components/CreateTask';
import EditTask from './components/EditTask';
import './App.css';

function App() {
  return (
    <ErrorBoundary>
      <TaskProvider>
        <Router>
          <div className="App">
            <header style={{ backgroundColor: '#343a40', color: 'white', padding: '20px' }}>
              <h1>Task Manager App</h1>
            </header>
            <Navigation />
            <main>
              <Routes>
                <Route path="/" element={<TaskList />} />
                <Route path="/create" element={<CreateTask />} />
                <Route path="/edit/:id" element={<EditTask />} />
              </Routes>
            </main>
          </div>
        </Router>
      </TaskProvider>
    </ErrorBoundary>
  );
}

export default App;
```

Complete app structure with error handling

✅ **Chapter 8 Complete**: You now have a fully functional CRUD task management application!

Chapter 9: Advanced Concepts

🎯 **Objective**: Learn important concepts for production-ready applications

Security Best Practices

```javascript
// Input validation in React
const validateTask = (task) => {
  const errors = {};
  if (!task.title || task.title.trim().length < 3) {
    errors.title = 'Title must be at least 3 characters';
  }
  if (task.title && task.title.length > 100) {
    errors.title = 'Title must be less than 100 characters';
  }
  return errors;
};

// Sanitize user input
const sanitizeInput = (input) => {
  return input.trim().replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi, '');
};
```

Always validate and sanitize user input

Error Handling & Loading States

```javascript
```

```
// Centralized error handling
const useApiCall = () => {
  const [state, setState] = useState({
    data: null,
    loading: false,
    error: null
  });

  const execute = async (apiCall) => {
    setState(prev => ({ ...prev, loading: true, error: null }));
    try {
      const result = await apiCall();
      setState({ data: result, loading: false, error: null });
      return result;
    } catch (error) {
      setState({ data: null, loading: false, error: error.message });
      throw error;
    }
  };

  return { ...state, execute };
};
```

Reusable pattern for API calls with consistent error handling

Caching Strategy

```
javascript
```

```javascript
// Simple in-memory cache
const cache = new Map();

const cachedApiCall = async (key, apiCall, ttl = 300000) => { // 5 minutes
  const cached = cache.get(key);
  if (cached && Date.now() - cached.timestamp < ttl) {
    return cached.data;
  }

  const data = await apiCall();
  cache.set(key, { data, timestamp: Date.now() });
  return data;
};

// Usage
const fetchTasksWithCache = () => cachedApiCall('tasks', taskService.getAllTasks);
```

Reduces API calls and improves performance

API Versioning

```javascript
// Version your API endpoints
const API_BASE = 'http://localhost:8080/api/v1';

const taskService = {
  getAllTasks: () => axios.get(`${API_BASE}/tasks`),
  // If you need to support older versions
  getAllTasksV2: () => axios.get(`${API_BASE}/v2/tasks`)
};
```

Plan for future API changes

✅ **Chapter 9 Complete**: You understand important concepts for building production-ready applications.

🎉 Final Steps & Testing

Step 1 → Start your applications

```bash
```

```
# Terminal 1 - Start MySQL (make sure it's running)
mysql -u root -p
CREATE DATABASE taskdb;

# Terminal 2 - Start Spring Boot
cd backend
./mvnw spring-boot:run

# Terminal 3 - Start React
cd frontend
npm start
```

Gets your full-stack app running

Step 2 → Test all functionality

1. ✅ View tasks at http://localhost:3000
2. ✅ Create a new task using the form
3. ✅ Edit an existing task
4. ✅ Toggle task completion status
5. ✅ Delete a task with confirmation
6. ✅ Navigate between pages

Step 3 → Common troubleshooting

```bash
# If Spring Boot fails to start:
# Check MySQL is running and credentials are correct in application.properties

# If React can't connect to API:
# Verify CORS configuration in Spring Boot
# Check API is running on http://localhost:8080

# If database connection fails:
# Make sure MySQL is running
# Verify database name, username, password in application.properties
```

🏆 Congratulations!

You've built a complete full-stack application with:

- ✅ Java Spring Boot REST API
- ✅ MySQL database integration
- ✅ React frontend with modern hooks
- ✅ Global state management with Context API
- ✅ Navigation with React Router
- ✅ Complete CRUD operations
- ✅ Error handling and loading states
- ✅ CORS configuration
- ✅ Proper project structure

What you've learned:

- Client-server architecture
- REST API design and HTTP methods
- Database connections and JPA
- React hooks and state management
- Async/await and Promise handling
- Context API for global state
- React Router for navigation
- Error boundaries and error handling
- Security best practices

Next steps:

- Add authentication with JWT tokens
- Implement pagination for large datasets
- Add real-time updates with WebSockets
- Deploy your app to cloud platforms
- Add automated testing
- Implement advanced filtering and search

You now have a solid foundation in full-stack development! 🚀