Low Level Design

Project Name: Weather App

Introduction

What is Low-Level design document?

The goal of LLD or a low-level design document (LLDD) is to give the internal logical design of the

actual program code for Food Recommendation System. LLD describes the class diagrams with the

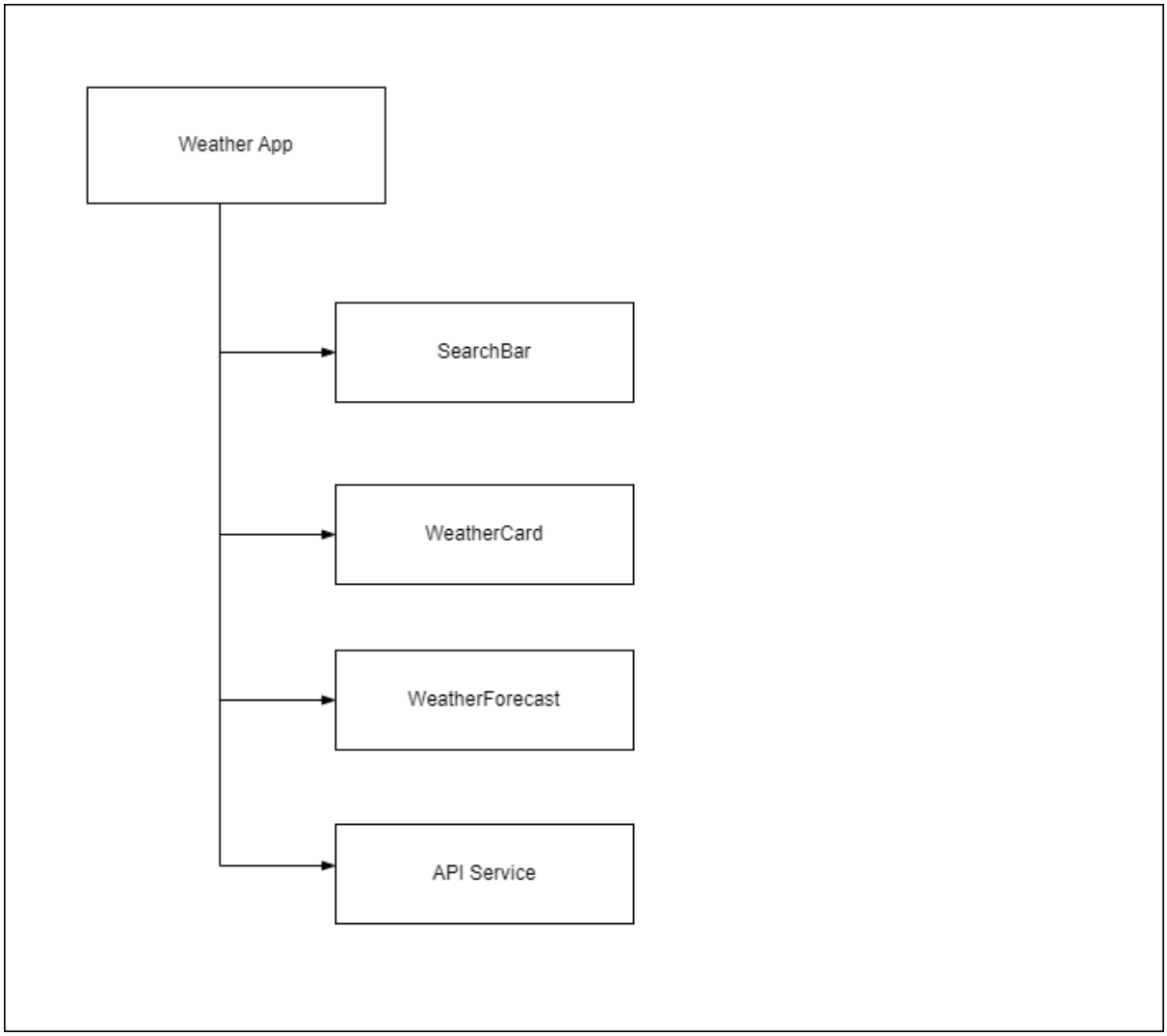
methods and relations between classes and program specs. It describes the modules so that the

programmer can directly code the program from the document.

Scope

Low-level design (LLD) is a component-level design process that follows a step-by-

step refinement process. This process can be used for designing data structures, required software architecture, source code and ultimately, performance algorithms. Overall, the data organization may be defined during requirement analysis and then refined during data design work


Weather App
SearchBar
WeatherCard
WeatherForecast
API Service

In this simplified diagram, the Weather App component represents the main component that acts as the entry point for the application. It connects and manages the other components in the architecture.

The SearchBar component allows users to search for a specific location's weather information.

The WeatherCard component is responsible for displaying the current weather information for the selected location.

The WeatherForecast component displays the weather forecast for upcoming days.

The API Service component handles the communication with the weather API, fetching data for the requested location.

Please note that this is a high-level representation and there may be additional components and interactions based on your specific app requirements. It's important to adapt and extend this diagram to reflect the specific architecture and relationships of your React weather app.

Architecture Description

The architecture of a weather app typically follows a client-server model, with the client-side implemented using React.js and the server-side involving an external weather API. Here is a description of the key components and their interactions within the weather app architecture:

1. Client-Side (React.js):

React Components: The app consists of various React components that handle the presentation layer and user interface.

State Management: React's built-in state or external state management libraries (e.g., Redux, Context API) are used to manage and share data between components.

UI Components: These components represent different elements of the user interface, such as search bar, weather card, and forecast display.

API Integration: An API service component fetches weather data from the server-side API and updates the app's state with the received information.

Routing: React Router or similar libraries handle client-side routing, enabling navigation between different screens or components within the app.

Error Handling: The app incorporates error handling mechanisms to capture, log, and display errors to the user in a meaningful way.

2. Server-Side (External Weather API):

Weather API: The app integrates with an external weather API (e.g., OpenWeatherMap, Weatherbit) to fetch weather data for different locations.

API Key: The app authenticates with the weather API using an API key obtained from the API provider.

API Requests: The client-side app makes HTTP requests to the weather API's endpoints, passing necessary parameters such as the location.

Response Parsing: The app parses the JSON or XML responses received from the weather API to extract relevant weather information.

Data Transfer: The weather data is transferred from the server-side API to the client-side app in response to the client's requests.

3. Data Flow:

User Interaction: Users input a location or interact with the app's UI components.

API Requests: The app sends requests to the weather API, passing the location or other parameters.

API Response: The weather API processes the requests and returns weather data as a response.

Data Parsing: The app parses and transforms the received data into a usable format for the React components.

Component Rendering: The React components receive the weather data and render the information on the user interface.

State Update: The app updates its state with the received weather data, triggering re-rendering of the affected components.

User Experience: Users view and interact with the weather information displayed on the app's UI components.

The weather app architecture combines the client-side capabilities of React.js with the server-side weather API to provide users with up-to-date weather information for their desired locations. This separation of concerns ensures a scalable and maintainable application, where the React components handle the UI and user interactions, while the weather API focuses on fetching and providing weather data.

Unit Test cases

Test case for fetching weather data:

Mock the API service function used to fetch weather data.

Assert that the function is called with the correct parameters, such as the location.

Verify that the function returns the expected weather data.

Test case for displaying weather information:

Render the WeatherCard component with sample weather data as props.

Assert that the component correctly displays the temperature, weather conditions, and icons.

Verify that the component handles different weather conditions appropriately (e.g., displaying a sunny icon for clear weather).

Test case for handling search functionality:

Simulate a user entering a location in the SearchBar component.

Verify that the app triggers the search functionality correctly.

Assert that the app makes the expected API request with the entered location.

Test case for error handling:

Mock an error response from the API service ensure that the app displays an error message or fallback UI when an error occurs verifies that the app handles different error scenarios, such as network errors or invalid API responses.

Test case for responsive design:

Use a testing library to render the app on different screen sizes.

Verify that the UI elements and layout adjust correctly to different screen resolutions.

Assert that the app provides a consistent and responsive user experience across different devices.

Test case for data caching:

Mock multiple API requests for the same location.

Verifies that subsequent requests are served from the cache instead of making additional API calls assert that the app correctly manages the cache and updates the data when necessary.

Test case for forecast display:

Render the WeatherForecast component with sample forecast data as props verifies that the component correctly displays the forecasted weather information, including temperature, weather conditions, and dates.

Asserts that the component handles different forecast scenarios, such as displaying multiple days of forecast data.

Test case for user interactions:

Simulate user interactions, such as clicking on a location in the auto-suggestion dropdown or navigating between screens.

Verifies that the app responds appropriately to user actions, such as triggering a search or updating the UI based on the selected location.

Test case for data parsing and transformation:

Test the functions responsible for parsing and transforming the weather data received from the API assert that the data is correctly processed and formatted for display in the app's UI components.