****

**COMSATS University Islamabad (CUI) Attock Campus**

Software Requirement Specification  
(SRS DOCUMENT)

for

**Wheat Shield**

Version 1.0

***By***

**Kashif Hussain CIIT/FA20-BCS-019/ATK**

**Muhammad Zubair CIIT/FA20-BCS-041/ATK**

***Supervisor*Mr. Muhammad Wasim Khan**

*Bachelor of Science in Computer Science (2020-2024)*

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason for Changes** | **Version** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Application Evaluation History

|  |  |
| --- | --- |
| **Comments (by committee)**  **\*include the ones given at scope time both in doc and presentation** | **Action Taken** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

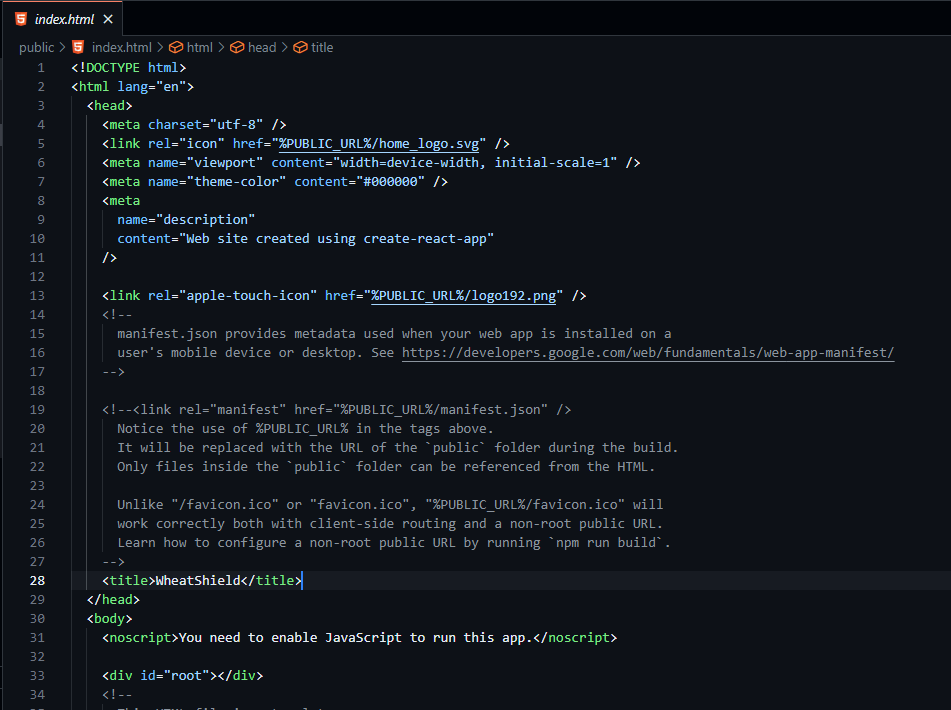
Supervised by

Mr. Muhammad Wasim Khan

Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_

React Js Code Explain

**Code:**



**Summary:**

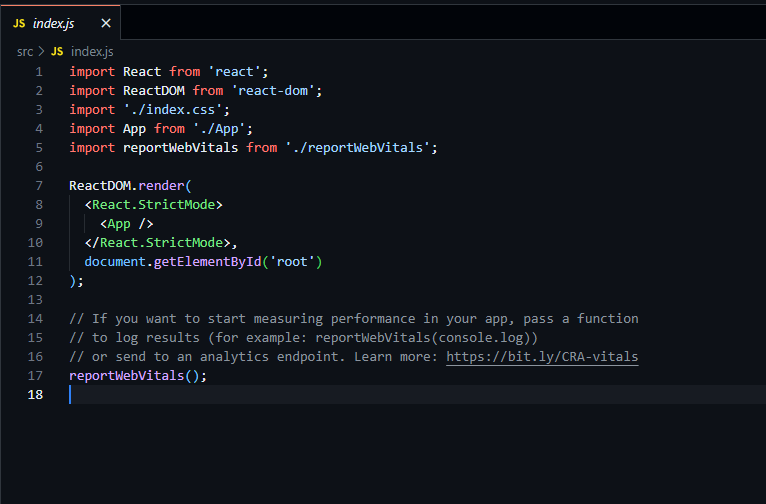
This HTML template serves as the entry point for a React.js web application. It contains essential metadata and links to resources necessary for the application to function correctly. Notable points include:

* **Meta Tags:** Standard meta tags are included for character set, viewport configuration, theme color, and description.
* **Favicon:** The favicon, which appears in the browser tab, is set to a specified SVG file.
* **Manifest.json:** A commented out link to the manifest file, used for configuring the web app's metadata during installation on mobile devices or desktops.
* **Title:** The title of the web application is set to "WheatShield."
* **JavaScript Requirement:** A message for users without JavaScript enabled, indicating that JavaScript must be enabled to run the app.
* **Root Element:** The root **div** element with the ID "root" serves as the mounting point for the React components rendered by the application.

This HTML file acts as the foundation upon which the React.js application is built, providing essential structure and configurations for the web application.

Top of Form

**Code:**



**Summary:**

This JavaScript code is the entry point of a React.js application, where it renders the main application component (**<App />**) into the HTML document. Key points include:

* **Import Statements:** React and ReactDOM libraries are imported, along with the application's styles, main component (**App**), and the **reportWebVitals** function.
* **Rendering the App:** The **ReactDOM.render()** function is used to render the **<App />** component. The **React.StrictMode** wrapper is applied for enhanced error checking and potential future compatibility.
* **Mounting Point:** The rendered component is mounted onto the HTML element with the ID "root," as specified in the **document.getElementById('root')** call.
* **Performance Measurement:** The **reportWebVitals()** function can be used to measure the application's performance. It's currently called, but the results are not explicitly logged.

This code initializes the React.js application, rendering the main component into the specified HTML element and setting the stage for the application's functionality and user interface.

**Code:**



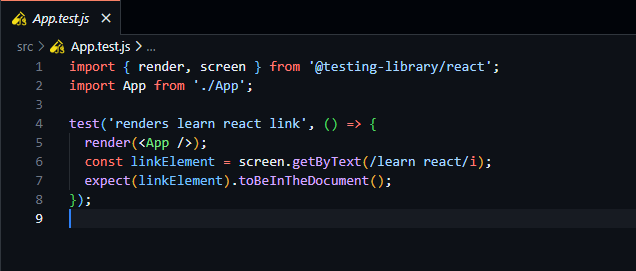
**Summary:**

This code exports a function named **reportWebVitals** intended for measuring various web performance metrics. Here are the key points:

* **Function Signature:** The function takes a callback function **onPerfEntry** as its parameter. This function is expected to handle the received performance entries.
* **Dynamic Import:** The **import('web-vitals')** statement dynamically imports the **web-vitals** library, which provides methods to measure Core Web Vitals. This dynamic import ensures that the library is only loaded when needed, reducing initial bundle size.
* **Core Web Vitals:** Within the dynamic import block, the **getCLS**, **getFID**, **getFCP**, **getLCP**, and **getTTFB** functions from the **web-vitals** library are extracted. These functions measure Cumulative Layout Shift (CLS), First Input Delay (FID), First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Time to First Byte (TTFB) respectively, which are important metrics for web performance.
* **Measurement and Callback:** Each of the Core Web Vitals functions is called with the **onPerfEntry** callback, allowing the measured values to be passed to the provided callback function for further processing or logging.
* **Export:** The **reportWebVitals** function is exported, making it available for use in other parts of the application. This export enables the integration of performance monitoring and analysis into the React.js application.

This code snippet sets up the infrastructure for capturing Core Web Vitals data, enabling developers to measure and analyze the performance of the web application.

**Code:**



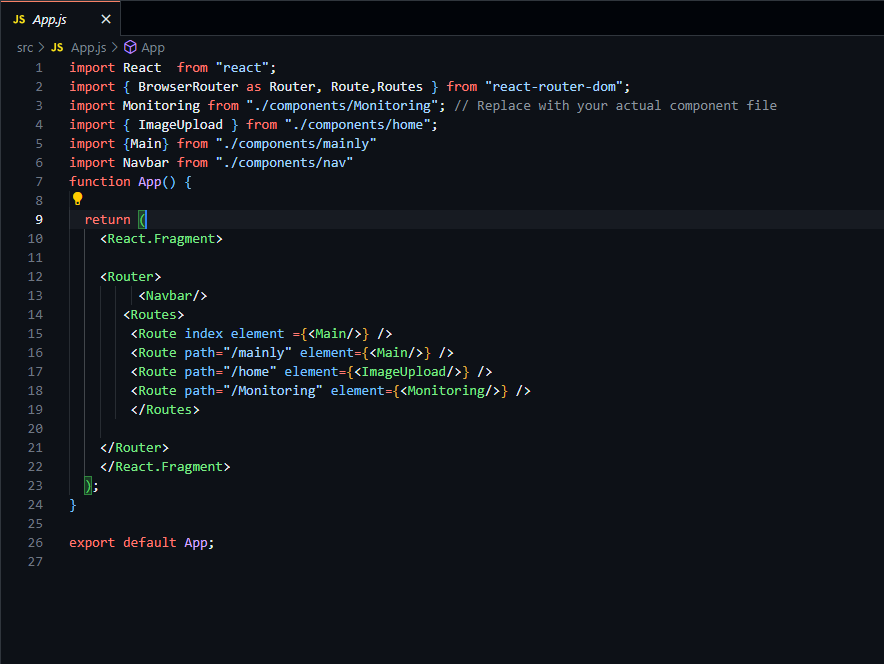
**Summary:**

This code snippet represents a unit test for a React component using the **@testing-library/react** library. Here's a breakdown of the key points:

* **Testing Library Functions:** The **render** function from **@testing-library/react** is used to render the **App** component in a virtual DOM environment. This allows testing the component in isolation.
* **Screen Queries:** The **screen.getByText** function is used to query elements in the rendered component. In this case, it searches for an element containing the text "learn react" (case-insensitive) and returns the matching element.
* **Expectation:** The **expect** statement asserts that the **linkElement** (the element containing "learn react") is present in the rendered output. If the element is found, it means that the text "learn react" is present in the **App** component, satisfying the test condition.
* **Test Description:** The **test** function wraps the test logic, providing a description ("renders learn react link") for the test case.

This test verifies whether the "learn react" text is rendered within the **App** component. If the text is found, the test passes; otherwise, it fails, indicating a problem with the rendering logic. This type of test ensures that the component renders expected content, helping maintain consistent UI behavior across changes.

**Code:**



**Summary:**

This code snippet defines the main structure of a React application using React Router for navigation. Here's a breakdown of the key points:

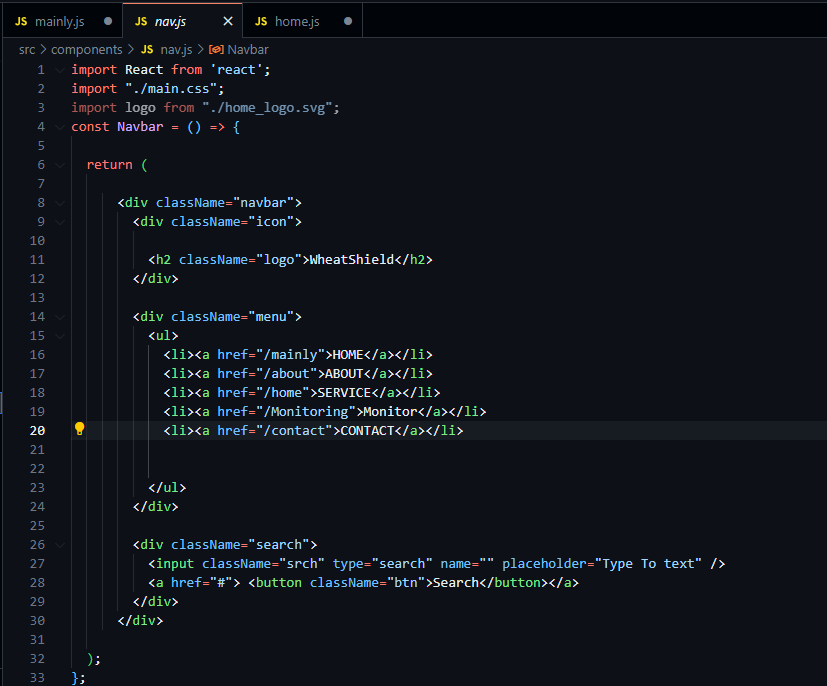
* **Import Statements:** The code imports necessary modules from React and other components/modules required for the application, including **BrowserRouter**, **Route**, **Routes**, and various components (**Monitoring**, **ImageUpload**, **Main**, **Navbar**).
* **Functional Component:** The **App** function is defined as a functional component. It returns the JSX elements that compose the application's UI.
* **React Router Setup:** Inside the **App** component, a **Router** component is used from React Router to enable client-side routing. The **Router** component wraps the entire application.
* **Routes Configuration:** The **Routes** component is used to define different routes within the application. Each **Route** element specifies a path and the corresponding component (**Main**, **ImageUpload**, **Monitoring**) that should be rendered when the URL matches the specified path.
* **Navbar Component:** The **Navbar** component is rendered outside the **Routes** component, indicating that it appears on every page of the application.
* **Export:** The **App** component is exported as the default export of the module, making it accessible for use in other parts of the application.

**Summary of Functionalities:**

* The application uses React Router for client-side navigation.
* It defines three routes: "/mainly", "/home", and "/Monitoring", each associated with a specific component (**Main**, **ImageUpload**, **Monitoring**).
* The **Navbar** component is displayed on all pages, providing consistent navigation.
* The application structure allows users to navigate between different sections of the app using the specified routes.

Top of Form

Code:



**Summary:**

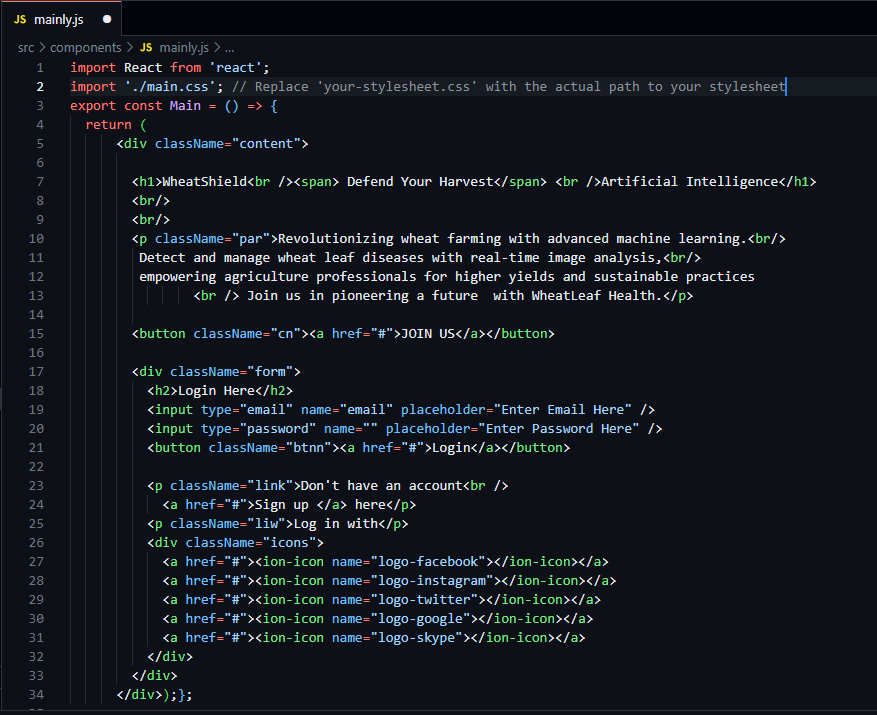
The **Navbar** component provides the application's navigation bar with links and a search functionality. Here's a summary of the code:

* **Imports:** The component imports React and a CSS file (**main.css**) for styling. It also imports an image (**home\_logo.svg**) to be used as the logo.
* **Logo:** The component displays the application logo (**WheatShield**) using an **h2** element. The logo is styled using CSS classes.
* **Navigation Links:** The component includes a list of navigation links (**HOME**, **ABOUT**, **SERVICE**, **Monitor**, and **CONTACT**) inside an unordered list (**ul**). Each link is an **a** element pointing to specific routes (**/mainly**, **/about**, **/home**, **/Monitoring**, and **/contact**).
* **Search Bar:** The component features a search bar with an input field and a "Search" button. Users can enter text into the search field and click the button to initiate a search. The search functionality is not implemented in the provided code snippet.
* **Styling:** The component applies styles using the CSS classes defined in the **main.css** file. It uses classes to style the logo, navigation links, search bar, and button.

**Functionalities:**

* Display the application logo (**WheatShield**) in the navigation bar.
* Provide navigation links for different sections of the application, allowing users to navigate to specific routes.
* Include a search bar, though the search functionality is not implemented in this code snippet.
* The component offers a clean and straightforward user interface for navigation and potential search actions.

Code;



**Summary:**

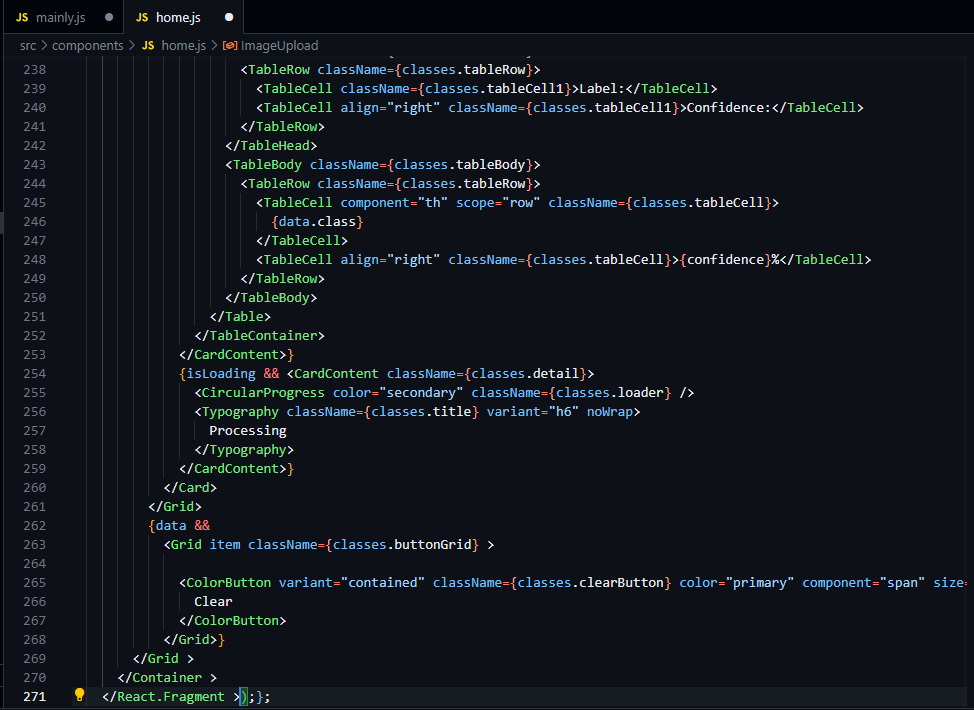
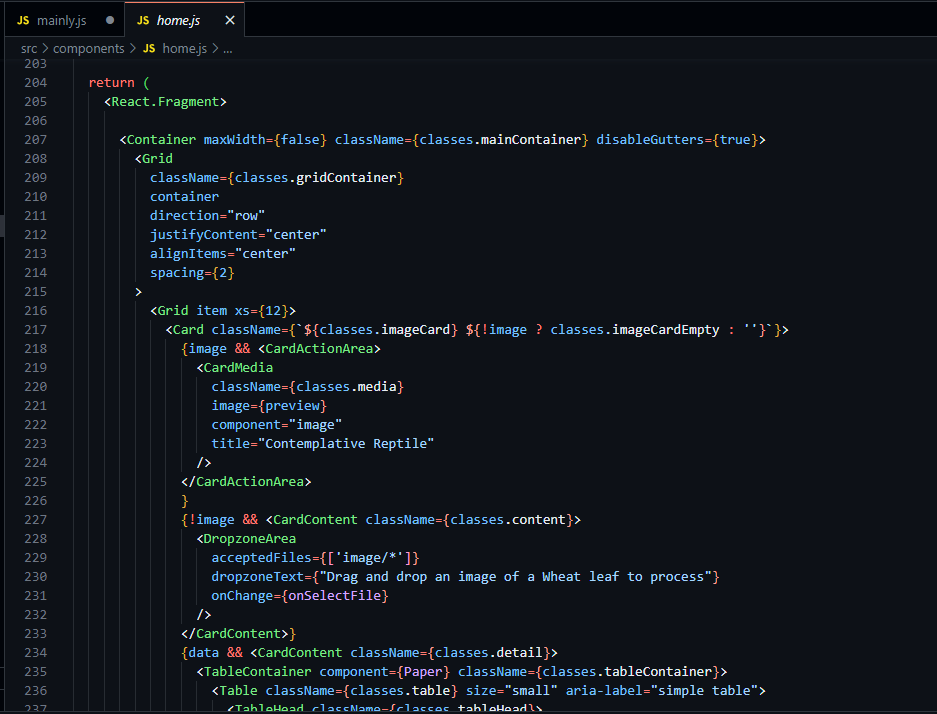
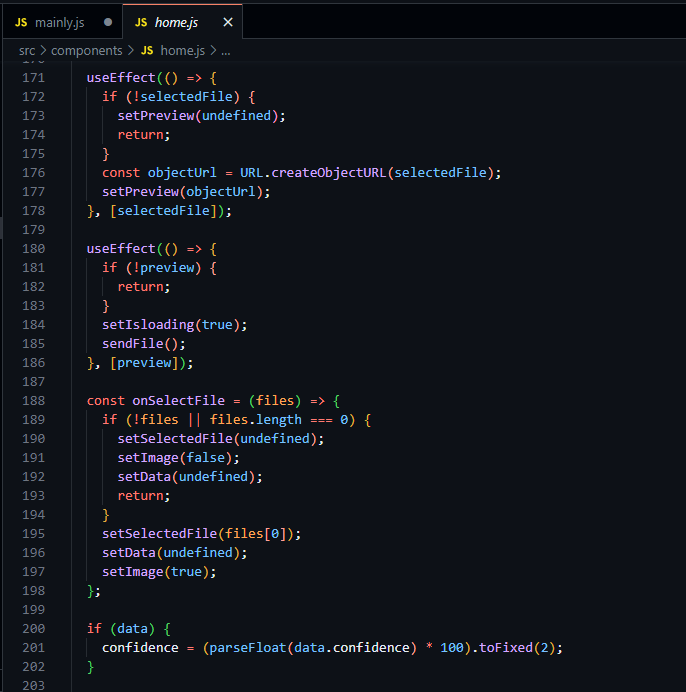
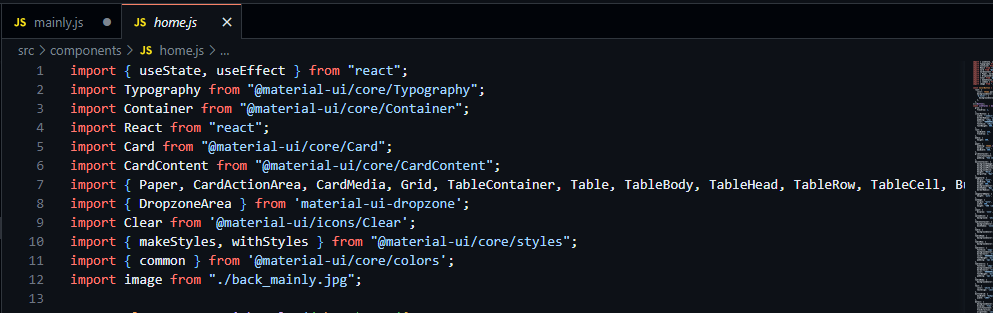
This code snippet defines a functional component named **Main**. The component represents the main content of the application's landing page. Here's a summary of the code:

* **Import Statement:** The code imports React and a CSS file named **main.css** for styling.
* **Functional Component:** The **Main** component is defined as a functional component. It returns JSX elements representing the content of the landing page.
* **HTML Structure:** Within the component, there is a **<div>** element with the class name "content". Inside this **<div>**, there are several elements including:
  + An **<h1>** element displaying the title "WheatShield - Defend Your Harvest - Artificial Intelligence".
  + A paragraph (**<p>**) with class name "par" describing the application's features related to wheat farming and disease detection.
  + A "JOIN US" button with a link.
  + A login form with email and password input fields, a "Login" button, and links for account registration.
  + Social media icons for Facebook, Instagram, Twitter, Google, and Skype.
* **Styling:** The styles for the component are defined in an external CSS file (**main.css**), which is imported at the beginning of the component file. The CSS classes mentioned in the JSX elements are used for applying specific styles to the respective elements.

**Summary of Functionalities:**

* The **Main** component represents the landing page of the application.
* It includes a prominent title, description, and a call-to-action button encouraging users to join the platform.
* There's a login form with input fields for email and password, along with a login button and registration links.
* Social media icons are provided as links, allowing users to connect through various platforms.
* The component is styled using an external CSS file for a visually appealing layout.

Top of Form

**Code:**

**Summary:**

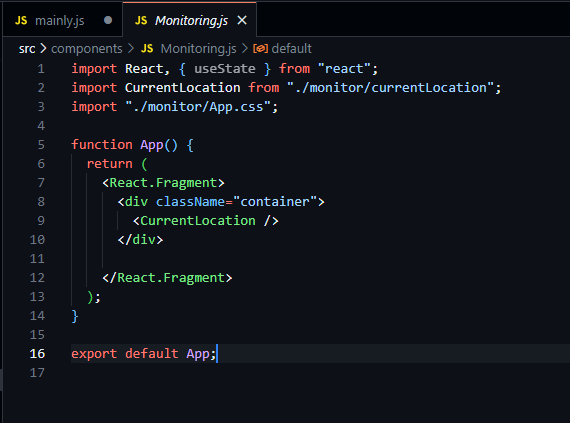
This React component, named **ImageUpload**, represents the image upload and processing functionality of the application. Here's a summary of the code:

* **Imports:** The component imports necessary dependencies from Material-UI and other libraries, including React, Material-UI components, and icons.
* **State Management:** The component uses the **useState** hook to manage various states, including **selectedFile** (the uploaded image file), **preview** (URL of the preview image), **data** (result data from the API), **image** (boolean indicating whether an image is uploaded), and **isLoading** (boolean indicating loading state).
* **File Upload:** The component includes a file upload section allowing users to upload an image of a wheat leaf. It uses the **DropzoneArea** component from Material-UI for handling file drops. When an image is selected, it triggers the **onSelectFile** function, which sets the **selectedFile** state, displays a preview, and initiates the image processing.
* **Image Processing:** The component sends the uploaded image to the server API (**process.env.REACT\_APP\_API\_URL**) using an Axios POST request for processing. It updates the **data** state with the API response, representing the detected class and confidence level.
* **Result Display:** If a result is received (**data** is not null), the component displays the detected class label and confidence level in a table format. The confidence level is calculated as a percentage and displayed with two decimal places.
* **Loading State:** While the image is being processed, a loading spinner (**CircularProgress**) is displayed along with a "Processing" text.
* **Clear Functionality:** After processing, a "Clear" button is displayed. When clicked, it resets the component's state, allowing users to upload a new image for processing.
* **Styling:** The component applies styles using Material-UI's styling system (**makeStyles** and **withStyles**). It uses classes to style various components such as the card, media, table, buttons, and loading spinner.
* **Responsive Design:** The component is designed to be responsive and adapt to different screen sizes.

**Functionalities:**

* Users can upload an image of a wheat leaf for processing.
* The component displays the uploaded image and processes it using the specified API endpoint.
* Detected class label and confidence level are displayed in a table format.
* Users can clear the uploaded image and results to upload a new image for processing.
* The component provides a visually appealing and responsive user interface for image upload and processing.

**Code:**



**Summary:**

The **App** component serves as the entry point of the monitoring feature in the application. Here's a summary of the code:

* **Imports:** The component imports React, the **useState** hook, and the **CurrentLocation** component. It also imports a CSS file (**App.css**) for styling.
* **Component Structure:** The **App** component is a functional component. Inside the component, there's a **div** element with the class name **container**.
* **Child Component:** The **CurrentLocation** component is rendered inside the **div** element. This component likely contains the logic and UI elements related to displaying the current location or monitoring functionality. The specifics of the **CurrentLocation** component's behavior are not provided in this code snippet.
* **Styling:** The component applies styles using the CSS classes defined in the **App.css** file. However, the specific styles and their definitions are not included in the provided code snippet.

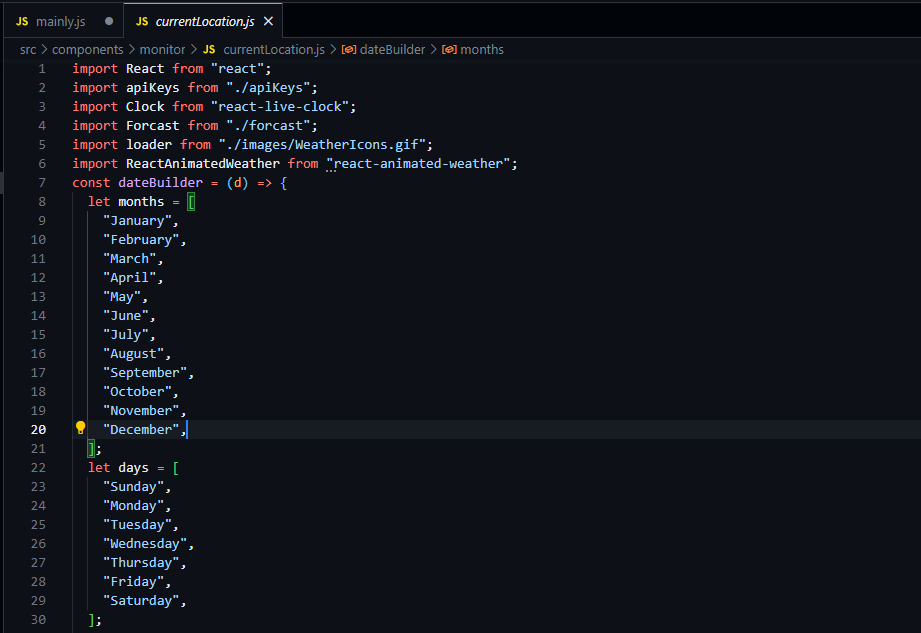
**Functionalities:**

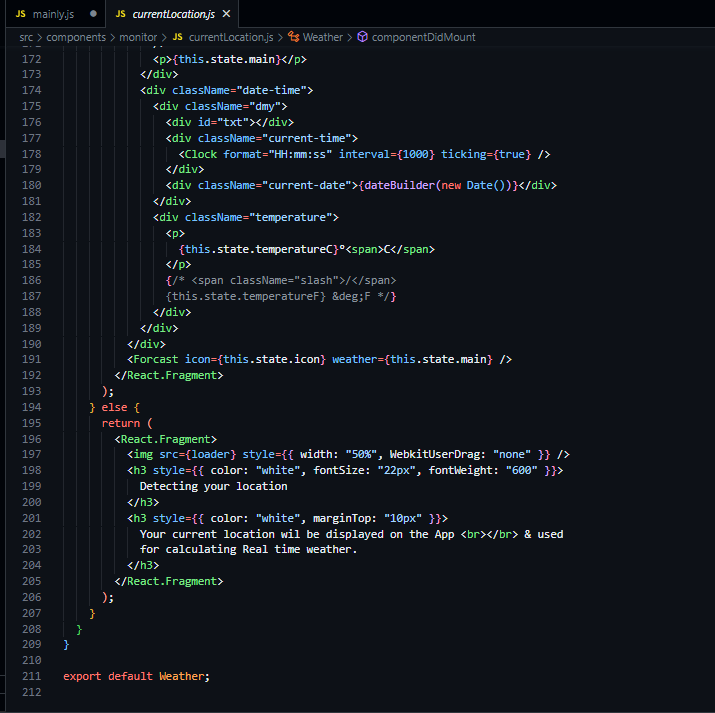
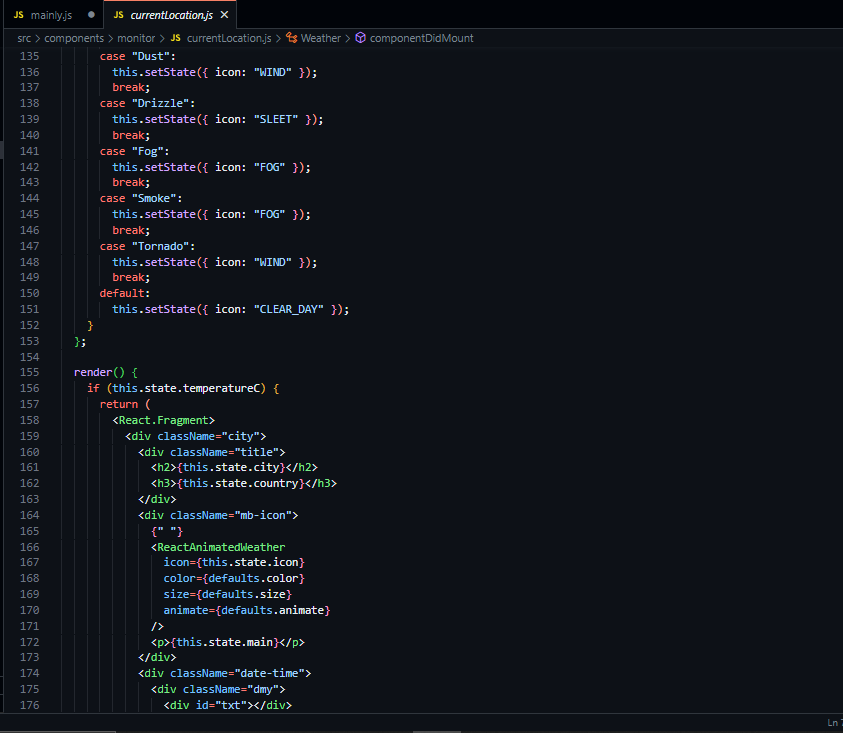
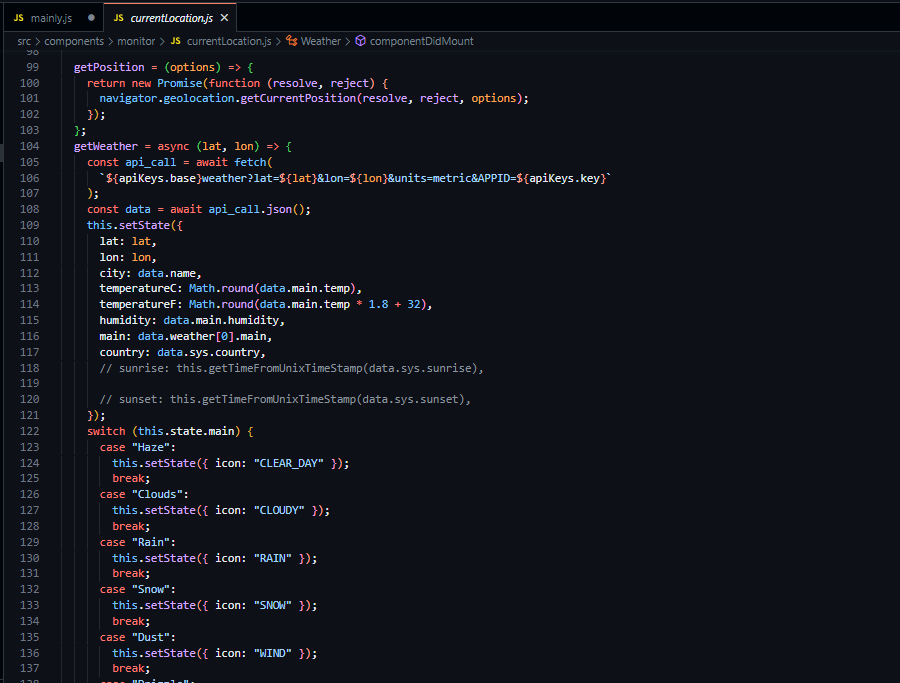
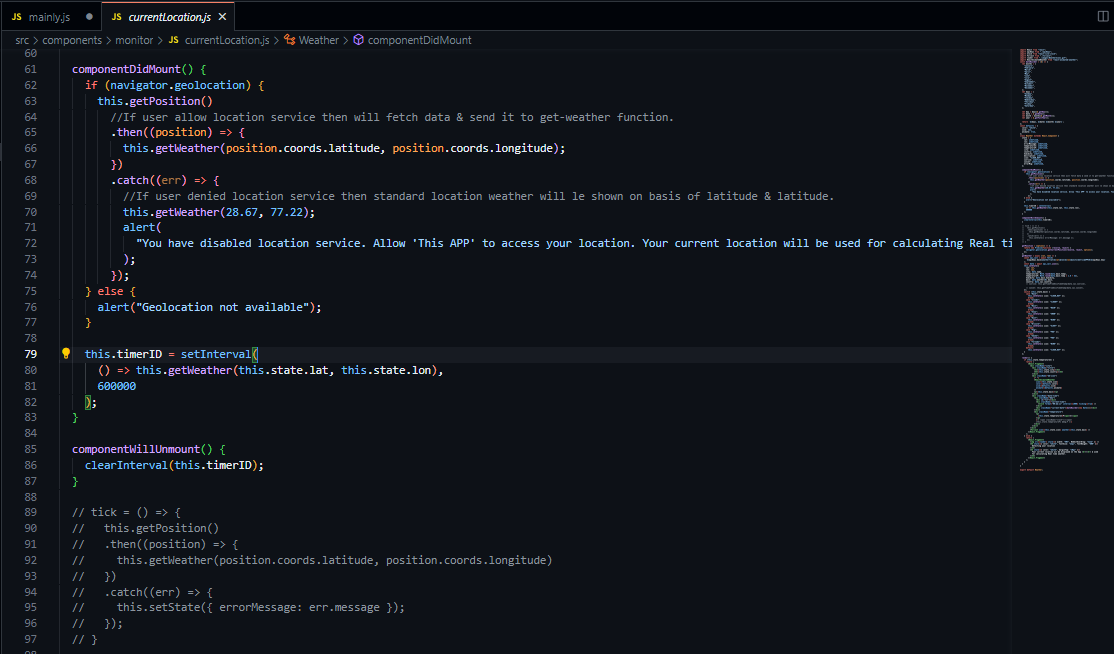
* Render the **CurrentLocation** component, which presumably handles displaying the current location or monitoring information.
* The component doesn't utilize local state or provide any additional functionality beyond rendering the **CurrentLocation** component.

**Note:** The provided code snippet is concise and renders the **CurrentLocation** component within a container. For a detailed understanding of the monitoring functionality, the implementation details within the **CurrentLocation** component would need to be examined.

Top of Form

**Code:**





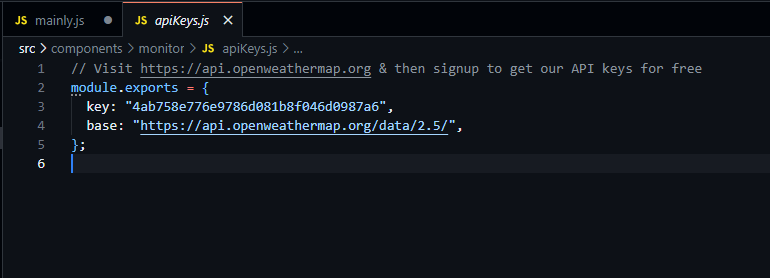
**Summary:**

The **Weather** component is a class-based React component responsible for fetching and displaying weather data. Here's a summary of the code:

* **Imports:** The component imports React, various dependencies, and configurations such as **apiKeys** for API access, **Clock** for real-time clock, **Forcast** component, and weather icons.
* **State:** The component manages various states, including latitude (**lat**), longitude (**lon**), weather details such as temperature in Celsius and Fahrenheit, city, country, humidity, weather description, weather icon, sunrise, sunset, and error messages.
* **Component Lifecycle:** The component makes use of the **componentDidMount** lifecycle method to fetch weather data using geolocation. It sets up an interval to update weather data every 10 minutes (**600000** milliseconds).
* **Methods:**
  + **getPosition**: Utilizes the Geolocation API to get the user's current position.
  + **getWeather**: Fetches weather data from a weather API using the provided latitude and longitude. It converts weather conditions (**main**) to corresponding weather icons.
  + **dateBuilder**: Takes a **Date** object as input and returns a formatted date string.
* **Rendering:**
  + If weather data is available, it displays the city name, country code, weather icon, current time, date, and temperature in Celsius. It also includes a forecast component.
  + If weather data is not available (initial loading or geolocation denied), it displays a loading spinner and messages indicating that the location is being detected.

**Note:** The component uses geolocation to fetch weather data based on the user's location. It provides real-time weather updates and uses appropriate weather icons based on the weather conditions. The actual API endpoints and the **Forcast** component's implementation details are not provided in this code snippet. Additional styling and error handling might be required for a production-ready application.

**Code:**



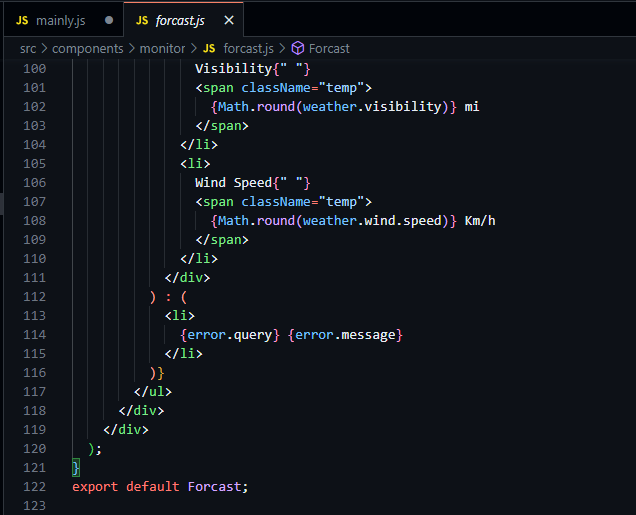
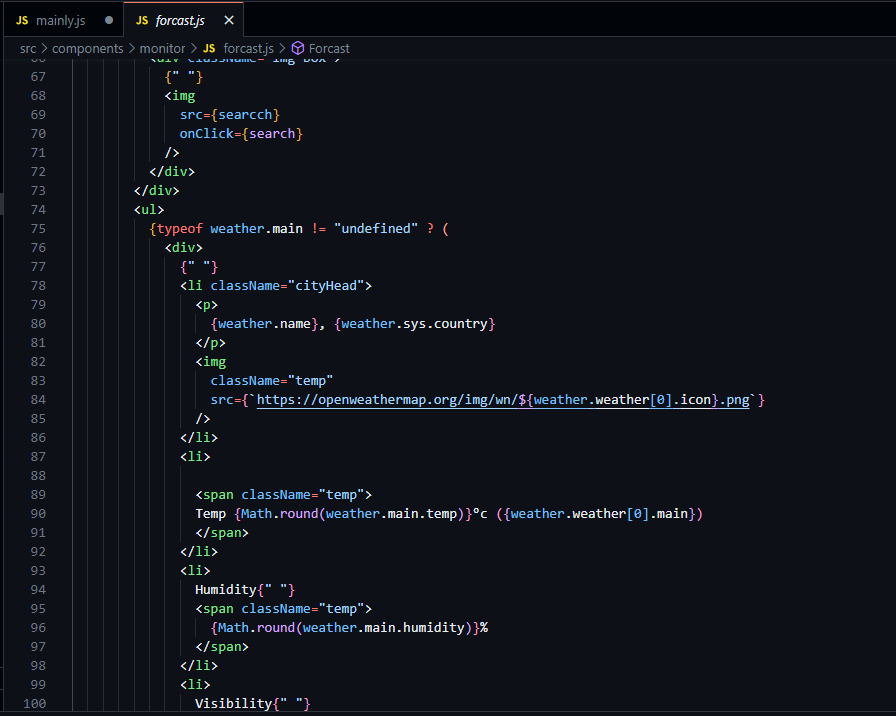
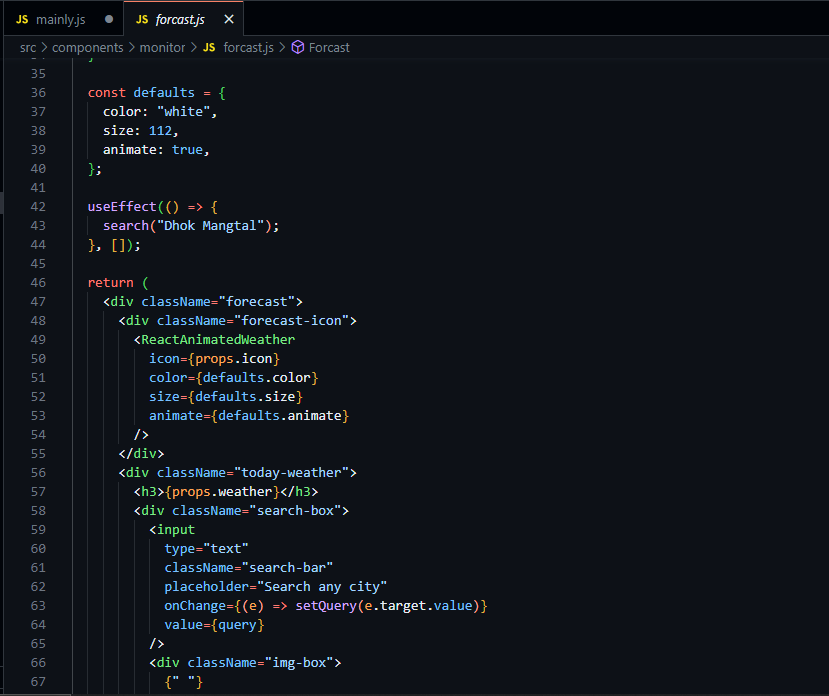
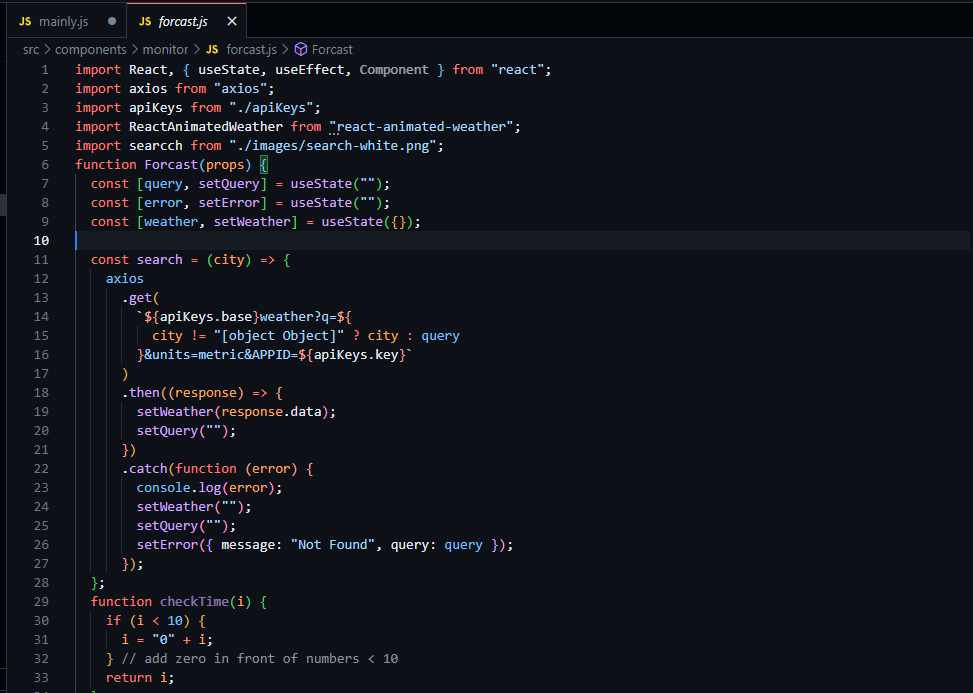
**Summary:**

The provided code exports an object with two properties: **key** and **base**. This module is used to store API-related configurations for accessing weather data. Here's a summary of the code:

* **Exported Object:** The module exports an object with two properties:
  + **key**: A string representing the API key for accessing the OpenWeatherMap API.
  + **base**: A string representing the base URL of the OpenWeatherMap API endpoint (**https://api.openweathermap.org/data/2.5/**).

**Note:** The API key is a sensitive piece of information and should be kept secure. It is typically used to authenticate and authorize requests to the API. Ensure that the API key is kept confidential and not exposed in client-side code or public repositories.

**Code:**



**Summary:**

The provided code defines a React functional component called **Forcast** that displays weather forecast information for a given city. Here's a summary of the code:

* **Imports:**
  + The component imports necessary dependencies such as React, useState, useEffect from React, axios for making API requests, **apiKeys** for storing API configurations, and **ReactAnimatedWeather** for displaying weather icons.
  + It also imports an image file named **search-white.png** for the search icon.
* **Component Functionality:**
  + The component initializes state variables using the **useState** hook. **query** is used to store the user's input for city search, **error** is used to handle errors, and **weather** is used to store the retrieved weather data.
  + The **search** function makes an API request to OpenWeatherMap using the **axios** library. It handles successful responses by updating the **weather** state with the retrieved data and resetting the **query**. In case of errors, it updates the **error** state with an error message.
  + The **checkTime** function is defined but not utilized in the component.
  + The component contains JSX elements for displaying weather information, including the weather icon, city name, temperature, humidity, visibility, and wind speed. It also provides a search bar for users to input a city name for weather data retrieval.
  + The weather information is displayed conditionally based on the availability of data. If data is available, it displays the weather details; otherwise, it shows an error message.
* **Default Weather Icon:**
  + The component receives a **props.icon** value and uses the **ReactAnimatedWeather** component to display the weather icon. The default weather icon properties such as color, size, and animation are specified.
* **Initial API Request:**
  + The **useEffect** hook is used to make an initial API request for weather data of a default city ("Dhok Mangtal") when the component is first rendered.

**Note:** Ensure that the API key (**apiKeys.key**) is kept secure and not exposed in client-side code or public repositories to maintain the security of your API access.

All Summary:

**1. Backend Model Training:**

* **Data Preparation:**
  + **Dataset:** Prepared a dataset containing images of healthy wheat leaves and leaves affected by diseases like septoria and stripe rust.
  + **Preprocessing:** Implemented resizing and normalization techniques on images. Applied data augmentation methods to augment the dataset for robust model training.
* **Model Architecture:**
  + **Deep Learning Model:** Utilized TensorFlow and Keras to construct a Convolutional Neural Network (CNN).
  + **Layers:** Designed the model with convolutional layers for feature extraction, pooling layers for dimensionality reduction, and dense layers for classification.
  + **Output Layer:** Implemented the output layer with softmax activation for multi-class classification.
* **Training and Validation:**
  + **Loss Function:** Employed categorical cross-entropy as the loss function for training the model.
  + **Optimizer:** Used the Adam optimizer to minimize the loss function during training.
  + **Validation:** Split the dataset into training and validation sets, enabling model evaluation and preventing overfitting.

**2. FastAPI Backend API:**

* **Framework & Middleware:**
  + **FastAPI Implementation:** Developed a backend API using FastAPI, leveraging its simplicity and performance.
  + **CORS Middleware:** Integrated CORS middleware to handle cross-origin requests securely.
* **Model Integration and Prediction:**
  + **Loading the Model:** Loaded the trained TensorFlow model into the FastAPI application.
  + **Prediction Endpoint:** Created an API endpoint ("/predict") to receive images, process them, and return disease predictions along with confidence scores.
  + **Health Check Endpoint:** Implemented a health check endpoint ("/ping") to ensure the API's availability.
* **Image Processing:**
  + **Image Handling:** Utilized PIL and NumPy to process images received as API requests.
  + **Conversion:** Converted images into NumPy arrays, making them compatible with the model for prediction.

**3. Frontend React Application:**

* **Components & Views:**
  + **Main Dashboard:** Designed the main dashboard with interactive elements for user engagement.
  + **Image Upload:** Created an image upload component allowing users to upload images for disease classification.
  + **Monitoring Section:** Implemented a monitoring section to display real-time weather information and disease predictions.
  + **Navigation Bar:** Developed a navigation bar for seamless navigation between different sections of the application.
* **User Interface & Interactivity:**
  + **Responsive Design:** Ensured responsiveness across various devices and screen sizes for optimal user experience.
  + **Material-UI Integration:** Integrated Material-UI components for a consistent and visually appealing UI design.
  + **Real-time Updates:** Enabled real-time updates for weather information, providing users with the latest data.

**4. Weather Integration and Additional Features:**

* **OpenWeatherMap API:**
  + **Weather Data:** Integrated OpenWeatherMap API to fetch real-time weather information based on the user's location.
  + **Dynamic Icons:** Utilized ReactAnimatedWeather to display dynamic weather icons corresponding to the current weather conditions.
* **Additional Features:**
  + **Search Functionality:** Implemented a search functionality allowing users to search for weather information in specific locations.
  + **Error Handling:** Implemented error handling mechanisms for API requests, ensuring a smooth user experience even in case of errors.

**5. Code Organization and Best Practices:**

* **Modularization:**
  + **Backend:** Organized backend code into separate modules, promoting modularity and ease of maintenance.
  + **Frontend:** Structured frontend codebase with separate files for components, styles, and utility functions, ensuring a clear code organization.
* **Security and Best Practices:**
  + **Sensitive Information:** Ensured the security of sensitive information such as API keys, following best practices for secure handling.
  + **Code Quality:** Adhered to coding standards, ensuring readability, maintainability, and collaborative development.
  + **Documentation:** Maintained comprehensive documentation, describing each module, function, and endpoint for future reference and collaboration.

This detailed summary covers the entire development process, from data preparation and model training to backend API implementation and frontend user interface design. It also highlights the integration of real-time weather data, error handling strategies, and adherence to best practices in coding and security.

Top of Form