DOCUMENTATION

GOOGLE KEEP CLONE

1. What is @emotion/react?

@emotion/react is a popular library for styling React components using CSS-in-JS (JavaScript) syntax. It provides a set of utility functions and React components that enable you to write and manage styles in a more modular and dynamic way.

2. What is @emotion/styled ?

@emotion/styled is a library that extends the functionality of @emotion/react and provides a way to create styled components in React using a CSS-in-JS approach.

styled is very similar to css except you call it with an html tag or React component and then call that with a template literal for string styles or a regular function call for object styles.

3. What is @mui/icons-material?

@mui/icons-material is a package provided by Material-UI, a popular UI component library for React.

With @mui/icons-material, you can easily include icons in your React application without the need for additional icon libraries or external resources. The package provides a wide range of icons, including common symbols, actions, social media logos, and more.

4. What is @mui/material?

@mui/material is the main package provided by Material-UI, a popular UI component library for React. It offers a comprehensive collection of pre-built, customizable, and reusable UI components following the principles of Material Design.

These components include buttons, forms, dialogs, navigation menus, grids, cards, and many more.

5. What is react?

React is a JavaScript library for building user interfaces.

It is a fundamental dependency for building React applications. It enables importing and using React components, hooks, and other features necessary for creating UI components, managing state, handling events, and more.

6. What is react-dom?

react-dom is a package that serves as the entry point for rendering React applications in the browser. It provides the necessary methods and APIs for interacting with the Document Object Model (DOM) and rendering React components into the browser environment.

7. What is react-router-dom?

React Router DOM is an npm package that enables you to implement dynamic routing in a web app. It allows you to display pages and allow users to navigate them. It is a fully-featured client and server-side routing library for React.

8. What is react-scripts?

react-scripts are simply scripts to run the build tools required to transform React JSX syntax into plain JavaScript programmatically.

react-scripts handles many other tasks behind the scenes, such as transpiling modern JavaScript syntax, managing CSS and asset files, and providing a development environment with hot module reloading.

9. What is uuid?

The uuid package is a popular JavaScript library that provides functionality for generating and working with Universally Unique Identifiers (UUIDs). UUIDs are unique identifiers that are commonly used to identify and distinguish entities in various systems and applications.

Here is one example of a UUID: acde070d-8c4c-4f0d-9d8a-162843c10333

10. What is web-vitals?

The web-vitals package is a JavaScript library that provides tools and utilities for measuring and monitoring key web performance metrics. It is specifically designed to help developers track and analyze the performance of their web applications.

11. What is @testing-library/jest-dom?

@testing-library/jest-dom is a package that provides additional custom matchers and utilities for enhancing the testing experience when using Jest as a testing framework.

With @testing-library/jest-dom, you can use special "matchers" to check if certain things are true in your tests. For example, you can easily check if an element is present on the screen, if it has a specific text, or if it has certain attributes. These matchers make it easier to write assertions that accurately reflect what you expect to see in your components.

12. What is @testing-library/react?

@testing-library/react is a JavaScript testing library specifically designed for testing React components.

With @testing-library/react, you can render React components in a test environment, interact with them, and make assertions about their rendered output and behavior. The library promotes a user-centric testing approach, focusing on how users would interact with the components rather than testing implementation details.

13. What is @testing-library/user-event

user-event is a companion library for Testing Library that provides more advanced simulation of browser interactions than the built-in fireEvent method.

14. What is react-beautiful-dnd?

react-beautiful-dnd is an Atlassian's open source library that allows web developers to easily integrate drag-and-drop functionality into their applications.

15. What is @hello-pangea/dnd?

@hello-pangea/dnd is an open-source library that provides drag-and-drop functionality for your applications, similar to react-beautiful-dnd. It serves as an alternative to react-beautiful-dnd in scenarios where the latter may not work properly or if there are concerns about its maintenance since it is no longer actively maintained by Atlassian.

Understanding some of the concepts inside the Components.

1. App.js

In React, context is a way to share data or state across multiple components without explicitly passing it through props. The DataProvider component likely creates a context and provides some data or functionality to be consumed by its child components.

```
function App() {
  return (
    <DataProvider>
      <Home />
    </DataProvider>
  );
}
```

By wrapping **Home** with the **DataProvider**, the Home component and any of its child

components can access the data or functionality provided by the **DataProvider context**. This allows Home and its child components to consume and utilize the shared data or functionality without the need for prop drilling (passing down the data through multiple levels of components).

2. DataProvider.jsx

In this component, a context called **DataContext** is created using the **createContext** function provided by React. The DataContext context will be used to share data across components in the application.

```
const DataProvider = ({children})=>{

    const [notes, setNotes] =useState([]);
    const [archiveNotes, setArchiveNotes] =useState([]);
    const [deletedNotes, setDeletedNotes] =useState([]);

    return(
        <DataContext.Provider
            value={{
                notes,
                setNotes,
                archiveNotes,
                setArchiveNotes,
                deletedNotes,
                setDeletedNotes
            }}
        >
            {children}
        </DataContext.Provider>
    )
```

The **DataProvider** component is also defined, which serves as the provider for the DataContext. It wraps its children components, indicated by the **children** prop, with the **DataContext.Provider.**

Within the DataProvider component, three state variables are defined using the useState hook: notes, archiveNotes, and deletedNotes. These state variables are used to manage the data related to notes, archived notes, and deleted notes, respectively.

The **value** prop of the **DataContext.Provider** is set to an object that contains the state variables and their corresponding setter functions. This allows components that consume the DataContext to access and update these state values.

By using the DataContext.Provider, any component nested within the DataProvider

component can access the provided values via the useContext hook or by wrapping the component with the DataContext.Consumer component.

3. Form.jsx

The **Form** component represents a form for adding notes.

A **note object** is defined with initial properties for id, heading, and text.

A **containerRef** is created using the **useRef hook** to reference the container element of the form. It is used to target and manipulate the specific element and its properties.

In the JSX part of the component, a **ClickAwayListener** component is used to detect clicks outside the form and trigger the handleClickAway function.

Inside the **ClickAwayListener**, the Container component is rendered, representing the note form. If **showTextField** is **true**, a text field is displayed for the note's heading. The **TextField** component is also rendered for the note's text, which is a multiline input.

The **handleClickAway function** handles the logic of resetting the form and updating the notes state when a click occurs outside the form.

```
const handleClickAway = () => {
    setShowTextField(false);

    containerRef.current.style.minHeight='30px';

    setAddNote({...note, id: uuid()});

    if(addNote.heading || addNote.text)
    {
        setNotes(prevArr => [addNote, ...prevArr]);
    }
    console.log(notes);
}
```

setAddNote({...note, id: uuid()}): This line updates the addNote state by spreading the note object and assigning a new id generated using the uuid() function. This resets the addNote state to an empty note with a new unique identifier.

By updating the addNote state with a new empty note object and unique identifier using setAddNote({...note, id: uuid()}), you ensure that the addNote state is reset to an empty note every time you click away, preventing any unintended duplication of notes.

if(addNote.heading || addNote.text) { setNotes(prevArr => [addNote, ...prevArr]) }: This condition checks if either the heading or text property of the addNote state has a value. If there is content in either of them, a new note object is created using the current addNote state, and it is added to the notes state using the setNotes function. The existing notes are preserved by spreading prevArr and prepending the new note to the array. This ensures that the new note is added to the beginning of the notes array, preserving the order of previously added notes.

4. Note.jsx

The **Note** component represents a single note card in the application. The component receives a note prop, which contains the data for the specific note to be displayed.

```
const archiveNote = (note)=>{
    const updatedNotes = notes.filter(data => data.id !== note.id);
    setNotes(updatedNotes);
    setArchiveNotes(prevArr => [note, ...prevArr]);
}
```

The **archiveNote function** takes the note object as an argument, representing the note to be archived.

Within the **archiveNote function**, the existing array of notes (notes) is filtered using the filter method. The filter condition checks for each note's id property and excludes the note being archived. This creates a new array (updatedNotes) without the archived note.

data.id represents the id of each note in the notes array, and note.id represents the id of the particular note you want to archive.

The **setNotes** function, which likely comes from the **DataContext**, is called to update the state with the **updatedNotes array**. This effectively removes the archived note from the list of active notes.

The **setArchiveNotes function**, also likely coming from the **DataContext**, is called to

update a separate array (archiveNotes) that stores the archived notes. The archiveNote is appended to the beginning of the array using the spread operator and the previous array values (prevArr).

```
const deleteNote = (note) => {
    const updatedNotes = notes.filter(data => data.id !== note.id);
    setNotes(updatedNotes);
    setDeletedNotes(prevArr => [note, ...prevArr]);
}
```

The **deleteNote function** serves the same purpose as the **archiveNote function** but with a slight difference. It removes a note from the notes array and adds it to the **deletedNotes** array.

5. Notes.jsx

The **Notes** component represents a collection of notes that can be dragged and dropped using the @hello-pangea/dnd library.

The **DataContext** is imported from the **DataProvider context**, which suggests that the Notes component needs access to the notes state value provided by the context.

Within the Notes component, the notes state and setNotes function are obtained from the DataContext using the useContext hook.

In the JSX part of the component, a grid layout is used to display the notes. If there are notes present **(notes.length > 0)**, the drag and drop functionality is enabled using the DragDropContext, Droppable, and Draggable components from @hello-pangea/dnd. The notes array is mapped over, and each note is rendered as a draggable item using the Draggable component. The **Note component** is then rendered inside the draggable item.

If there are no notes **(notes.length === 0)**, the **EmptyNotes component** is rendered, indicating that there are no notes to display.

The **reorder function** is defined, which is a helper function used to reorder the notes array based on the drag and drop result.

```
const onDragEnd = (result) => {

    if (!result.destination) {
        return;
    }

    const items = reorder(
        notes,
        result.source.index,
        result.destination.index
    );

    setNotes(items);
}
```

The **onDragEnd function** is defined as a callback for the onDragEnd event of the DragDropContext. This function is executed when a note is dragged and dropped. It reorders the notes array based on the drag and drop result and updates the notes state using setNotes

APIs

- <DragDropContext /> - Wraps the part of your application you want to have drag and drop enabled for
- <Droppable /> - An area that can be dropped into. Contains <Draggable />s
- <Draggable /> - What can be dragged around

6. Archive.jsx

This component provides the functionality to unarchive or delete a note from the **archiveNotes array**, depending on the user's interaction with the corresponding icons. It utilizes the shared data and functions from the DataContext to update the state and reflect the changes in the application.

```
const UnarchiveNote = (note)=>{
    const updatedNotes = archiveNotes.filter(data => data.id !== note.id);
    setArchiveNotes(updatedNotes);
    setNotes(prevArr => [note, ...prevArr]);
}

const deleteNote = (note)=>{
    const updatedNotes = archiveNotes.filter(data => data.id !== note.id);
    setArchiveNotes(updatedNotes);
    setDeletedNotes(prevArr => [note, ...prevArr]);
}
```

UnarchiveNote Function:

- The UnarchiveNote function is triggered when the "Unarchive" icon is clicked.
- It takes the selected note as a parameter.
- It filters out the selected note from the archiveNotes array using the filter method, creating a new array (updatedNotes) without the selected note.
- The setArchiveNotes function is used to update the archiveNotes state with the updatedNotes array.
- The setNotes function is used to add the selected note to the beginning of the notes array, effectively unarchiving the note.

deleteNote Function:

- The deleteNote function is triggered when the "Delete" icon is clicked.
- It takes the selected note as a parameter.
- Similar to UnarchiveNote, it filters out the selected note from the archiveNotes array and updates the archiveNotes state.
- Additionally, it adds the selected note to the beginning of the deletedNotes array using the setDeletedNotes function

7. Archives.jsx

This component displays the archived notes by rendering the **Archive component** for each archived note in the archiveNotes array obtained from the shared data context. The notes are arranged in a grid layout using the Grid component, and the Archive component is responsible for rendering each individual archived note.

8. DeleteNote.jsx

This component provides the functionality to delete or restore a note from the notes array, depending on the user's interaction with the corresponding icons. It utilizes the shared data and functions from the DataContext to update the state and reflect the changes in the application.

```
const restoreNote = (note)=>{
    const updatedNotes = deletedNotes.filter(data => data.id !== note.id);
    setDeletedNotes(updatedNotes);
    setNotes(prevArr => [note, ...prevArr]);
}

const deleteNote = (note)=>{
    const updatedNotes = notes.filter(data => data.id !== note.id);
    setDeletedNotes(updatedNotes);
}
```

restoreNote Function: Define the restoreNote function, which is triggered when the "Restore" icon is clicked. It takes the selected note as a parameter. Inside the function:

- Filter out the selected note from the deletedNotes array using the filter method, creating a new array updatedNotes without the selected note.
- Update the deletedNotes state by calling the setDeletedNotes function with the updatedNotes array.
- Add the selected note to the beginning of the notes array using the setNotes function, effectively restoring the note.

deleteNote Function: Define the deleteNote function, which is triggered when the "Delete" icon is clicked. It takes the selected note as a parameter. Inside the function:

- Filter out the selected note from the notes array using the filter method, creating a new array updatedNotes without the selected note.
- Update the deletedNotes state by calling the setDeletedNotes function with the updatedNotes array.

9. DeleteNotes.jsx

This component renders a list of deleted notes by iterating over the **deletedNotes array**. For each note, it renders a **DeleteNote component**, passing the note as a prop. It utilizes the shared DataContext to access the deletedNotes array.