REDUX PERSIST

Context:

const persistConfig={key: "root", storage, version: 1};

//persisted state stored under defined key

const persistedReducer= persistReducer(persistConfig, authReducer);

//we are basically creating a central redux store using the created reducer so that dispatch actions make sure it is sent to store

const store=configureStore({

  reducer: persistedReducer,

  middleware: (getDefaultMiddleware)=>

    getDefaultMiddleware({

      serializableCheck:{

        ignoredActions:[FLUSH,REHYDRATE,PAUSE,PERSIST,PURGE,REGISTER],

      },

    })

});

**Serialization**

**Serialization** is the process of converting an object or data structure into a format that can be easily stored or transmitted and later reconstructed. Commonly, data is serialized into formats like JSON, XML, or binary formats.

In the context of Redux and JavaScript, **serialization** typically means converting JavaScript objects into JSON, which is a text-based format that can be easily stored in localStorage, sent over a network, or persisted in a database.

**Why Serialization Matters in Redux**

Redux state and actions should be serializable for a few reasons:

1. **Persistence**: To save the state to localStorage or sessionStorage (as Redux Persist does), the state must be in a serializable format (like JSON).
2. **Debugging**: Redux DevTools relies on the ability to serialize and deserialize actions and state to provide time-travel debugging.
3. **Data Integrity**: Ensuring state and actions are serializable helps maintain predictable and consistent state management.

**Non-Serializable Data**

Certain JavaScript objects and data types are **non-serializable**, meaning they cannot be converted to JSON. Examples include:

* Functions
* Promises
* Symbols
* Class instances (custom objects that are not plain JavaScript objects)
* Circular references (objects referencing themselves)

**serializableCheck Middleware**

Redux Toolkit includes a **serializableCheck** middleware by default. This middleware:

* Ensures that all actions and the state are serializable.
* Warns or throws errors if non-serializable data is detected.

**Ignoring Certain Actions**

Some libraries, like Redux Persist, involve actions that are inherently non-serializable (e.g., involving dates, functions, etc.). Therefore, we need to tell **serializableCheck** to ignore these specific actions:

javascript

Copy code

ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],

### Redux Persist Actions

1. **FLUSH**
2. **REHYDRATE**
3. **PAUSE**
4. **PERSIST**
5. **PURGE**
6. **REGISTER**

### Explanation of Each Action

#### 1. FLUSH

**Purpose**: Ensures that any persisted state is written to storage.

**Why Ignored**: This action might involve complex data operations that may not be serializable, such as writing to the storage asynchronously.

#### 2. REHYDRATE

**Purpose**: Rehydrates the Redux state from the persisted storage during store initialization.

**Why Ignored**: When rehydrating state, the middleware might handle non-serializable values like Dates or custom objects that were previously stored.

#### 3. PAUSE

**Purpose**: Pauses persistence, stopping any updates to the persisted storage.

**Why Ignored**: Pausing persistence is a control action that may include non-serializable commands or state management logic.

#### 4. PERSIST

**Purpose**: Starts the persistence layer, enabling the persistence of the Redux state.

**Why Ignored**: This action initializes persistence and might include configuration details or callbacks that aren't serializable.

#### 5. PURGE

**Purpose**: Clears the persisted state from the storage.

**Why Ignored**: Purging the state involves operations to clear storage which might be non-serializable.

#### 6. REGISTER

**Purpose**: Registers a new reducer with the persistence layer.

**Why Ignored**: Registration might involve adding new reducers or middleware which aren't plain serializable objects.

### Why These Actions Are Ignored

* **Non-Serializable Data**: These actions often involve complex objects, functions, or non-serializable data types that are part of the persistence management process.
* **Control Operations**: They are control operations for managing the state persistence lifecycle and aren't directly related to the serializable state itself.
* **Middleware Behavior**: Ignoring these actions prevents the **serializableCheck** middleware from flagging them as errors, ensuring smooth operation of Redux Persist.

ReactDom CREATION OF ROOT:

### Role of root

In a React application, the **root** serves as the entry point for rendering your entire React component tree. Think of it as the starting point where React will mount your components into the DOM.

### Rendering Multiple React Components

Yes, you can absolutely render multiple React components within the selected **<div>** (the **root**). Once you've created the **root**, you can render any number of components inside it.

Here's an example of rendering multiple React components within the **root**:

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

import Header from './Header';

import Footer from './Footer';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<React.StrictMode>

<Header />

<App />

<Footer />

</React.StrictMode>

);

In this example:

* **<Header />**, **<App />**, and **<Footer />** are React components.
* Each component is rendered within the **root** using **root.render()**.
* The components will be mounted inside the **<div id="root"></div>** element in your HTML file.

**Benefits of Rendering Multiple Components**

Rendering multiple components within the **root** allows you to build complex UIs composed of smaller, reusable components. Each component can encapsulate its own logic and UI, making your codebase easier to manage and maintain.

**Summary**

* **root**: Serves as the entry point for rendering your React component tree.
* **Rendering Multiple Components**: You can render multiple React components within the **root**, allowing you to build complex UIs composed of smaller, reusable components.
* **Encapsulation**: Each component encapsulates its own logic and UI, promoting code reusability and maintainability.

I hope this clarifies the role of **root** and how you can render multiple React components within it! Let me know if you have any further questions or if there's anything else you'd like to understand.

MUI THEME SETTINGS

You create the themesettings object applying font changes and colors for different modes. Then you go to App.js and apply the following changes from these packages

We appy the themesettings using ThemeProvider

IMP NOTE: npm run start is used to start the front end development server(it runs sgtart script defined in package.json)

WE RUN npm run start WHILEIN THE FRONTEND FOLDER PATH ON THE TERMINAL. The default start point is index.js and index.js has a line that renders App so that begins

ICONBUTTON MUI

IconButton is the outer component specifying action on clicking. Within it we define the icons. To turn icons into buttons we define it inside IconButton

FORMCONTROL, SELECT:

**<FormControl>**

* **Role**: <FormControl> is a wrapper component that provides context and styling to form elements. It helps manage the form control state and integrates with other Material-UI form components.
* **Variant**: The variant="standard" prop specifies the style variant for the form control. Other possible variants include outlined and filled.
* **Value**: The value={fullName} prop sets the value of the form control.

**<Select>**

* **Role**: <Select> is a dropdown component that allows users to select an option from a list. It integrates with other Material-UI components to provide a consistent design and behavior.
* **Value**: The value={fullName} prop sets the currently selected value of the dropdown.
* **Styles**: The sx prop allows you to apply custom styles using the Material-UI sx syntax. This includes changing the background color, width, border-radius, padding, and styles for internal class names.

**Custom Styling with sx**

The sx prop is used to apply custom styles to the <Select> component. Here’s what each style does:

* **backgroundColor: neutralLight**: Sets the background color of the select element.
* **width: "150px"**: Sets the width of the select element.
* **borderRadius: "0.25rem"**: Rounds the corners of the select element.
* **padding: "0.25rem 1rem"**: Adds padding inside the select element.
* **"& .MuiSvgIcon-root"**: Targets the internal SVG icon used in the select dropdown, changing its padding and width.
* **"& .MuiSelect-select:focus"**: Targets the select element when it is focused, changing its background color.
* In Material-UI, the value prop on a <MenuItem> component is used to define the value associated with that particular option in the <Select> dropdown. When a user selects a MenuItem, the value of the selected MenuItem is passed to the Select component's value prop. This allows the Select component to know which option is currently selected and to display the corresponding MenuItem label.

**Explanation:**

It seems like you're creating a form control with a dropdown select menu. The Select component renders the dropdown, and MenuItem components represent the options inside the dropdown. The FormControl is used to manage the state and layout of the form control.

It's likely that fullName is a variable holding the name of a user or something similar, and it's being used as the value for the select dropdown and the label for one of the options.

The sx prop is used to style the components, including the dropdown, its options, and the SVG icons inside it.

If you need more specific help or have further questions about any part of this code, feel free to ask! *Slatt!*

3.5

**1. styled() function:**

The styled() function is part of the Styled Components library, but in your context, it looks like it is being used from a similar library like Material-UI (@mui/system). These libraries allow you to create custom-styled components using a template literal syntax or an object.

**2. Box component:**

In Material-UI, Box is a versatile component that serves as a wrapper for other components or content. By default, it renders a div element. It allows you to apply common CSS properties directly via props or by passing a sx prop for a more detailed styling.

**3. styled(Box):**

When you pass Box to the styled() function, you are creating a new component that inherits all the properties and behaviors of the Box component, but with additional styles applied.

MARGIN PADDING

**Margin:**

* **Definition**: Margins are the space outside the border of an element. They create space between elements.
* **Usage**: Used to create space around an element, pushing adjacent elements away.
* **Impact on Layout**: Margins affect the positioning of elements relative to their parent and sibling elements.
* **Centering**: margin: auto can be used to center block-level elements horizontally within their parent container.
* **Example**: margin: 2rem auto; sets 2rem top and bottom margins and centers the element horizontally within its parent.

**Padding:**

* **Definition**: Padding is the space between the content of an element and its border. It creates space inside the element.
* **Usage**: Used to create space inside an element's boundaries.
* **Impact on Layout**: Padding affects the size and positioning of the element's content relative to its border.
* **Centering**: Padding does not center elements; it affects the space inside an element but doesn't move the element itself.
* **Example**: padding: 1rem; sets 1rem of padding inside all four sides of the element.

FORMIK

Formik is a React library for handling forms, and in this snippet, it is used to manage form state, validation, and submission. The code provides a form that conditionally handles either a login or registration scenario based on the isLogin flag.

**Detailed Breakdown**

1. **Formik Component**:
   * <Formik> is the main component that initializes the form. It takes several props to set up the form.
2. **Props Passed to Formik**:
   * **onSubmit={handleFormSubmit}**: This prop specifies the function to call when the form is submitted.
   * **initialValues={isLogin ? initialValuesLogin : initialValuesRegister}**: This prop sets the initial values for the form fields. It conditionally uses either initialValuesLogin or initialValuesRegister based on the isLogin flag.
   * **validationSchema={isLogin ? loginSchema : registerSchema}**: This prop provides the validation schema for the form. It conditionally uses either loginSchema or registerSchema based on the isLogin flag.
3. **Render Prop Function**:
   * The Formik component uses a render prop pattern, meaning it takes a function as its child. This function receives an object with several properties and methods for managing the form.
4. **Destructuring the Render Prop Function**:
   * The function inside the Formik component destructures several properties and methods from the Formik context:
     + **values**: The current values of the form fields.
     + **errors**: An object containing validation errors for the form fields.
     + **touched**: An object indicating which fields have been visited (blurred).
     + **handleBlur**: A function to call when a form field loses focus.
     + **handleChange**: A function to call when the value of a form field changes.
     + **handleSubmit**: A function to call when the form is submitted.
     + **setFieldValue**: A function to programmatically set the value of a form field.
     + **resetForm**: A function to reset the form to its initial state.
5. **Form Structure**:
   * Inside the render prop function, a standard HTML form is created with the <form> tag.
   * The onSubmit={handleSubmit} attribute on the form ensures that the Formik handleSubmit method is called when the form is submitted.
6. **Box Component**:
   * The Box component (likely from Material-UI or a similar UI library) is used to style the form.
   * **display="grid"**: Sets the display of the Box to CSS Grid, enabling grid layout.
   * **gap="30px"**: Specifies the gap between grid items.
   * **gridTemplateColumns="repeat"**: This part of the code seems incomplete. Typically, you would specify the number and size of columns in a grid here, like gridTemplateColumns="repeat(2, 1fr)".

**Complete Example with Hypothetical Inputs:**

In Formik, the setFieldValue function is used to dynamically update the value of a field within a Formik-managed form. Here's a breakdown of what it does and how it's used:

1. **Purpose**:
   * setFieldValue is a function provided by Formik that allows you to programmatically set the value of a form field. This can be useful in scenarios where you need to update form values based on user interactions, API responses, or other dynamic data changes.
2. **Parameters**:
   * It typically takes two parameters:
     + field: string: The name or key of the form field whose value you want to update.
     + value: any: The new value you want to assign to the specified form field.
3. **Usage**:
   * You can call setFieldValue within event handlers (like onClick, onChange, etc.), useEffect hooks, or any other relevant places in your component's lifecycle where you need to update form values.

DROPZONE

* getRootProps() is obtained from the useDropzone() hook (assuming useDropzone is from a library like react-dropzone).
* {...getRootProps()} spreads all the necessary props onto the <Box> component, allowing it to handle drag-and-drop events.
* The <Box> component receives additional styling through the sx prop, which sets a dashed border with the primary color (border={2px dashed ${palette.primary.main}}) and changes the cursor to a pointer on hover (sx={{ "&:hover": { cursor: "pointer" } }}).

This approach combines the functionality provided by useDropzone for drag-and-drop interactions with styled components from Material-UI (@mui/system), enhancing both functionality and visual presentation within your React application.

These props could include event handlers (like onDragEnter, onDragOver, etc.), attributes (like role, aria-dropeffect, etc.), and any other necessary properties to ensure proper functioning of the drag-and-drop feature

getInputProps:  
In the context of libraries like react-dropzone, getInputProps() is used to manage and configure the <input> element that is used for file selection within a dropzone component. Let's break down why getInputProps() is added to the <input> element:

**Purpose of getInputProps()**

1. **Managing Input Properties**:
   * getInputProps() is a function provided by libraries like react-dropzone to generate and manage the necessary properties (props) for the <input> element used for file selection.
2. **Functionality**:
   * The <input> element is a fundamental part of the file upload process in HTML forms. By spreading {...getInputProps()} onto the <input> element, you automatically apply all the required properties and event handlers needed for file selection functionality.
   * This includes properties like type="file", onChange event handler to handle file selection, and other attributes that manage accepted file types, multiple file uploads, etc., based on the configuration provided to getInputProps().
3. **Integration with Dropzone Component**:
   * In components like react-dropzone, the dropzone area is typically designed to visually represent where users can drag and drop files. However, the actual file selection is handled by an <input type="file"> element.
   * getInputProps() ensures that this <input> element is correctly configured and integrated with the dropzone component, allowing users to either click to select files or drag files into the dropzone area.

**Example Usage:**