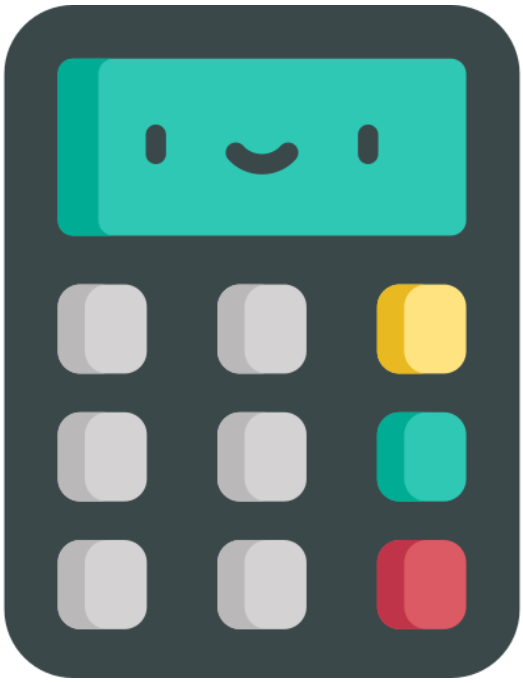
Retirement Calculator using React-redux



1.Project Overview

1.1 Description

The retirement calculator is a web application designed to assist users in planning for their retirement by estimating the required savings and monthly contributions. The primary goal of the project is to provide a user-friendly interface that allows individuals to input essential information such as their current age, desired retirement age, current retirement savings, current savings contribution, and the monthly income required at retirement. The calculator then employs a predetermined algorithm to compute the required retirement savings and monthly contribution.

1.2 Features

- User Input Form: A structured form that captures key financial details necessary for retirement planning.

- Calculation Logic: Utilizes a robust algorithm to process user input and calculate the required retirement savings and monthly contribution.
- Result Display: Presents the calculated results in an intuitive and visually appealing manner, using a donut chart and numerical values.
- Error Handling: Provides clear error messages in case of invalid inputs or calculation issues, ensuring a seamless user experience.
- FAQ Link: Includes a link to a FAQ section for users seeking additional information about the calculation methodology.
- Responsive Design: Implements responsive design principles for a consistent and accessible user experience across various devices.

2. Architecture and Technologies

2.1 Frontend

Frontend: React with TypeScript.
State Management: Redux.
Chart Library: ReactApexChart for visually representing the retirement plan results.
Styling: Tailwind CSS for a modern and responsive design.

2.2 State Management

In my project, state management is achieved through the use of Redux, a powerful state management library for React applications. The central hub for managing the application's state is the Redux store, configured using the configureStore function from the @reduxjs/toolkit package. This store acts as a container for all the data that the application needs.

To organize the state and actions related to the retirement calculator feature, I've structured the code using slices. In this context, a slice represents a portion of the overall application state and includes the reducer, actions, and initial state related to the calculator.

Actions in Redux represent the different ways in which the state can change. In my project, I've defined actions like setInputField and calculateRetirement. These actions describe the changes I want to make to the state when certain events occur, such as user input or the initiation of a retirement calculation.

Reducers, on the other hand, are responsible for specifying how the state should change in response to dispatched actions. The logic for handling these changes is encapsulated within the reducer functions.

Selectors are used to extract specific pieces of data from the Redux store. For the calculator feature, I've defined a selector called selectCalculator to easily access the calculator slice of the state.

Components play a crucial role in this state management flow. They use the useSelector hook to subscribe to changes in the state and selectively extract the data they need. When user interactions occur, components dispatch actions using the useDispatch hook, triggering the state changes specified in the reducers.

This approach to state management enhances the predictability and maintainability of the project. It provides a clear structure for handling state changes, making it easier to reason about the flow of data in the application. Additionally, the use of Redux devtools allows for efficient debugging and monitoring of state changes during development.

.

2.3 Chart Library

The chart library used for rendering the retirement plan results is **ReactApexChart**. The **ReactApexChart** library is employed to create visually appealing and interactive charts, enhancing the user experience when displaying the calculated results of the retirement savings and monthly contribution.

3. Components

3.1 Calculator Component

In the main Calculator component of my project, I've designed a structured and user-friendly interface for users to input their financial information and receive retirement planning results. Let me describe the key aspects of its structure and functionality:

Structure:

1. Header Section:
   - Displays a welcoming message, incorporating the user's name if provided.

2. Input Form:
   - Contains form fields for essential financial information, including the user's name, current age, retirement age, current retirement savings, current retirement savings contribution, and the monthly income required at retirement.

3. FAQ Link:
   - Provides a link to a FAQ section, guiding users to additional information about the calculation methodology.

4. Calculate Button:
   - Initiates the retirement calculation process when clicked.

5. Error Message Display:
   - Shows error messages if there are issues with user input or the calculation process.

6. Results Section:
   - Displays the calculated retirement plan results, including a donut chart using the ReactApexChart library and numerical values for required retirement savings and monthly contribution.

Functionality:

1. handleChange Function:
   - This function is triggered when any input field in the form is changed.
   - It dispatches the setInputField action to update the corresponding field in the Redux store based on the user's input.
   - Ensures controlled components by keeping the input values synchronized with the Redux state.

2. handleCalculate Function:
   - Invoked when the user clicks the "Calculate" button.

- Dispatches the calculateRetirement action, initiating the calculation process.
- Sets a state variable (isCalculated) to true, triggering the display of the calculated results.

3. Chart Rendering:
- Utilizes the ReactApexChart library to present a donut chart, visually representing the required retirement savings and monthly contribution.
- Takes input from the Redux store to dynamically update the chart based on the calculated results.

4. Result Display:
- Displays the calculated retirement plan results in a visually appealing format.
- Communicates the required retirement savings and monthly contribution in both numerical and graphical forms.

5. Error Handling:
- Displays error messages in a prominent location if there are issues with user input or the calculation process.
- Ensures a clear and informative user experience.

3.2 Redux Integration

In the Calculator component, I connect to the Redux store using the useSelector and useDispatch hooks:

1. useSelector Hook:
- Imports useSelector to access specific state slices.
- Retrieves the calculator slice from the Redux store, allowing the component to access state properties.

2. useDispatch Hook:
- Imports useDispatch to obtain the dispatch function.
- Uses dispatch to trigger actions, such as updating input fields or initiating calculations.

4. Input Handling

4.1 Controlled Components

In the Calculator component of my project, input fields follow the controlled component pattern. Here's a brief explanation:

Controlled Components:

- In controlled components, the value of an input field is controlled by the component's state.
- The input field's value is not managed by the DOM but is instead controlled through React state.

handleChange Function:

- The handleChange function is invoked when users interact with an input field.
- It dispatches the setInputField action to update the relevant field in the Redux store, making it the single source of truth for input values.

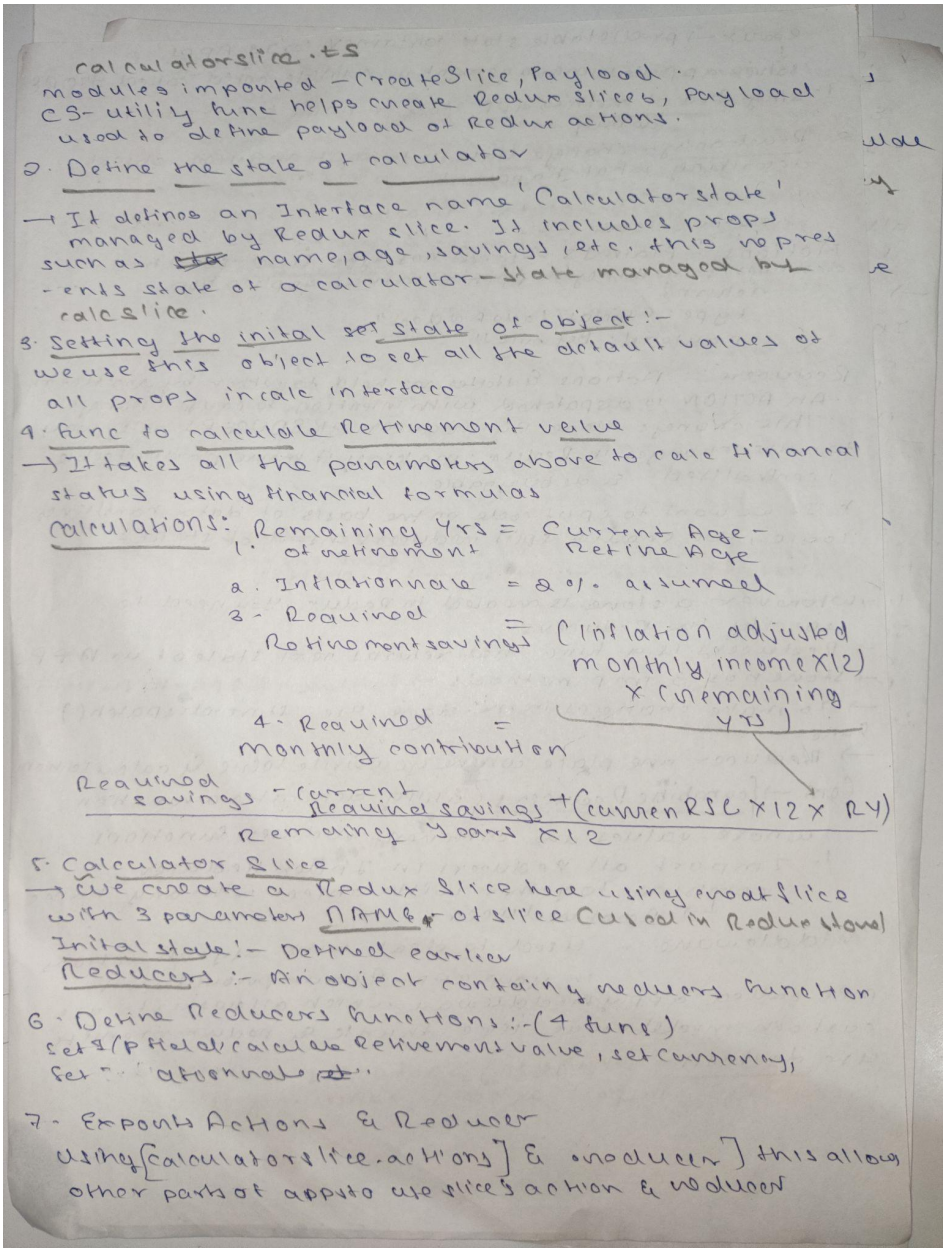
Redux Store Update:

- The setInputField action updates the Redux store with the new input value.
- This ensures that the input field value is synchronized with the global state, allowing for consistent and predictable data flow.

Benefits:

- Controlled components enable a straightforward way to manage and validate user input.
- The centralized state management in the Redux store ensures a unified and predictable data flow throughout the application.

5. Calculation Logic



calculatorslice.ts
modules imported – createSlice, PayloadAction
CS- utility func helps create Redux slices, PayloadAction used to define payload of Redux actions.

2. Define the state of calculator
→ It defines an interface name 'CalculatorState' managed by Redux slice. It includes props such as name, age, savings, etc. this represents state of a calculator – state managed by calcslice.

3. Setting the inital set state of object :-
we use this object to set all the default values of all props in calc interface

4. func to calculate Retirement value
→ It takes all the parameters above to calc financial status using financial formulas

calculations: 1. Remaining Yrs = Current Age – Retire Age of retirement
2. Inflationrate = 2% assumed
3. Required Retirement savings = (Inflation adjusted monthly income X12) X (remaining YRS)
4. Required monthly contribution =

$$\frac{\text{Required savings} - \text{current Required savings} + (\text{currenRSL} \times 12 \times \text{RY})}{\text{Remaing Years} \times 12}$$

5. Calculator Slice
→ We create a Redux slice here using createSlice with 3 parameters NAME of slice Called in Redux tool
Inital state :- Defined earlier
Reducers :- An object containg reducers function

6. Define Reducers functions :- (4 funs)
setS/P to calculate Retirement value, set currency, set ... attennual ...

7. Export Actions & Reducer
using [calculatorslice.actions] & [reducer] this allows other parts of app to use slice's action & reducer

In 'createSlice' → uses Immer library. We can write code that is mutable by state directly. Immer takes care by producing a new state in an immutable way

→ Destructing of exported values → directly extract the reducers & action creators allowing cleaner import statements. for example the increment & decrement (actions creators)

1. Code for calculatorslice.js

1. Interface calculatorState :- (Prop may be undefined) Optional field. Includes clear structure for the state Redux store to manage. The key: string allows flexible or dynamic prop that might add to state during app lifecycle.

2. Inital state :- Starting point of application

→ When Redux store is initalized, it will have this state structure.

→ As user interact with app, the state will be updated based on actions dispatched to the Redux store

→ Reducers handle (store) tx from ~~reducers~~ inital state

3. CalculateReturnValues :- [const calcu]

→ used toFixed(2) method to 2 decimal placeValues

→ number() function to convert string produced by fixed method back to numbers.

→ Return :- an obj with calculated values.

5.1 Redux Actions

setInputField Action:

Dispatched when users interact with input fields, updating specific properties in the Redux store related to user inputs (e.g., name, age, savings).

calculateRetirement Action:

Dispatched when users click the "Calculate" button.
Initiates the retirement calculation process, computing the required retirement savings and monthly contributions based on user inputs.

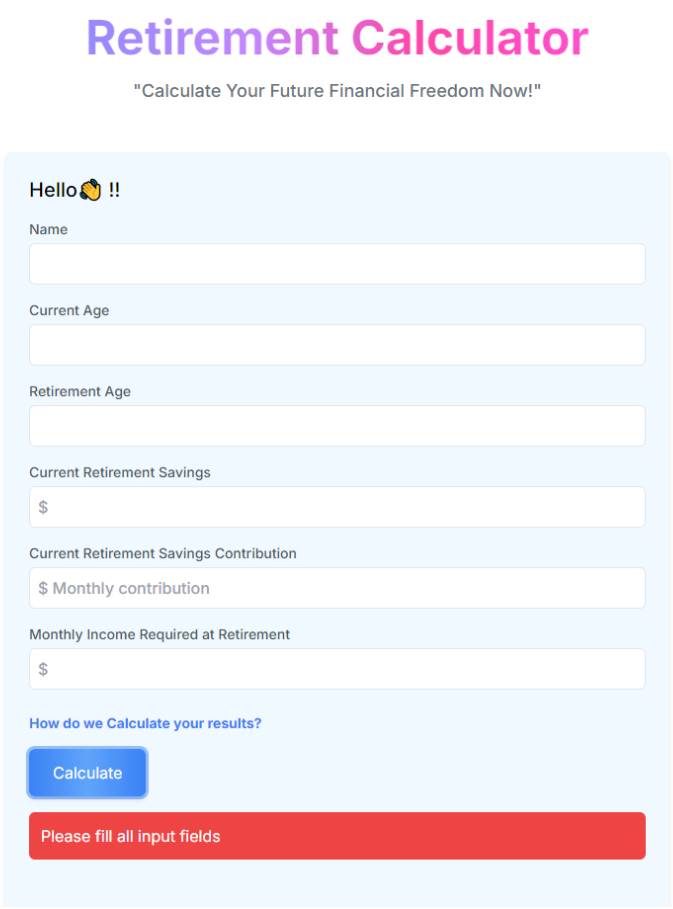
Logic in Reducer:

The calculateRetirement action triggers logic within the reducer, performing the actual calculation based on the provided user inputs.
The results are then updated in the Redux store.

5.2 Error Handling

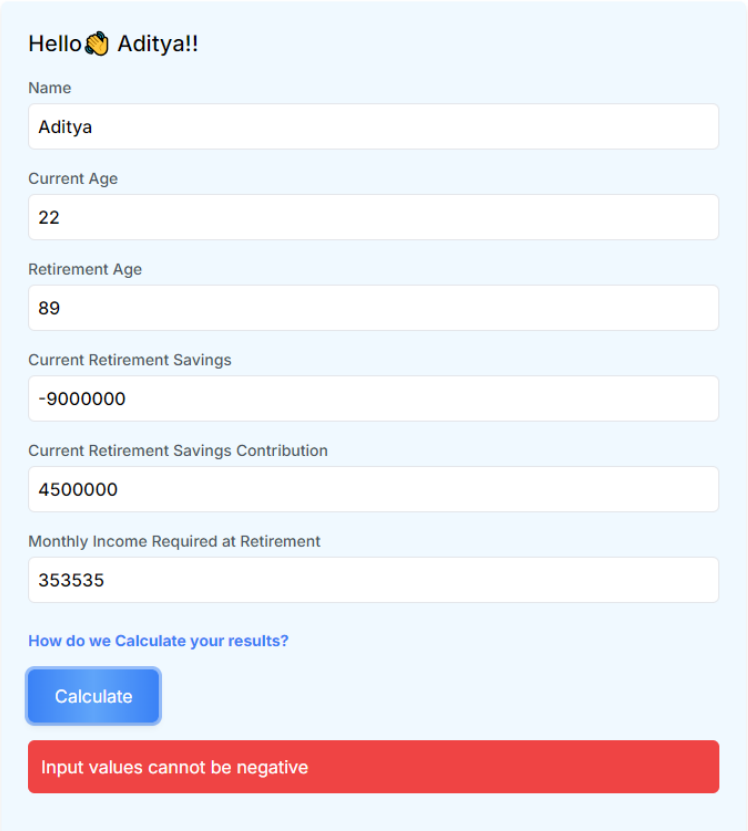
Required Fields Check

Before performing calculations, the calculateRetirement action checks if all required input fields (requiredFields) have been filled. If any required field is undefined, an error message "Please fill all input fields" is stored in the error property.



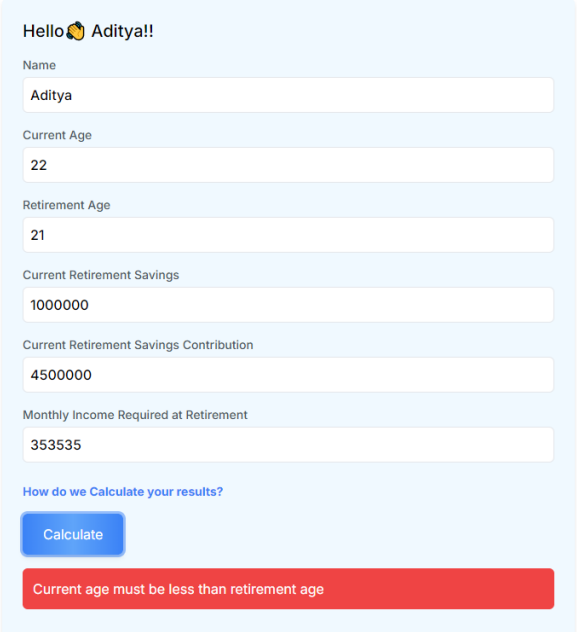
Negative Values Check:

The action further checks if any input values are negative. If any required field or the inflation rate is negative, an error message "Input values cannot be negative" is stored in the error property.



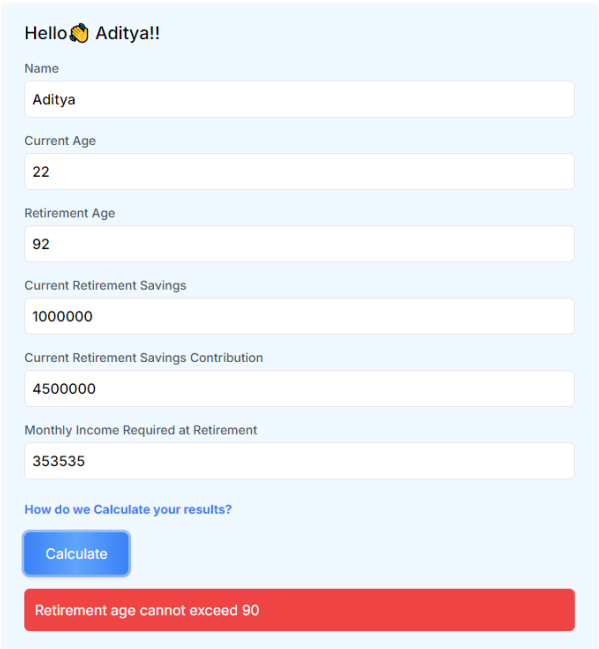
Current Age vs. Retirement Age Check

The action compares the inputted current age and retirement age. If the current age is equal to or greater than the retirement age, an error message "Current age must be less than retirement age" is stored in the error property.



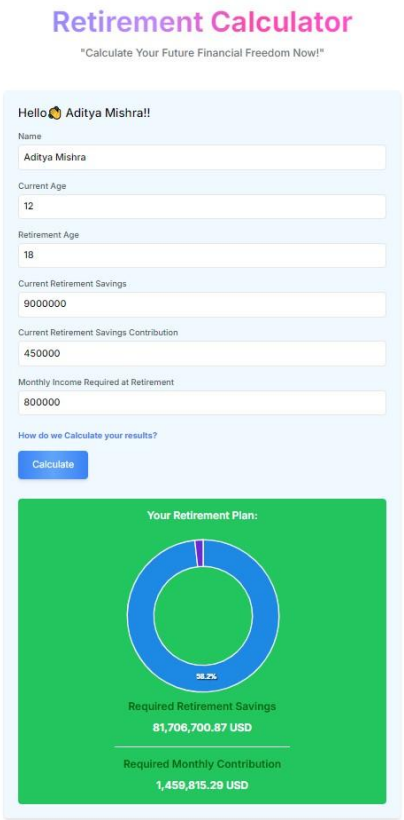
Retirement Age Limit

The action also checks if the retirement age exceeds 90. If the retirement age is greater than 90, an error message "Retirement age cannot exceed 90" is stored in the



error property.
Error Message Display: If any of these error checks are triggered, the calculateRetirement action does not proceed with calculations and instead stores the corresponding error message in the error property of the calculator state. This error message is then accessible to the user interface for display.

6. Result Display - Calculated Results:



In my project, the calculated results are presented in a visually appealing manner, utilizing the ReactApexChart component.

Result Container:

The Calculator component includes a designated section, the "result-container," where the calculated results are displayed.

Donut Chart with ReactApexChart:

The ReactApexChart library is employed to create a donut chart that visually represents the calculated results.
The chart is configured with options such as labels, colors, and legends to enhance the visual representation of required retirement savings and monthly contributions.

Numeric Display:

Alongside the donut chart, the numerical results are displayed in a structured format. The calculated values, such as required retirement savings and monthly contributions, are presented in a clear and easy-to-read manner.

Styling and Layout:

The result container and its contents are styled to provide a cohesive and visually pleasing display.
Layout elements, such as dividers, are used to enhance the overall presentation.

Conditional Rendering:

The result display section is conditionally rendered based on whether the calculation process has been completed successfully (isCalculated state variable).

Dynamic Updates:

As the Redux store is updated during the calculation process, the React component automatically re-renders to reflect the changes in the calculated results.

7. Conclusion

Calculator State: Defines the structure of the calculator state, including properties for user inputs (current age, retirement age, current savings, savings contribution, monthly income required, inflation rate) and calculated values (required savings, required monthly contribution).

Input Validation: Implements input validation to ensure users provide valid and consistent inputs before performing calculations. Checks for missing inputs, negative values, and unrealistic values (e.g., current age exceeding retirement age or retirement age exceeding 90).

Retirement Calculation: Defines a function to calculate the required retirement savings and monthly contribution based on the validated user inputs. Considers inflation in monthly income required and retirement savings goals.

Error Handling: Incorporates error handling to detect and display error messages to the user. Validates inputs before calculations and stores error messages in the calculator state if any inconsistencies are found.

Redux Integration: Utilizes Redux toolkit to manage the calculator state and handle state updates triggered by user actions (input changes, currency selection, inflation rate adjustment).