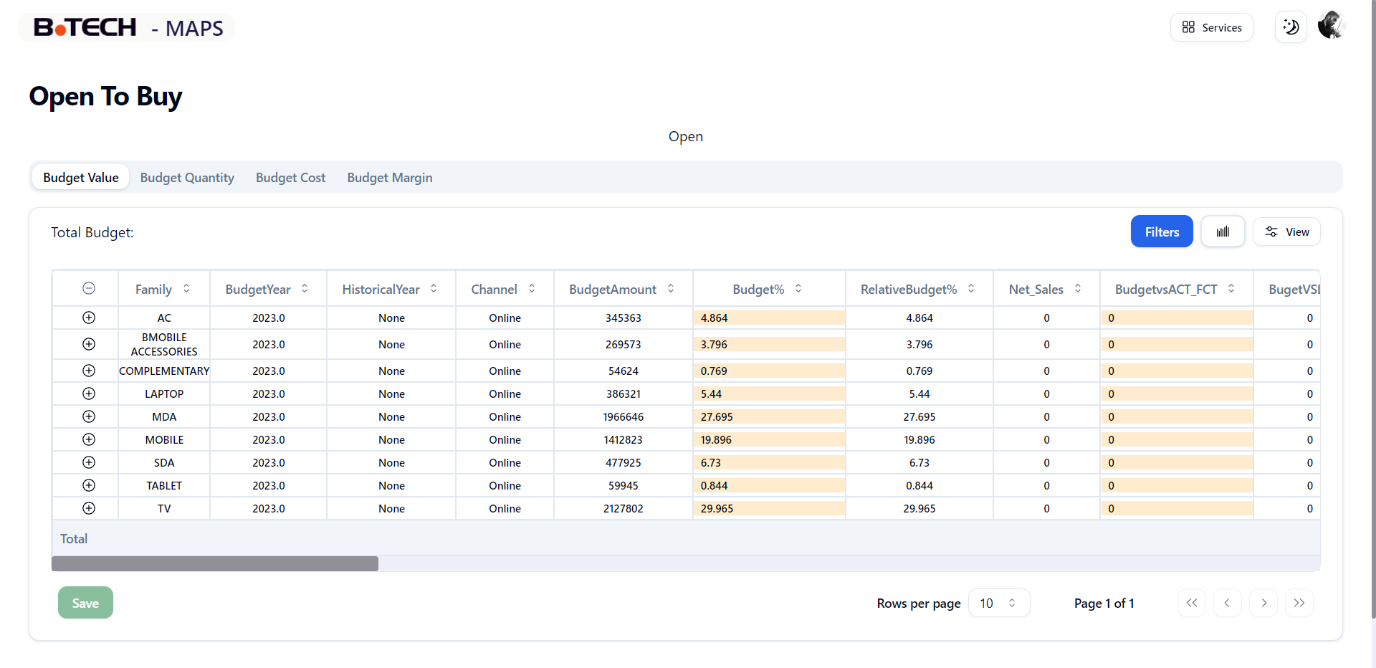
**OTB – Code Documentation**

**Introduction**

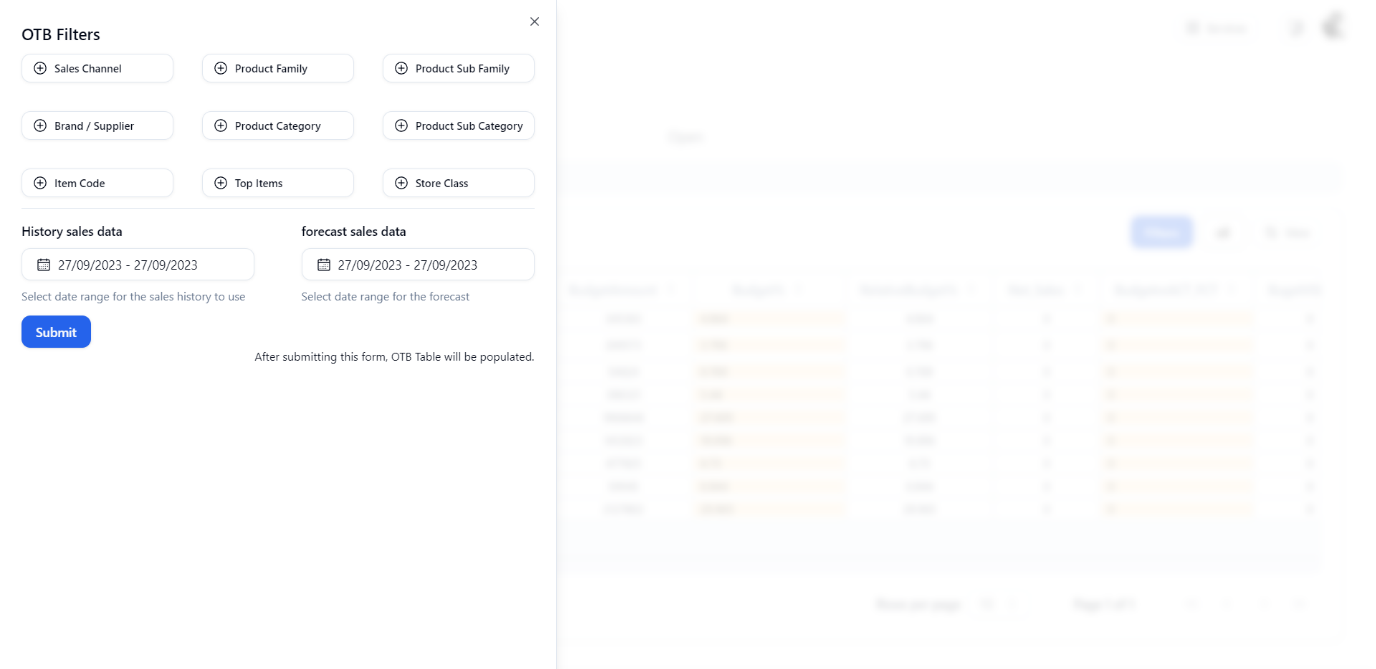
The BMPAS (Retail OTB Analysis System) project is designed to empower retailers with the analytical tools and insights needed to navigate market challenges and make informed decisions, especially for Open-to-Buy (OTB) transactions. Open To Buy (OTB) module, built using Next.js and TypeScript. OTB module is designed to provided with a user-friendly interface created using Radix UI. OTB is equipped with various filtering options, such as sales channel, product family, supplier, item code, and more, enabling you to analyse your inventory data with precision. Additionally, it offers graphical representations of your data using the Recharts library, enhancing your ability to make informed decisions regarding inventory management. In this document, we will provide a comprehensive explanation of the code and its functionalities.

Here’s a UI of OTB module:

Ui of otb module,



Ui of filters,



**Page**

**-**page.tsx

In Next.js, you can handle page routing by creating .tsx files for each of your pages. Here's a brief overview of how page routing works in Next.js using a page.tsx file. The name of default function in page.tsx that export is OtbPage() is given below:

export default function OtbPage() {

  return (

    <div className="flex-1 h-full space-y-4 p-8 pt-6">

      <OtbController />

    </div>

  );

}

**Description:**

1. Import OtbController component

2. Define OtbPage component

3. Return JSX for the component

4. JSX for the component includes a div with className "flex-1 h-full space-y-4 p-8 pt-6"

5. Inside the div, there are two elements:

- A div with className "flex items-center justify-between space-y-2" that contains an h2 element with text "Open To Buy"

- An instance of the OtbController component

6. The OtbPage component is exported as the default export

This code defines a React component called OtbPage that renders a page with a title "Open To Buy" and a controller for managing the Open To Buy functionality.

**OtbController**

**-** OtbController.tsx

OtbController is the parent component of all other components in OTB module, which returns the function OtbController() and all states and manged in this parent component and passed through context hook to child components.

**Description:**

1. Import the necessary hooks and components from React and Material-UI.

import { createContext, useEffect, useState } from "react";

import WebSocketData from "./Websocket";

import { EditableTableCell } from "./otb\_table/EditableCell";

import { ExpandComponent } from "./otb\_table/ExpandableComponent";

import CollapseComponent from "./otb\_table/CollapseComponent";

import { DateRange } from "react-day-picker";

import OtbTabView from "./OtbTabView";

2. Handling States

const [transformedData, setTransformedData] = useState([]);

  const [columnNames, setColumnNames] = useState([]);

  const [mainFilters, setMainFilters] = useState([]);

  const [subFilters, setSubFilters] = useState([]);

  const [mainFilterOptions, setMainFilterOptions] = useState({});

  const [mainFilterValues, setMainFilterValues] = useState({});

  const [loading, setLoading] = useState(false);

  const [webSocketData, setWebSocketData] = useState<any>({});

  const [expandHistory, setExpandHistory] = useState([]);

  const [history\_date, setHistoryDate] = useState<DateRange | undefined>({

    from: new Date(),

    to: new Date(),

  });

  const [forecast\_date, setForecastDate] = useState<DateRange | undefined>({

    from: new Date(),

    to: new Date(),

  });

This code is written in React, a popular JavaScript library for building user interfaces. It uses the useState hook to manage the state of several variables in the component.

Here's a breakdown of the code:

1. `const [transformedData, setTransformedData] = useState([]);`

This line initializes a state variable called `transformedData` with an initial value of an empty array. The `setTransformedData` function is used to update the value of `transformedData`.

2. `const [columnNames, setColumnNames] = useState([]);`

This line initializes a state variable called `columnNames` with an initial value of an empty array. The `setColumnNames` function is used to update the value of `columnNames`.

3. `const [mainFilters, setMainFilters] = useState([]);`

This line initializes a state variable called `mainFilters` with an initial value of an empty array. The `setMainFilters` function is used to update the value of `mainFilters`.

4. `const [subFilters, setSubFilters] = useState([]);`

This line initializes a state variable called `subFilters` with an initial value of an empty array. The `setSubFilters` function is used to update the value of `subFilters`.

5. `const [mainFilterOptions, setMainFilterOptions] = useState({});`

This line initializes a state variable called `mainFilterOptions` with an initial value of an empty object. The `setMainFilterOptions` function is used to update the value of `mainFilterOptions`.

6. `const [mainFilterValues, setMainFilterValues] = useState({});`

This line initializes a state variable called `mainFilterValues` with an initial value of an empty object. The `setMainFilterValues` function is used to update the value of `mainFilterValues`.

7. `const [loading, setLoading] = useState(false);`

This line initializes a state variable called `loading` with an initial value of `false`. The `setLoading` function is used to update the value of `loading`.

8. `const [webSocketData, setWebSocketData] = useState<any>({});`

The `setWebSocketData` function is used to update the value of the WebSocket state variable.

The code is a part below transforms data received from a WebSocket connection into a more readable format.

  const [body, setBody] = useState<any>({});

  function transformData(\_columns: any = [], \_data: any) {

    let arr: any = [];

    const \_transformedData = \_data?.map((item: any, index: any) => {

      let obj = {};

      item?.map((subItem: any, subIndex: any) => {

        obj = { ...obj, [\_columns[subIndex]]: subItem };

      });

      arr = [...arr, obj];

    });

    setColumnNames(\_columns);

    setTransformedData(arr);

  }

1. `const [body, setBody] = useState<any>({});`

This line of code is using the `useState` hook to create a state variable called `body`. The `useState` hook is a function that takes an initial value for the state variable and returns an array with two elements: the current value of the state variable and a function to update it.

In this case, the initial value of the state variable is an empty object (`{}`). The type of the state variable is specified as `any`, which means it can hold any type of value.

The `setBody` function is used to update the value of the `body` state variable.

2. `function transformData(\_columns: any = [], \_data: any) { ... }`

This function takes two arguments: `\_columns` and `\_data`. `\_columns` is an array of column names, and `\_data` is an array of data received from the WebSocket connection.

Inside the function, a new empty array called `arr` is created.

3. `\_data?.map((item: any, index: any) => { ... })`

This line of code uses the `map` function to iterate over each item in the `\_data` array. For each item, the following code is executed:

4. `let obj = {}`

This line of code creates a new empty object called `obj`.

5. `item?.map((subItem: any, subIndex: any) => { ... })`

This line of code uses the `map` function to iterate over each sub-item in the `item` array. For each sub-item, the following code is executed:

6. `obj = { ...obj, [\_columns[subIndex]]: subItem }`

This line of code updates the `obj` object by adding a new property with a key equal to the corresponding column name from the `\_columns` array and a value equal to the sub-item.

7. `arr = [...arr, obj]

This line of code adds the updated `obj` object to the `arr` array.

8. `setColumnNames(\_columns);

This line of code updates the `columnNames

 useEffect(() => {

    if (webSocketData) {

      transformData(webSocketData?.columns, webSocketData?.data);

    }

  }, [webSocketData]);

this code snippet is using the `useEffect` hook to listen for changes in the `webSocketData` variable. When `webSocketData` changes, the function inside `useEffect` is executed, which in turn calls the `transformData` function to transform the data received from the WebSocket.

  return (

    <OtbContext.Provider

      value={{

        body,

        setBody,

        mainFilters,

        setMainFilters,

        subFilters,

        setSubFilters,

        mainFilterOptions,

        setMainFilterOptions,

        mainFilterValues,

        setMainFilterValues,

        setWebSocketData,

        webSocketData,

        loading,

        setLoading,

        transformedData,

        setTransformedData,

        history\_date,

        setHistoryDate,

        forecast\_date,

        setForecastDate,

        expandHistory,

        setExpandHistory,

      }}

    >

      <WebSocketData />

      <OtbTabView columns={columns} />

    </OtbContext.Provider>

  );

The `OtbContext.Provider` is wrapped around two child components: `WebSocketData` and `OtbTabView`. The `WebSocketData` component is responsible for handling the WebSocket connection and receiving data from the server. The `OtbTabView` component is responsible for rendering the data in a tabular format

The `OtbContext.Provider` makes the context properties available to all child components of the component that uses the `OtbContext.Provider`. This allows the child components to access and modify the state of the parent component without having to pass the state down through props.

In summary, this code is a React component that uses a context to manage its state and provides this context to its child components. The child components use the context properties to render the data in a tabular format and to handle the WebSocket connection.

**Websocket**

**-** Websocket.tsx

This code is a React component that uses the `react-use-websocket` library to establish a WebSocket connection to a server. The component is responsible for sending messages to the server and receiving messages from the server.

1. Import necessary libraries and components:

- `React`, `useEffect`, and `useContext` are imported from the "react" library.

- `useWebSocket` is imported from the "react-use-websocket" library.

- `OtbContext` is imported from the "./OtbController" file.

- `useToast` is imported from the "@/components/shared/use-toast" file.

2. Define the `WebSocketData` component:

 const socketUrl = `${process.env.NEXT\_PUBLIC\_OTB\_WEBSOCKET\_BASE\_URL}/otb/get\_data\_ws`;

- The `socketUrl` variable is set to the WebSocket server URL. And its actual url is fetched form .env file.

const {

    sendMessage,

    lastMessage,

    readyState,

  } = useWebSocket(socketUrl, {

    shouldReconnect: (closeEvent) => {

      return true; },});

- The `useWebSocket` hook is used to establish a WebSocket connection to the server. The hook returns an object with several properties, including `sendMessage`, `lastMessage`, and `readyState`.

- The `useContext` hook is used to access the `OtbContext`, which provides access to the state and methods of the OTB application.

const { body, setWebSocketData, setLoading } =

    useContext(OtbContext);

  const data = lastMessage ? JSON?.parse(lastMessage?.data) : null;

  const { toast } = useToast();

- The `useToast` hook is used to display toast notifications to the user.

useEffect(() => {

    if (data) {

      setWebSocketData(data); }

    setLoading(false);

  }, [JSON?.stringify(data)]);

this code is a React component that uses the `useEffect` hook to listen for changes in the `data` object and update its state accordingly.

  function getOtbDataFromWebSocketApi(body: any) {

    sendMessage(JSON.stringify({...body,}));}

The `getOtbDataFromWebSocketApi` function is used to send a request to a WebSocket API and receive data in response. The `body` object is used to specify the details of the request, such as the type of data being requested or any filters that should be applied to the data.

 useEffect(() => {

    if (Object?.keys(body).length) {

      getOtbDataFromWebSocketApi(body);}}, [body]);

In this case, the useEffect function is an arrow function that checks if the `body` object has any keys. If it does, the function calls the `getOtbDataFromWebSocketApi` function and passes the `body` object as an argument.

 const connectionStatus = {

    [ReadyState.CONNECTING]: "Connecting",

    [ReadyState.OPEN]: "Open",

    [ReadyState.CLOSING]: "Closing",

    [ReadyState.CLOSED]: "Closed",

    [ReadyState.UNINSTANTIATED]: "Uninstantiated",}[readyState];

  useEffect(() => {

    window.addEventListener("offline", function () {

      toast({

        variant: "destructive",

        title: "Warning",

        description: "No Internet Connection.",});});});

  useEffect(() => {

    if (connectionStatus === "Closed") {

      toast({variant: "destructive",

        description: "Error, Please wait...",});}},

[connectionStatus]);

The first `useEffect` hook listens for the "offline" event, which is triggered when the browser loses its connection to the internet. When this event is triggered, the code creates a toast notification with a "destructive" variant, a "Warning" title, and a "No Internet Connection" description.

The second `useEffect` hook runs when the `connectionStatus` value changes. If the `connectionStatus` value is "Closed", the code creates a toast notification with a "destructive" variant and a "Error, Please wait..." description.

useEffect(() => {

    setModuleConnectionStatus(readyState);}, [readyState]);

The `const connectionStatus` line is an object that maps the values of the `ReadyState` enumeration to human-readable strings. The code then uses the current value of the `readyState` variable to look up the corresponding human-readable string in the `connectionStatus` object.

useEffect(() => {

    return () => setModuleConnectionStatus(6);}, []);

this code is a React component returns the current connection status of a WebSocket.

**OtbTabView**

**–** OtbTabView.tsx

The code in OtbTabView.tsx is defining a React functional component called `OtbTabView` and exporting it as the default export.

Editable columns:

const editableColumns = [

    "Budget%",

    "BudgetvsACT\_FCT",

    "BudgetvsLY%",

    "BudgetvsLLY%",

    "BudgetQTY",

    "Logistic%",

    "ProposedSellThru%",

    "DisplayItem",

    "COR\_valueENDOfLifeStock",

  ];

This code defines a constant variable called `editableColumns`. The variable is an array that contains a list of strings. Each string represents the name of a column that can be edited in a table or data grid.

const getColumns = (tabName: string) => {

    function commonColumns(): any {

      let \_commonColumn: any = [];

      webSocketData?.tabs[tabName]?.map((item: any, index: any) => {

        if (webSocketData?.columns?.includes(item)) {

          \_commonColumn = [...\_commonColumn, item];}});

      return \_commonColumn;}

    if (Object?.keys(webSocketData)?.length) {

      if (tabName === "BudgetValue") {return [{

            id: "Expandable",

    header: expandHistory?.length ? <CollapseComponent /> : null,

    cell: ExpandComponent,enableSorting: false,enableHiding: false,

    footer: "Total", },...(webSocketData?.tabs[tabName]?.length > 0

            ? commonColumns()?.map((item: any, index: any) => {

                if (editableColumns?.includes(item)) {

          return { header: modifyHeaders(item),

                    accessorKey: item,

                    cell: EditableTableCell,

                    enableHiding: !mandatoryColumns?.includes(item),

                    footer: webSocketData?.total[item]

                      ? Number(webSocketData?.total[item])?.toLocaleString(

                          "en-US",{}): null,};

        } else { return {

                    header: modifyHeaders(item),

                    accessorKey: item,

                    cell: NormalCell,

                    enableHiding: !mandatoryColumns?.includes(item),

                    footer: webSocketData?.total[item]

                      ? Number(webSocketData?.total[item])?.toLocaleString(

                          "en-US", {} ) : null,

                    filterFn: (row: any, id: any, value: any) => {

             return value.includes(String(row.getValue(id)))},}}}) : []),];

 } else {return commonColumns()?.map((item: any, index: any) => {return {

            header: modifyHeaders(item),

            accessorKey: item,

            cell: NormalCell,footer: webSocketData?.total[item]? Number (webSocketData?.total[item])?.toLocaleString("en-US", {}) : null,

    filterFn: (row: any, id: any, value: any) => {

 return value.includes(String(row.getValue(id)));},};});}} else return [];};

commonColumns() Function:

This inner function is defined to extract common columns from the webSocketData associated with the given tabName. It iterates through the items in the webSocketData and checks if each item is included in the webSocketData.columns array. If it is, the item is added to the \_commonColumn array.

Main Function Logic:

The main getColumns function begins by checking if the webSocketData object has keys (i.e., if it's not empty). If it is empty, an empty array is returned.

If the tabName is "BudgetValue," it returns a specific array of column configurations:

It includes an expandable column with an optional header based on the expandHistory length.

It then processes the common columns obtained using the commonColumns function.

For each common column, it checks if it's in the editableColumns. If yes, it creates a configuration for an editable cell (EditableTableCell) with specific headers, accessor keys, and footers.

If the column is not in editableColumns, it configures it for a normal cell (NormalCell) with headers, accessor keys, and footers.

Additionally, it sets up a filter function for each column to filter rows based on the provided value.

If the tabName is not "BudgetValue," it simply returns an array of column configurations for the common columns, following a similar structure as described above.

Overall, this code is used to dynamically generate an array of column configurations based on the specified tabName and the data stored in the webSocketData object. The resulting array can then be used in a data table or grid component to define how data is displayed and interacted with in the user interface.

**OtbSideDrawer**

**-** OtbSideDrawer.tsx

React component called `OtbSideDrawer`. The component uses the `Sheet`, `SheetContent`, `SheetFooter`, `SheetHeader`, `SheetTitle`, and `SheetTrigger` components from the `@/src/components/shared/sheet` module to create a side drawer that can be opened and closed.

export function OtbSideDrawer() {

  return (<>

<Sheet>

        <SheetTrigger asChild>

          <Button className={"whitespace-nowrap m-2"} value={"OTB"}>

            Filters

          </Button>

        </SheetTrigger>

        <SheetContent side={"left"}>

          <SheetHeader className="mb-2">

            <SheetTitle>OTB Filters</SheetTitle>

          </SheetHeader>

          <OtbFilterForm />

          <SheetFooter className="justify-start">

            <small>

              After submitting this form, OTB Table will be populated.

            </small>

          </SheetFooter>

        </SheetContent>

      </Sheet> </>);}

The `SheetTrigger` component is used to define the button that opens the side drawer. The `Button` component is used to create the button, and its `value` prop is set to the string `"OTB"`.

The `SheetContent` component is used to define the content of the side drawer. The `side` prop of the `SheetContent` component is set to the string `"left"`, which indicates that the side drawer will open from the left side of the screen.

The `SheetHeader` component is used to define the header of the side drawer. The `SheetTitle` component is used to set the title of the side drawer to the string `"OTB Filters"`.

The `OtbFilterForm` component is used to define the content of the side drawer. This component is responsible for rendering the form that allows users to filter the OTB table.

The `SheetFooter` component is used to define the footer of the side drawer. The `justify-start` class is added to the `SheetFooter` component to align the text in the footer to the left side of the container.

The `small` element is used to display a small piece of text in the footer of the side drawer. This text informs the user that after submitting the form, the OTB table will be populated.

**OtbForm**

-OtbForm.tsx

The component returns form includes filters for sales channel, product family, product sub-family, brand/supplier, product category, product sub-category, item code, top items, and store class. The filters are implemented using the MainFilters and HierarchyFilter components, which are custom components that render the appropriate filter options based on the provided filter type.

const formSchema = z.object({

  history\_dates: z.string(),

  forecast\_dates: z.string(),

});

This code defines a form schema using the Zod library. The form schema is an object with two properties: history\_dates and forecast\_dates. Both properties are strings.

The Zod library is a schema validation library for JavaScript and TypeScript. It allows you to define the shape of your data and validate it against that shape.

In this code, the form schema is defined as an object with two properties: history\_dates and forecast\_dates. Both properties are strings, which means that they can only contain text.

The form schema is then assigned to the variable formSchema. This variable can be used to validate form data against the schema.

 const form = useForm<z.infer<typeof formSchema>>({

    resolver: zodResolver(formSchema),

    defaultValues: {

      history\_dates: "",

      forecast\_dates: "",},});

This code is using the React Hook Form library to create a form with validation using the Zod library.

1. `const form = useForm<z.infer<typeof formSchema>>({...})`: This line is using the `useForm` hook from React Hook Form to create a form with a specific type. The type is inferred from the `formSchema` variable, which is a Zod schema.

2. `resolver: zodResolver(formSchema)`: This line is setting the resolver for the form to a Zod resolver. The Zod resolver is a function that takes a Zod schema and returns a resolver function that can be used to validate form data against that schema.

3. `defaultValues: {...}`: This line is setting the default values for the form. In this case, the default values are an empty string for both the `history\_dates` and `forecast\_dates` fields.

function handleSubmit() {

    setExpandHistory([]);

  const historyAndForecastData = {

    history\_date\_range: {

      fro: history\_date?.from? format(history\_date?.from, "yyyy-MM-dd"): "",

      to: history\_date?.to? format(history\_date?.to, "yyyy-MM-dd") : "",},

    forecast\_date\_range: {

      fro: forecast\_date?.from

        ? format(forecast\_date?.from, "yyyy-MM-dd"): "",

   to: forecast\_date?.to ? format(forecast\_date?.to, "yyyy-MM-dd") : "",},};

    const otherFilters: any = {

      sales\_channel: mainFilterValues?.channel ?? [],

      product\_family: mainFilterValues?.family ?? [],

      sub\_families: mainFilterValues?.sub\_family ?? [],

      category: mainFilterValues?.category ?? [],

      sub\_category: mainFilterValues?.sub\_category ?? [],

      suppliers: mainFilterValues?.suppliers ?? [],

      sku: mainFilterValues?.sku ?? [],

      top\_items: mainFilterValues?.top\_items ?? [],

      store\_class: mainFilterValues?.class ?? [],};

    const group\_by = Object?.keys(otherFilters)?.filter((item, index) => {

      return otherFilters[item]?.length > 0;});

    const additionalData = {

      page\_number: 0,

      page\_size: 10,

      fetch\_from\_db: true,

      group\_by: { status: true, columns: group\_by ?? [] },

      expand: { status: false, row: {} },

      sort: {},

      table\_changes: {}, };

    const data = {

      ...historyAndForecastData,

      ...otherFilters,

      ...additionalData, };

    if (JSON?.stringify(body) !== JSON?.stringify(data)) {

      setBody(data);

      setLoading(true);}

    toast({

      title: "You submitted the following values:",

      description: (

        <pre className="mt-2 w-[340px] rounded-md bg-slate-950 p-4">

          <code className="text-white">

            {JSON.stringify(

              { ...historyAndForecastData, ...otherFilters },null,2)}

          </code>

        </pre>),});}

1. The `handleSubmit()` function is called when the form is submitted.

2. The function first sets the `expandHistory` state to an empty array.

3. The function then creates an object called `historyAndForecastData` that contains the `history\_date\_range` and `forecast\_date\_range` properties. These properties are objects that contain the `fro` and `to` properties, which are the formatted dates from the `history\_date` and `forecast\_date` objects.

4. The function then creates an object called `otherFilters` that contains the properties for the main filter values.

5. The function then creates an array called `group\_by` that contains the keys of the `otherFilters` object that have a length greater than 0.

6. The function then creates an object called `additionalData` that contains the properties for the page number, page size, fetch from database, group by, expand, sort, and table changes.

7. The function then creates an object called `data` that is a combination of the `historyAndForecastData`, `otherFilters`, and `additionalData` objects.

8. The function then checks if the `body` state is not equal to the `data` object. If it is not equal, the function sets the `body` state to the `data` object and sets the `loading` state to true.

9. Finally, the function creates a toast notification that displays the submitted values. The toast notification contains a title and a description that displays the submitted values in a formatted JSON object.

**HierarchyFilter**

-hierarchyFilter.tsx

hierarchyFilter.tsx is component created in mainFilter folder. This code is used to filter data based on a hierarchical structure. The component takes in three props: title, filter, and parent. The title prop is used to display the title of the filter, the filter prop is used to identify the filter, and the parent prop is used to identify the parent filter.

export default function HierarchyFilter({ title, filter, parent }: FilterProp) {const {

    mainFilterOptions,

    setMainFilterOptions,

    mainFilterValues,

    setMainFilterValues,

  } = useContext(OtbContext);

  useEffect(() => {

    setMainFilterValues((s: any) => ({ ...s, [filter]: [] }));

    if (mainFilterValues[parent]?.length > 0) {

     getHierarchicalFilters({

      filterName: `${parent?.toLowerCase()}2${filter?.toLocaleLowerCase()}`,

       filterData: mainFilterValues[parent] || [],})

        .then((response) => {

          setMainFilterOptions((s: any) => ({...s,

            [filter]: response?.filter\_values,}));})

        .catch((err) => console.log({ err }));} else {

      setMainFilterOptions((s: any) => ({ ...s, [filter]: [] }));}

  }, [mainFilterValues[parent]]);

  function handleChange(e: any) {

    setMainFilterValues((s: any) => ({ ...s, [filter]: e }));}

The useEffect hook is used to fetch the available values for the current filter when the parent filter value changes. The getHierarchicalFilters function is used to fetch the available values for the current filter. The function takes in the name of the filter and the selected values of the parent filter, and returns the available values for the current filter.

The handleChange function is used to handle changes to the selected values of the current filter. When the user selects a value, the handleChange function updates the mainFilterValues state variable.

The component returns a MultiSelectDD component, which is used to display the available values for the current filter and allow the user to select values.

**MainFilter**

-mainFilter.tsx

mainFilter.tsx is another component created in mainFilter folder that allows users to select values for a filter. The component takes in two props: title and filter.

export default function MainFilters({ title, filter }: FilterProp) {

  const {

    mainFilterOptions,

    setMainFilterOptions,

    mainFilterValues,

    setMainFilterValues,

  } = useContext(OtbContext);

  useEffect(() => {

    if (!mainFilterOptions[filter]) {

      getMasterFilters([filter])

        .then((resp: any) => {

          setMainFilterOptions((s: any) => ({...s,

            [filter]: resp?.filter\_values[filter],}));})

        .catch((err) => console.log({ err }));}

  }, [mainFilterOptions[filter]]);

  function handleChange(e: any) {

    setMainFilterValues((s: any) => ({ ...s, [filter]: e }));}

The component state is managed using the useContext hook, which allows the component to access the mainFilterOptions, setMainFilterOptions, mainFilterValues, and setMainFilterValues state variables from the OtbContext.

The useEffect hook is used to fetch the available values for the filter if they are not already available in the mainFilterOptions state variable. The getMasterFilters function is used to fetch the available values for the filter from the server.

The handleChange function is used to handle changes to the selected values of the filter. When the user selects a value, the handleChange function updates the mainFilterValues state variable.

The component returns a MultiSelectDD component, which is used to display the available values for the filter and allow the user to select values.

**OtbTopDrawer**

-OtbTopDrawer.tsx

This is another component in otb\_graph folder which has path to OTB graph, given below is the sample code;

export function OtbTopDrawer() {

  return (

    <>

      <Sheet>

        <SheetTrigger asChild>

          <Button className=" border-2 mr-2 bg-white">

            <svg width="15" height="15" viewBox="0 0 15 15"

              fill="none"

              xmlns="http://www.w3.org/2000/svg"  >

   <path d="M11.5 1C11.7761 1 12 1.22386 12 1.5V13.5C12 13.7761 …"

                fill="#000000"

                fill-rule="evenodd"

                clip-rule="evenodd"></path>

            </svg>

          </Button>

        </SheetTrigger>

        <SheetContent side={"top"}>

          <SheetHeader className="mb-2">

            <SheetTitle>OTB Chart</SheetTitle>

          </SheetHeader>

          <div className=" w-96 h-96 ">

            <OtbGraph />

          </div>

        </SheetContent>

      </Sheet></>);}

The component has three main parts:

- The `Sheet` component, which is used to create the container for the chart.

- The `SheetTrigger` component, which is used to display the button that triggers the opening of the chart.

- The `SheetContent` component, which is used to display the chart itself.

The `Sheet` component has a `side` prop, which is used to specify the side of the screen on which the chart will be displayed. In this case, the chart will be displayed on the top of the screen.

The `SheetTrigger` component has a `className` prop, which is used to specify the CSS class that will be applied to the button. In this case, the button will have the `border-2 mr-2 bg-white` CSS class applied to it.

The `SheetContent` component has a `className` prop, which is used to specify the CSS class that will be applied to the chart. In this case, the chart will have the `w-96 h-96` CSS class applied to it.

The `OtbGraph` component is used to display the chart itself. This component is not included in the code you provided, but it is likely that it is a custom component that is used to display the chart data.

**OTB Table Folder:**

**OtbDataTable**

**-** OtbDataTable.tsx

OtbDataTable is component in OTB table folder which exports function OtbDataTable() with destructured props columns, data, setData and loading coming form otb controller parent component,

 const { body, setBody } = React.useContext(OtbContext);

const [rowSelection, setRowSelection] = React.useState({});

const [columnVisibility, setColumnVisibility] =

React.useState<VisibilityState>(nonDefaultColumns);

const [columnFilters, setColumnFilters] = React.useState<ColumnFiltersState>([]);

const [sorting, setSorting] = React.useState<SortingState>([]);

this code is used to handle states and display a table of data with a search bar and pagination. The component is highly customizable and can be used to display a variety of different types of data.

  const table = useReactTable({

    data,

    columns,

    state: {

      sorting,

      columnVisibility,

      rowSelection,

      columnFilters,

    },

    meta: {

      updateData: (rowIndex: number, columnId: string, value: string) => {

        setData((old: any) =>

          old.map((row: any, index: any) => {

            if (index === rowIndex) {

              return {

                ...old[rowIndex], [columnId]: value,

              };} return row; }));},},

const table = useReactTable({ ... }): This line initializes a React Table instance using the useReactTable hook. It takes an object with various configuration options as its argument.

data: This is likely an array of data that you want to display in your table. It represents the rows of your table.

columns: This is an array of column definitions for your table. Each object in the columns array likely defines properties like Header (the column header title) and accessor (a function that specifies how to access data for that column from each row).

state: This property holds the initial state for your table. It includes settings like sorting, column visibility, row selection, and column filters. These settings control the initial behaviour and appearance of your table.

meta: This property is an object containing metadata and callbacks related to your table.

updateData: This is a callback function that is likely used to update the data in your table. It appears to update a specific cell's value at a given row and column. It takes three parameters:

rowIndex: The index of the row you want to update.

columnId: The identifier of the column you want to update.

value: The new value to set for the specified cell.

Inside the updateData function, it seems to update the data state using the setData function, which is not shown in your provided code but is likely a React useState hook to manage the table's data. It uses the map function to iterate over the existing data array and create a new array with the updated value for the specified cell.

This code represents the setup for your React Table component, and it appears to be part of a larger component or application. You would typically render your table UI and handle user interactions using this configuration. If you have more specific questions or need further assistance with this code, please provide additional context or specific questions.

 useEffect(() => {

    const \_sort = JSON?.stringify(table?.getState()?.sorting[0] ?? {});

    const \_body\_sort = JSON?.stringify(body?.sort);

    if (!(\_sort === \_body\_sort) && Object?.keys(body)?.length) {

      setBody((s: any) => ({...s,

        fetch\_from\_db: false,

        table\_changes: {},

        sort: table?.getState()?.sorting[0] ?? {},

      })); }}, [table?.getState().sorting]);

This code is a React useEffect that monitors changes to the sorting state of a table component and updates the body state accordingly;

useEffect(() => { ... }, [table?.getState().sorting]);:

This useEffect hook is triggered whenever there's a change in the sorting state of the table object. It depends on the table?.getState().sorting value.

const \_sort = JSON?.stringify(table?.getState()?.sorting[0] ?? {});:

Here, \_sort is a variable that stores a JSON string representation of the first sorting configuration in the table component's state. If there are no sorting configurations, it defaults to an empty object {}.

const \_body\_sort = JSON?.stringify(body?.sort);:

This line creates a JSON string representation of the sort property in the body object. The body object is assumed to contain some state related to the table.

if (!(\_sort === \_body\_sort) && Object?.keys(body)?.length):

This condition checks if \_sort (the current sorting configuration in the table) is not equal to \_body\_sort (the sorting configuration stored in the body state), and if the body object has at least one key (i.e., it's not empty).

Inside the condition:

It updates the body state by creating a new object ({ ...s } where s is the current state).

It sets fetch\_from\_db to false, indicating that there's no need to fetch data from a database.

It resets the table\_changes property to an empty object.

It updates the sort property in the body state with the current sorting configuration from the table component.

**OtbTablePagination**

-OtbTablePagination.tsx

It uses the useContext hook to access data from the OtbContext, which likely contains state information related to the data table and its pagination.

  const { body, setBody, webSocketData } = useContext(OtbContext);

  const totalElements = webSocketData?.data?.length ? webSocketData?.items:0;

  const totalPage = Math.ceil(totalElements / body?.page\_size);

This code calculates the totalElements, representing the total number of data elements available from the webSocketData. If there is no data available, it defaults to 0.

It calculates the totalPage by dividing the totalElements by the current number of rows per page specified in body.page\_size. This value represents the total number of pages needed to display all the data elements based on the chosen page size.

In summary, this code snippet retrieves data from the context, calculates the total number of elements and pages based on that data, and stores these values for use in the pagination controls displayed by the DataTablePagination component.

The code generates a user interface for paginating data in a table, including options to change the number of rows per page and navigate through different pages of data. The behavior of these controls is tied to the body state and context provided by the OtbContext.

**NonDefaultColumns**

**-** nonDefaultColumns.ts

export const nonDefaultColumns = {…};

this type script component is used to define the header values that need to be hidden while initial loading of the table. The values will be header names as key and will be defined as false.

**ExpandableComponent**

- ExpandableComponent.tsx

The component used for expanding rows in a table or grid view. When the button is clicked, it updates the context state to keep track of expanded rows and makes changes to the setBody state to enable row expansion. The conditional rendering of the "plus" icon suggests that row expansion might be optional for some rows based on the presence of the "ItemLookupCode" property in the data.

export const ExpandComponent = ({ getValue, row, column, table }: any) => {

  const { setBody, expandHistory, setExpandHistory } = useContext(OtbContext);

  function handleExpand() {

    setExpandHistory((s: any) => [...s, { ...row?.original }]);

    setBody((s: any) => ({ ...s,

      table\_changes: {}, fetch\_from\_db: false,

      group\_by: { ...s?.group\_by, status: false },

      expand: { status: true, row: { ...row?.original } },}));}

  return (

    <button

      className="  pl-4 w-full flex items-center justify-center"

      onClick={handleExpand} >

      {Object?.keys(row?.original).includes("ItemLookupCode") ? null : (

        <PlusCircledIcon className="mr-2 h-4 w-4" />)}

    </button>); };

component receives some props as parameters: getValue, row, column, and table. These props are likely used to access data and context within the component.

It uses the useContext hook to access values from a context called OtbContext. This context likely contains some shared state or functions that are used within this component.

Inside the handleExpand function, it updates the state stored in the OtbContext. Specifically, it adds the current row.original to the expandHistory array and updates the setBody state with some changes.

It adds a new item to the expandHistory array, containing a copy of the row.original data.

It updates the setBody state with various changes, such as resetting the table\_changes, setting fetch\_from\_db to false, updating group\_by with { status: false }, and enabling row expansion by setting expand with { status: true, row: { ...row?.original } }.

It renders a button with a click event handler that calls the handleExpand function when clicked.

Within the button's content, it conditionally renders an icon (likely a "plus" icon) using the PlusCircledIcon component. The icon is displayed if the row.original object does not have a property named "ItemLookupCode."

**EditableCell**

- EditableCell.tsx

The components imports the necessary dependencies from React, including useContext, useEffect, and useState.

export const EditableTableCell = ({ getValue, row, column, table }: any) => {

  const initialValue = getValue();

  const [value, setValue] = useState(initialValue);

  const { body, setBody, transformedData, webSocketData } = useContext(OtbContext);

  useEffect(() => { setValue(initialValue); }, [initialValue]);

  const onBlur = () => {

    table.options.meta?.updateData( row.index, column.id,

      transformedData[row.index][column.id]);};

  const onKeyDown = (e: any): any => {

    if (e.keyCode === 13 &&

      Number(transformedData[row.index][column.id]) !== Number(value)) {

      table.options.meta?.updateData(row.index, column.id, value);

      setBody({...body, fetch\_from\_db: false, table\_changes: {

          row: row?.original, columnId: column.id, newValue: Number(value),

        },});}};

It imports the OtbContext from an external module, presumably for accessing some shared context data.

Inside the component function, it initializes the initialValue state by calling the getValue function passed as a prop. This is the initial value displayed in the input field.

It uses the useState hook to create a value state, which is initialized with initialValue. This state represents the current value in the input field and can change as the user interacts with it.

It uses the useContext hook to access context data from OtbContext. It destructures properties such as body, setBody, transformedData, and webSocketData from the context.

There's an useEffect hook that sets the value state to initialValue whenever initialValue changes. This effect ensures that the input field's value stays in sync with the initialValue.

The onBlur function is called when the input field loses focus (i.e., when the user clicks outside of it). It appears to update the data using the updateData function from table.options.meta based on the row.index and column.id, using transformedData to provide the updated value.

The onKeyDown function is called when a key is pressed while the input field is focused. It checks if the Enter key (keyCode 13) is pressed and if the value has changed. If so, it updates the data in a similar manner to the onBlur function and sets the body context data accordingly.

Finally, the component renders an <input> element with the current value. It listens to changes using the onChange event to update the value state when the user types in a new value. It also handles keydown events using the onKeyDown function and onBlur events using the onBlur function.

**CollapseComponent**

- CollapseComponent.tsx

export default function CollapseComponent() {

  const { setBody, expandHistory, setExpandHistory } = useContext(OtbContext);

Inside the component, it uses the useContext hook to access specific values and functions from the OtbContext context. These values/functions include setBody, expandHistory, and setExpandHistory.

function handleCollapse() {

    if (expandHistory?.length === 1) {

      setBody((s: any) => ({ ...s,

        fetch\_from\_db: false, table\_changes: {},

        expand: { status: false, row: {},},

        group\_by: { ...s?.group\_by, status: true },

        page\_number: 0,

        page\_size: 10,})); setExpandHistory([]); } else {

      setBody((s: any) => ({ ...s,

        table\_changes: {}, fetch\_from\_db: false,

        expand: { status: true,

          row: { ...expandHistory[expandHistory?.length - 2] },},}));

      setExpandHistory((s: any) => {

        let \_arr = [...s]; \_arr.splice(s.length - 1, 1);

        return \_arr;});}}

The handleCollapse function is defined and serves as the click event handler for the button.

If expandHistory has a length of 1 (meaning there's only one history item), it sets the setBody and setExpandHistory functions to update the state as follows:

fetch\_from\_db is set to false.

table\_changes is reset to an empty object.

expand status is set to false, and row is an empty object.

group\_by status is set to true.

page\_number is set to 0.

page\_size is set to 10.

setExpandHistory is called with an empty array to clear the expand history.

If expandHistory has a length greater than 1, it sets the setBody and setExpandHistory functions to update the state as follows:

table\_changes is reset to an empty object.

fetch\_from\_db is set to false.

expand status is set to true, and row is populated with the previous history item.

The last item from expandHistory is removed by splicing the array.

This component allows you to collapse or expand content and manage the history of those actions using the provided context and state management functions.

**Data table view options**

- data-table-view-options.tsx

export function DataTableViewOptions<TData>({

  table,

}: DataTableViewOptionsProps<TData>) {

  return (

    <DropdownMenu>

      <DropdownMenuTrigger asChild>

        <Button variant="outline" size="sm"

          className="ml-auto hidden h-8 lg:flex" >

          <MixerHorizontalIcon className="mr-2 h-4 w-4" />View</Button>

     </DropdownMenuTrigger>

     <DropdownMenuContent align="end">

      <DropdownMenuLabel>Toggle columns</DropdownMenuLabel>

       <DropdownMenuSeparator />

         {table.getAllColumns().filter((column) =>

          typeof column.accessorFn !== "undefined" && column.getCanHide())

            .map((column) => {return (

             <DropdownMenuCheckboxItem

              key={column.id} className="capitalize"

              checked={column.getIsVisible()}

            onCheckedChange={(value) => column.toggleVisibility(!!value)}>

               {column.id}

          </DropdownMenuCheckboxItem>)})}

      </DropdownMenuContent> </DropdownMenu> )}

Defining the DataTableViewOptions component:

DataTableViewOptions is a functional React component that takes a table prop of type Table<TData>. It's used to configure the options for viewing data in the table.

Render method:

The component renders a DropdownMenu from Radix UI, which creates a dropdown menu.

Inside the DropdownMenu, there is a DropdownMenuTrigger that wraps a Button component. This button will trigger the dropdown menu when clicked.

The Button component includes a "View" label and an icon (MixerHorizontalIcon).

The dropdown menu content is wrapped in a DropdownMenuContent component. It is aligned to the end (right) and has a maximum height with scrollable content.

Inside the DropdownMenuContent, there are:

A DropdownMenuLabel displaying "Toggle columns".

A DropdownMenuSeparator for visual separation.

A list of checkboxes generated from the table's columns. These checkboxes allow the user to toggle the visibility of columns in the table.

Each checkbox corresponds to a column and is labeled with the column's ID.

The checked prop of each checkbox depends on whether the column is currently visible.

The onCheckedChange callback is used to toggle the visibility of the column when the checkbox is clicked.

Overall, this component provides a user interface element for customizing the visibility of columns in a table. When the "View" button is clicked, a dropdown menu with checkboxes for each column is displayed, allowing the user to control which columns are shown in the table.

**Data table toolbar**

-data-table-toolbar.tsx

export function DataTableToolbar({ table }: any) {

  const { body, setBody } = useContext(OtbContext);

  const [historicalYear, setHistoricalYear] = useState([]);

  const [budgetYear, setBudgetYear] = useState([]);

  function getYearRange(startDate: any, endDate: any){let \_arr: any = [];

    if (startDate && endDate) { for (

        let i = new Date(startDate).getFullYear();

  i <= new Date(endDate).getFullYear(); i++){\_arr=[...\_arr, String(i)];}}

  return \_arr;}

  useEffect(() => { setHistoricalYear(

 getYearRange(body?.history\_date\_range?.fro,body?.history\_date\_range?.to));

    setBudgetYear(

      getYearRange(

        body?.forecast\_date\_range?.fro,

        body?.forecast\_date\_range?.to));}, [body]);

This component is responsible for rendering a toolbar that appears to be part of a data table interface. Here's a brief description of what this code does:

It imports necessary dependencies, including React hooks and context.

Inside the component function, it initializes two state variables, historicalYear and budgetYear, both of which are initially set to empty arrays.

There is a getYearRange function that takes two date parameters (startDate and endDate) and returns an array of year strings within that date range. It checks if both startDate and endDate are provided and then generates an array of years between them.

The useEffect hook is used to set the historicalYear and budgetYear state variables based on date ranges provided from the OtbContext when the body object changes. It calls the getYearRange function with appropriate date range properties from the body object and updates the state variables accordingly.

The component returns a JSX structure, which includes:

Two MultiSelectDD components for selecting "Budget Year" and "Historical Year". These dropdown components display options based on the budgetYear and historicalYear state variables and allow users to select one or more years. When a selection is made, it updates the body object in the context, specifically the budget\_year and historical\_year properties.

OtbSideDrawer and OtbTopDrawer components, which are not defined in this code snippet, but they seem to be components for displaying side and top drawers, possibly for additional functionality.

A DataTableViewOptions component, which is also not defined here, but it receives the table prop.

**Data table faceted filter**

- data-table-faceted-filter.tsx

The component creates a user-friendly filter interface with options that can be selected and deselected, and it allows users to search for specific filter options. It also provides feedback on the number of selected values and facets associated with each option.

export function DataTableFacetedFilter<TData, TValue>({

  column,title,options,}: DataTableFacetedFilter<TData, TValue>) {

  const facets = column?.getFacetedUniqueValues()

  const selectedValues = new Set(column?.getFilterValue() as string[])

the code initializes two variables:

facets assigned the unique filter values associated with the column (if available).

selectedValues is assigned a Set containing the selected filter values retrieved from the column. These selected values are assumed to be strings.