HOC,

Custom hooks

Redux

Middleware

How do we connect with DB?

The <header> element represents a container for introductory content or a set of navigational links.

A <header> element typically contains:

* one or more heading elements (<h1> - <h6>)
* logo or icon
* authorship information

**Redux Overview**

Redux is a state management library for JavaScript applications, commonly used with React. It helps manage global state by providing a single source of truth for state, making it easier to predict and debug your application's behavior.

**Key Concepts**

Store: The central location that holds the entire state of the application.

Actions: Payloads that trigger state changes, typically sent from components.

Reducers: Pure functions that take the current state and an action, and return a new state.

Dispatch: The function that sends actions to the store.

How it Works

Components dispatch actions to the store.

The store passes the action to the reducer.

The reducer returns a new state, which is stored in the store.

The store notifies components that the state has changed.

// actions.js

export const INCREMENT\_COUNTER = 'INCREMENT\_COUNTER';

export function incrementCounter() {

return { type: INCREMENT\_COUNTER };

}

// reducer.js

import { INCREMENT\_COUNTER } from './actions';

const initialState = { counter: 0 };

export default function counterReducer(state = initialState, action) {

switch (action.type) {

case INCREMENT\_COUNTER:

return { counter: state.counter + 1 };

default:

return state;

}

}

// store.js

import { createStore } from 'redux';

import counterReducer from './reducer';

const store = createStore(counterReducer);

// component.js

import React from 'react';

import { incrementCounter } from './actions';

import { connect } from 'react-redux';

const Counter = ({ counter, incrementCounter }) => (

<div>

<p>Counter: {counter}</p>

<button onClick={incrementCounter}>Increment</button>

</div>

);

const mapStateToProps = state => ({ counter: state.counter });

const mapDispatchToProps = { incrementCounter };

export default connect(mapStateToProps, mapDispatchToProps)(Counter);

**How do we connect with DB:**

Connecting to a Database in ReactJS

ReactJS is a front-end library, and it doesn't provide a direct way to connect to a database. However, you can use various libraries and tools to interact with a database from your React application. Here are a few common approaches:

1. RESTful API

Create a RESTful API using a server-side language like Node.js, Python, or Ruby. The API will handle database interactions, and your React application will make HTTP requests to the API to retrieve or send data.

Example

Create a Node.js server using Express.js

Use a library like Mongoose (for MongoDB) or Sequelize (for SQL databases) to interact with the database

Create API endpoints to handle CRUD (Create, Read, Update, Delete) operations

In your React application, use the fetch API or a library like Axios to make HTTP requests to the API endpoints

2. GraphQL API

Create a GraphQL API using a library like Apollo Server or GraphQL Yoga. GraphQL provides a more flexible and efficient way to interact with your database.

Example

Create a GraphQL schema using Apollo Server

Use a library like Prisma (for SQL databases) or Mongoose (for MongoDB) to interact with the database

Create resolvers to handle GraphQL queries and mutations

In your React application, use a library like Apollo Client to interact with the GraphQL API

3. Direct Database Connection

Use a library like pg (for PostgreSQL) or mysql (for MySQL) to connect directly to your database from your React application. This approach is not recommended, as it exposes your database credentials and can lead to security vulnerabilities.

Example

Use the pg library to connect to a PostgreSQL database

Use the mysql library to connect to a MySQL database

**Higher Order Components**

Higher-Order Components (HOCs)

A Higher-Order Component (HOC) is a function that takes a component as an argument and returns a new component with additional props or behavior. HOCs are a powerful tool in React for code reuse, abstraction, and separation of concerns.

Why use HOCs?

Code reuse: HOCs allow you to extract common logic or functionality from multiple components and reuse it in a single place.

Decoupling: HOCs help decouple components from each other, making it easier to test, maintain, and update individual components.

Abstraction: HOCs provide a way to abstract away complex logic or implementation details, making your code more readable and maintainable.

How to create a HOC

A HOC is a function that takes a component as an argument and returns a new component. The new component is typically created by wrapping the original component with a new component that adds the desired behavior or props.

// withLoadingIndicator.js

import React from 'react';

const withLoadingIndicator = (WrappedComponent) => {

const [isLoading, setIsLoading] = React.useState(false);

const handleLoading = (isLoading) => {

setIsLoading(isLoading);

};

return function EnhancedComponent({ ...props }) {

return (

<div>

{isLoading ? (

<p>Loading...</p>

) : (

<WrappedComponent {...props} handleLoading={handleLoading} />

)}

</div>

);

};

};

export default withLoadingIndicator;

How to use a HOC

To use a HOC, you need to wrap your component with the HOC function. This will return a new component with the additional behavior or props.

// MyComponent.js

import React from 'react';

import withLoadingIndicator from './withLoadingIndicator';

const MyComponent = ({ handleLoading }) => {

const fetchData = async () => {

handleLoading(true);

// Simulate data fetching

await new Promise((resolve) => setTimeout(resolve, 2000));

handleLoading(false);

};

return (

<div>

<button onClick={fetchData}>Fetch Data</button>

</div>

);

};

const EnhancedMyComponent = withLoadingIndicator(MyComponent);

export default EnhancedMyComponent;

**Common use cases for HOCs**

Authentication: Create a HOC that checks if a user is authenticated before rendering a component.

Authorization: Create a HOC that checks if a user has the required permissions to access a component.

Loading indicators: Create a HOC that displays a loading indicator while data is being fetched.

Error handling: Create a HOC that catches and handles errors in a component.

Logging: Create a HOC that logs important events or data in a component.

Best practices for HOCs

Keep HOCs simple: Avoid complex logic or implementation details in your HOCs.

Use meaningful names: Use descriptive names for your HOCs and the props they add.

Document your HOCs: Document the props and behavior added by your HOCs.

Test your HOCs: Thoroughly test your HOCs to ensure they work as expected.

**Middleware:**

Redux Thunk Middleware

The project uses the redux-thunk middleware to handle asynchronous actions. redux-thunk is a popular middleware for Redux that allows you to write action creators that return a function instead of an object.

In the

src/store/index.tsx

file, you can see that the redux-thunk middleware is added to the Redux store:

**import thunk from 'redux-thunk';**

**const store = configureStore({**

**reducer: persistedReducer,**

**middleware: [thunk],**

**});**

This allows you to write action creators that return a function, which can be used to handle asynchronous actions.

Which Redux middleware is best?

Redux Thunk: Redux Thunk is one of the most popular middleware libraries for Redux. It allows users to write action creators that return functions instead of plain objects, enabling asynchronous logic such as API requests and side effects.

Redux Persist Middleware

The project also uses the redux-persist middleware to persist the Redux store. redux-persist is a middleware that allows you to persist the Redux store to a storage engine, such as local storage or a database.

In the

src/store/index.tsx

file, you can see that the redux-persist middleware is used to persist the Redux store:

**import { persistReducer, persistStore } from 'redux-persist';**

**import storage from 'redux-persist/lib/storage';**

**const persistConfig = { key: 'root', whitelist: ['branchInfo', 'counter'], storage };**

**const persistedReducer = persistReducer(persistConfig, reducer);**

**const store = configureStore({**

**reducer: persistedReducer,**

**middleware: [thunk],**

**});**

**const persistor = persistStore(store);**

This allows you to persist the Redux store to local storage, so that the state is preserved even when the user closes the browser or refreshes the page.

Overall, the middleware is used in the project to handle asynchronous actions and to persist the Redux store, which allows for a more robust and scalable application.

**What is Redux persist for?**

Redux Persist is a state management tool that allows the state in a Redux store to persist across browser and app sessions, improving user experience by pre-loading the store with persistent data.

In React, a middleware is a function that has access to the dispatch and getState methods of the Redux store. It is used to modify or extend the behavior of the Redux store.

Here are some common use cases for middleware in React:

Logging: Middleware can be used to log actions and state changes for debugging purposes.

Error handling: Middleware can catch and handle errors that occur during the dispatching of actions.

Authentication: Middleware can check if a user is authenticated before allowing certain actions to be dispatched.

API calls: Middleware can be used to make API calls and dispatch actions based on the response.

Caching: Middleware can cache the results of expensive function calls and return the cached result instead of recalculating it.

Some examples of popular middleware in React include:

redux-thunk: allows dispatching functions as actions

redux-promise: allows dispatching promises as actions

redux-logger: logs actions and state changes

Here is an example of a simple logging middleware:

const loggerMiddleware = store => next => action => {

console.log('Action:', action);

console.log('State:', store.getState());

return next(action);

};

This middleware logs the action and state to the console every time an action is dispatched.

You can apply middleware to your Redux store using the applyMiddleware function:

import { createStore, applyMiddleware } from 'redux';

import loggerMiddleware from './loggerMiddleware';

const store = createStore(reducer, applyMiddleware(loggerMiddleware));

--------------------------------------------------------------------------------------------

1. What is the difference between function and arrow function in JavaScript?

AI Overview

In JavaScript, the core difference between regular functions and arrow functions lies in their this binding, syntax, and how they handle arguments: arrow functions lexically bind this (from the surrounding context) and have a more concise syntax, while regular functions have dynamic this binding and can access an arguments object.

Here's a more detailed breakdown:

1. this Binding:

* **Regular Functions:**

Have their own this context, which is determined dynamically based on how the function is called (e.g., by call(), apply(), or implicitly by the calling context).

* **Arrow Functions:**

Lexically bind this to the surrounding scope (the context where the arrow function is defined), meaning they inherit this from the enclosing function or global scope.

2. Syntax:

* **Regular Functions:**

Defined using the function keyword, followed by the function name (if it's a function declaration), parentheses for parameters, and curly braces for the function body.

* **Arrow Functions:**

Use the => syntax (arrow operator) to define the function. They can be concise, especially for simple expressions, allowing for implicit return if the body is a single expression.

3. Arguments Object:

* **Regular Functions:**

Have access to the arguments object, which is an array-like object containing all the arguments passed to the function.

* **Arrow Functions:**

Do not have a direct arguments object. To access arguments, you can use the rest parameter syntax (...args).

4. Use Cases:

* **Regular Functions:**

Better suited for methods, constructors, or when you need dynamic this binding.

* **Arrow Functions:**

Ideal for short, simple functions, event handlers, or when you want to ensure this is bound lexically to the surrounding context.

Example:

JavaScript

*// Regular Function*  
function myRegularFunction() {  
 console.log(this); *// Will log the context in which the function is called*  
}  
  
*// Arrow Function*  
const myArrowFunction = () => {  
 console.log(this); *// Will log the context where the arrow function is defined*  
};

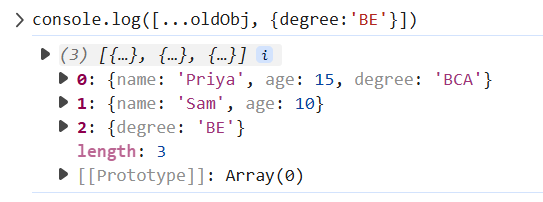
1. What is () => in JavaScript?

It is used to create a new type of function called an arrow function. Arrow functions have a simpler and more concise syntax than traditional function expressions. You can use them to define anonymous functions or to pass functions as arguments to other functions.

**Simple Program: To add a new object to an array of object**

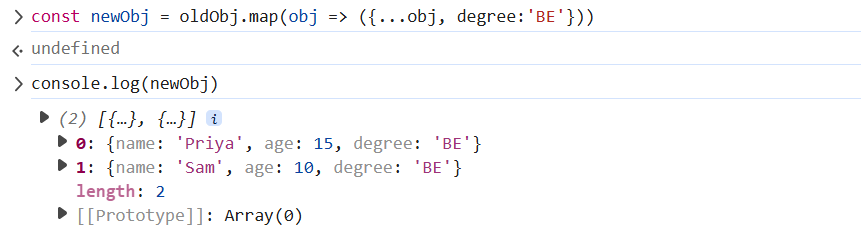
const oldObj =[{name: 'Priya', age:15, degree: 'BCA'},{name:'Sam', age: 10}]

console.log([...oldObj, {degree:'BE'}])

Output: 

**To add an attribute to each array element**





There are several ways to iterate through each property of an object in JavaScript.

1. **Using for...in loop**

const obj = { name: 'John', age: 30, city: 'New York' };

for (const prop in obj) {

console.log(prop, obj[prop]);

}

O/p:

name John

age 30

city New York

1. **Using Object.keys() and forEach()**

const obj = { name: 'John', age: 30, city: 'New York' };

Object.keys(obj).forEach((prop) => {

console.log(prop, obj[prop]);

});

1. **Using Object.entries() and forEach()**

const obj = { name: 'John', age: 30, city: 'New York' };

Object.entries(obj).forEach(([prop, value]) => {

console.log(prop, value);

});

1. **Using for...of loop with Object.entries()**

const obj = { name: 'John', age: 30, city: 'New York' };

for (const [prop, value] of Object.entries(obj)) {

console.log(prop, value);

}

This will output the same as above.

Note that the **for...in** loop will iterate over all enumerable properties, including inherited ones. If you only want to iterate over the object's own properties, use **Object.keys()** or **Object.entries()** instead.

**Hooks:**

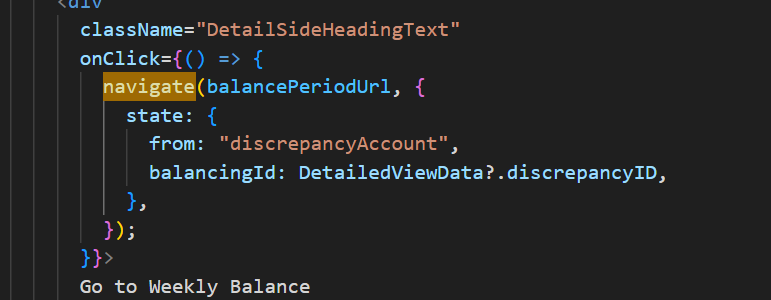
useParams <- to get the dynamic params from the url.

Example:

Example: const { accountingLocationID } = useParams();

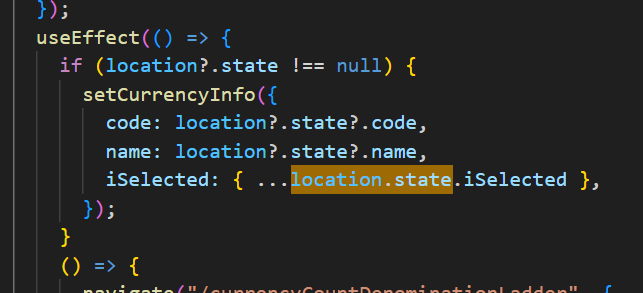
useNavigate <- to navigate to any url , also can send the state. To navigate to the previous screen

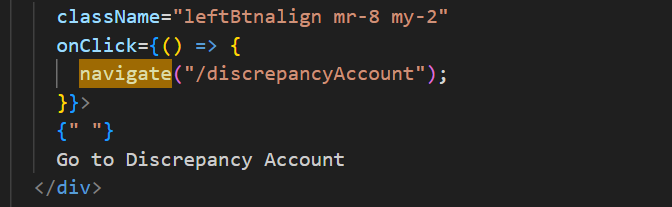


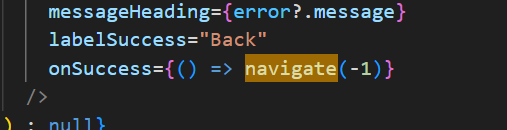


**To access the state passed in the useNavigate(): Need to use the useLocation**

const location = useLocation();







**What is the difference between async await and Promise chain?**

“async/await” executes operations sequentially, waiting for each operation to finish before proceeding. Promise chaining allows multiple independent operations to run concurrently if not interdependent

*"async and await make promises easier to write"*

**async** makes a function return a Promise

**await** makes a function wait for a Promise

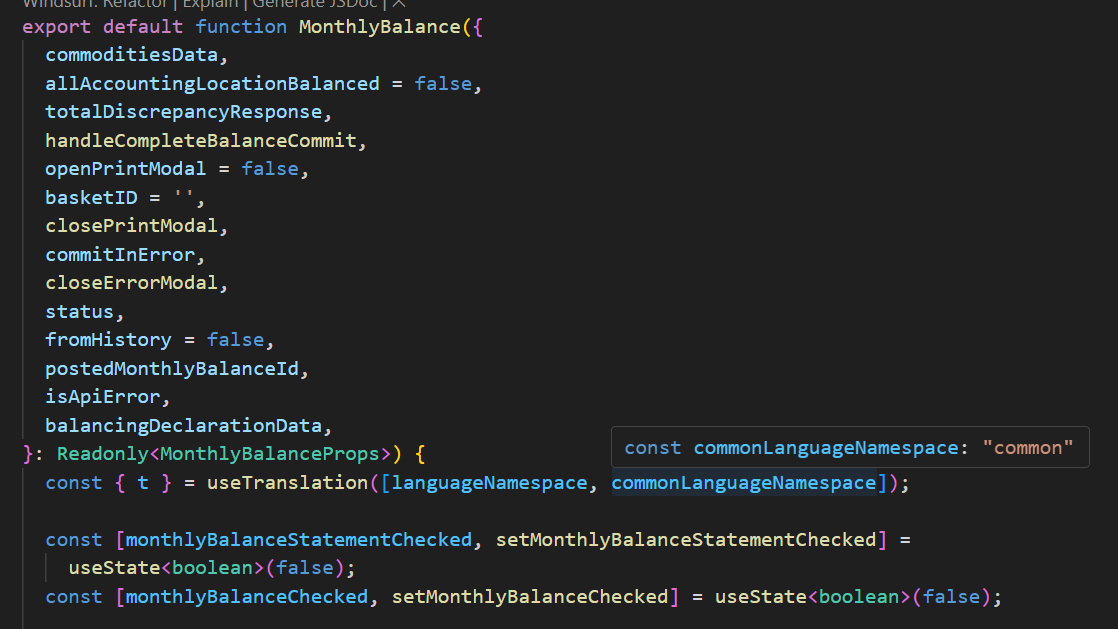
**Typescript:**

const balanceNumbers: Record<string, number> = {

  TP: 0,

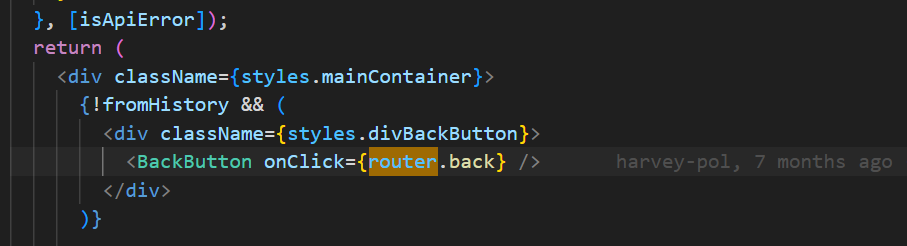
  BP: 0,

};

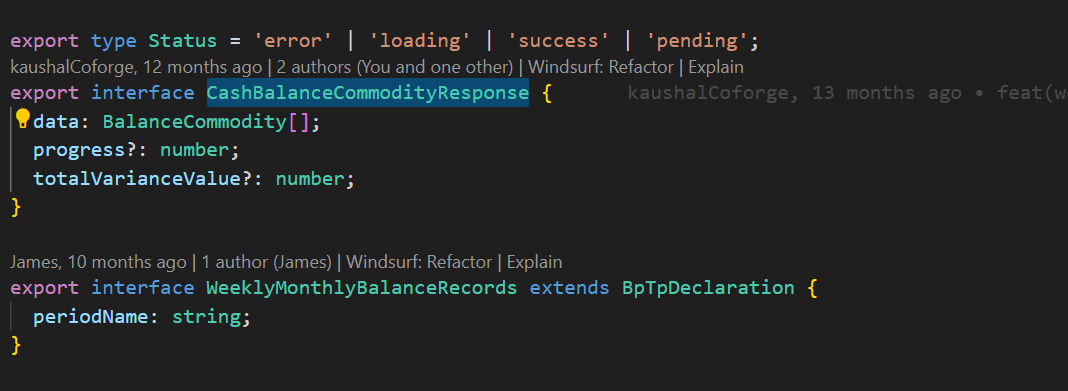


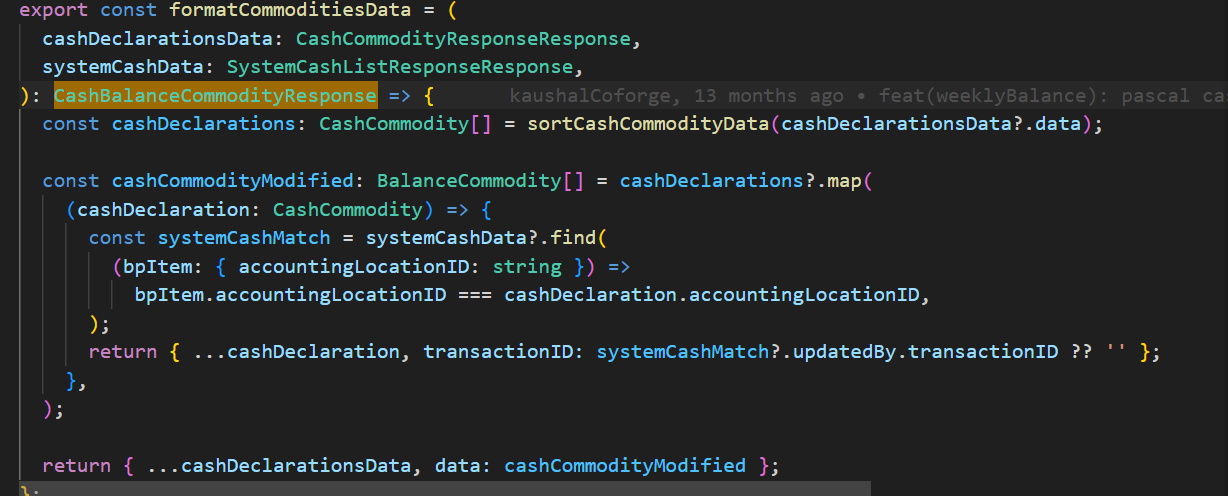


const router = useRouter();

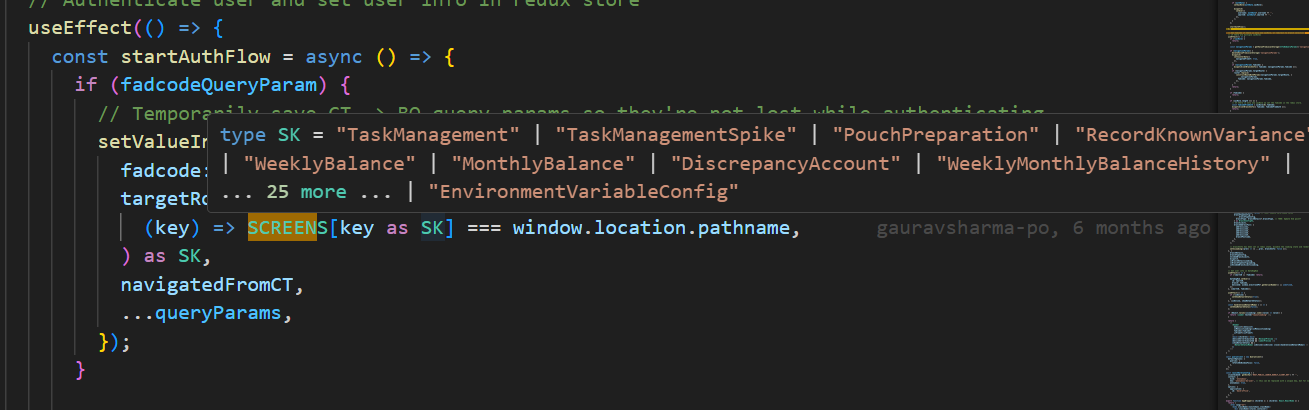


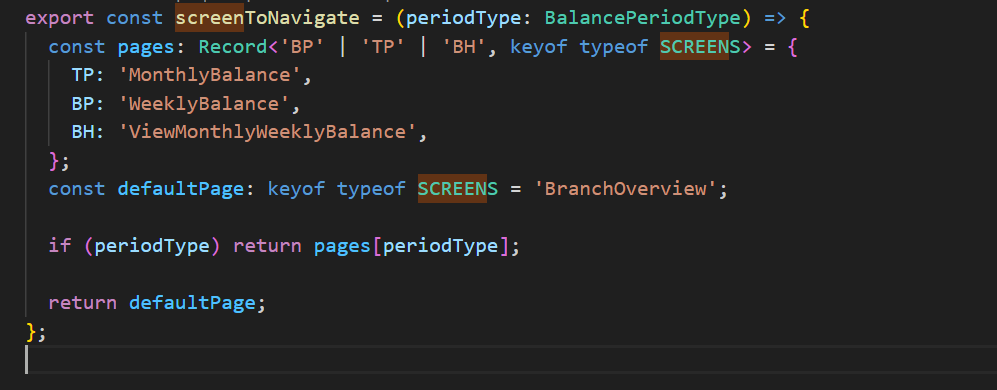
Definition and usage of the interface

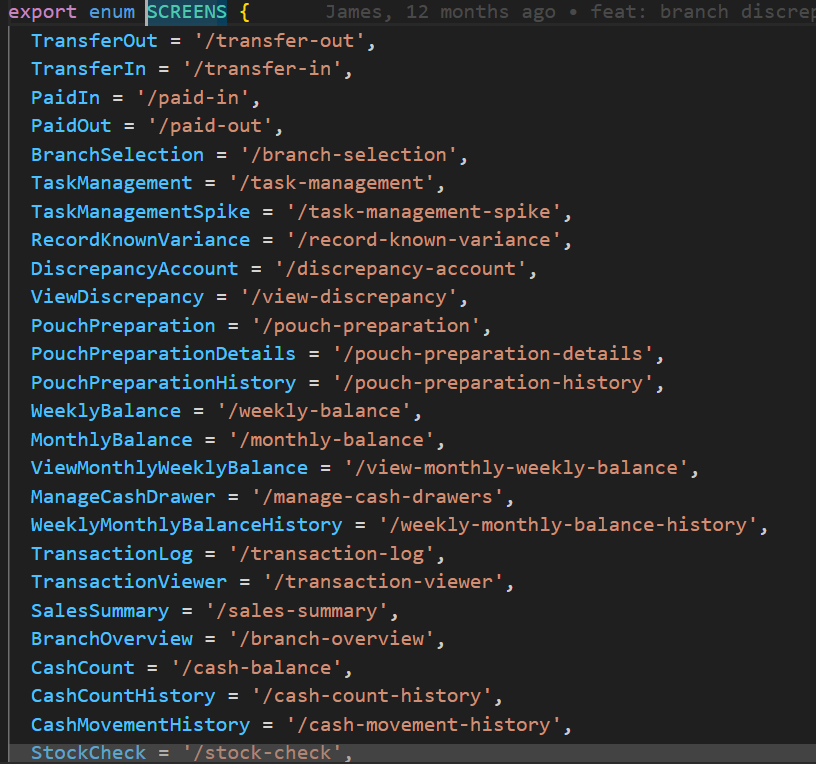


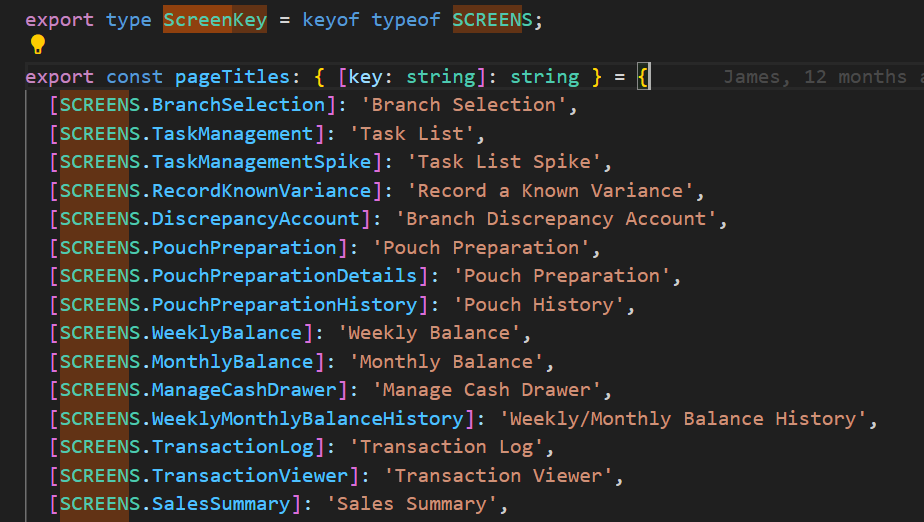


**Type casting the key as SK type**









**Tree Shaking**

Tree shaking is a form of dead code elimination, which removes unused code from the final build of an application. It relies on the static structure of ES modules (using import and export) to determine which code is actually used and which can be safely removed.

In React applications, tree shaking helps to reduce the size of the final bundle by eliminating unused code from third-party libraries and the application's own codebase. This leads to faster loading times and improved performance.

To enable tree shaking in a React application, it's important to:

* Use ES modules for imports and exports.
* Configure the build process to include a tool like webpack or Rollup, which support tree shaking.
* Ensure that Babel plugins that transform ES modules into CommonJS are not used, as this can break tree shaking.
* Define sideEffects property in package.json to inform webpack about files with side effects, ensuring correct tree shaking.

**How to create a react app:**

**React Router (v7)**

[**React Router**](https://reactrouter.com/start/framework/installation) **is the most popular routing library for React and can be paired with Vite to create a full-stack React framework**. It emphasizes standard Web APIs and has several [ready to deploy templates](https://github.com/remix-run/react-router-templates) for various JavaScript runtimes and platforms.

To create a new React Router framework project, run:

 Terminal

 Copy

npx create-react-router@latest

**To make api call using react query**

**Wrap the App component with QueryClientProvider for the useQuery to work**

**App.jsx**

import { QueryClient, QueryClientProvider } from 'react-query';

const queryClient = new QueryClient();

function App() {

return (

<QueryClientProvider client={queryClient}>

<GetData />

</QueryClientProvider>

);

}

**--------------------------------------------------------**

import axios from 'axios';

import { useQuery } from 'react-query';

function GetData() {

const { data, error, isLoading } = useQuery(

'data', // key

async () => {

const response = await axios.get('https://api.example.com/data');

return response.data;

}

);

if (isLoading) {

return <div>Loading...</div>;

}

if (error) {

return <div>Error: {error.message}</div>;

}

return <div>Data: {data}</div>;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

useQuery('data', async () => {

// ...

}, {

staleTime: 1000, // cache for 1 second

retry: 3, // retry 3 times

})

}

In this example, we're using the useQuery hook to fetch data from the API endpoint https://api.example.com/data. The first argument to useQuery is a unique key for the query, and the second argument is a function that returns a promise that resolves to the data.

You can also use the useQuery hook with a library like Axios:

-----------------------------------------------------------------------------------------------------------------------

Make sure to install react-query and import it in your React component.

Note: You can also use useQuery with other libraries like fetch or graphql etc.

Also, you can configure the query to handle caching, retry, and other options by passing options object as third argument to useQuery hook.