Filename: ReduxToolkit_Firebase_Expenses_Explanation.txt

Expenses App with Redux Toolkit + Firebase + Axios – Full Explanation

1. Axios

- Axios is a library to simplify HTTP requests (GET, POST, DELETE, PUT, PATCH) instead of using the native `fetch`.
- Advantages:
- Automatic JSON parsing.
- Handles HTTP errors automatically.
- Request cancellation and interceptors.
- Supports timeouts natively.
- Works on older browsers.
- Example of GET request:

```
import axios from "axios";

const response = await axios.get("https://example.com/data.json");
console.log(response.data);
```

- Example of POST request:

```
await axios.post("https://example.com/data.json", { title: "Expense 1" });
```

- Example of DELETE request:

```
await axios.delete("https://example.com/data/123.json");
```

---

2. Firebase Realtime Database

- Firebase Realtime Database stores data as JSON in the cloud.
- Each object has a **unique Firebase ID** generated automatically.
- Base URL example:

```
https://expenses-rtk-app-default-rtdb.firebaseio.com/expenses
```

- `.json` is required at the end for REST requests in Firebase.
- Fetch all data:

```
axios.get(`${FIREBASE_URL}.json`)
```

- Add new data:

```
axios.post(`${FIREBASE_URL}.json`, expense)
```

- Delete specific data:

```
axios.delete(`${FIREBASE_URL}/${firebaseId}.json`)
```

---

3. Redux Toolkit (RTK)

- RTK simplifies Redux setup with:
- `createSlice` → defines reducers and state.
- `createAsyncThunk` → handles asynchronous actions (like fetching from Firebase).
- `configureStore` → sets up the store in one line.
- Benefits:
- Less boilerplate.
- Built-in support for immutability (no need for `...state` manually).
- Async actions with `thunk` built-in.

3.1 Store Setup

```
import { configureStore, createSlice, createAsyncThunk } from "@reduxjs/
toolkit";
import axios from "axios";

const FIREBASE_URL = "https://expenses-rtk-app-default-rtdb.firebaseio.com/
expenses";
```

- We define the Firebase URL as a constant.

3.2 Fetching Data (AsyncThunk)

```
export const fetchExpenses = createAsyncThunk(
  "expenses/fetchExpenses",
  async () => {
    const response = await axios.get(`${FIREBASE_URL}.json`);
    const data = response.data;

    if (!data) return [];

    // Convert Firebase object to array
    return Object.keys(data).map(key => ({
      firebaseId: key,
      ...data[key],
    }));
  }
);
```

- `createAsyncThunk` automatically generates **pending, fulfilled, and rejected** action types. - Firebase returns objects keyed by ID, so we convert them to an array with `.map`.

3.3 Deleting an Expense

```
export const deleteExpenseFirebase = createAsyncThunk(
  "expenses/deleteExpenseFirebase",
  async (firebaseId) => {
    await axios.delete(`${FIREBASE_URL}/${firebaseId}.json`);
    return firebaseId;
  }
);
```

- Deletes an expense by Firebase ID and returns the ID to update Redux state.

3.4 Slice Setup

```
const expensesSlice = createSlice({
  name: "expenses",
  initialState: { items: [], loading: false, error: null },
  reducers: {
    addExpense(state, action) {
      state.items.push(action.payload);
    },
  },
```

```
  extraReducers: (builder) => {
    builder
      .addCase(fetchExpenses.pending, (state) => { state.loading = true; })
      .addCase(fetchExpenses.fulfilled, (state, action) => {
        state.loading = false;
        state.items = action.payload;
      })
      .addCase(fetchExpenses.rejected, (state) => {
        state.loading = false;
        state.error = "Failed to load expenses.";
      })
      .addCase(deleteExpenseFirebase.fulfilled, (state, action) => {
        state.items = state.items.filter(item => item.firebaseId !==
action.payload);
      });
  },
});
```

- `state.items` stores the array of expenses. - `addExpense` adds a local expense. - `extraReducers` handles async thunks. - `pending`, `fulfilled`, `rejected` are automatically generated by `createAsyncThunk`.

3.5 Store Configuration

```
export const { addExpense } = expensesSlice.actions;

const store = configureStore({
  reducer: {
    expenses: expensesSlice.reducer,
  },
});

export default store;
```

- `configureStore` automatically adds `thunk` middleware. - `store` is used with `<Provider store={store}>` in the React app.

---

4. Connecting Redux with React

- `useDispatch()` → dispatch actions (like `addExpense`, `fetchExpenses`).
- `useSelector()` → access state from the store. Example:

```
const expenses = useSelector(state => state.expenses.items);
const dispatch = useDispatch();
```

---

5. Expenses Form Component

- Uses `useRef()` to get input values.
- On submit:
- Validate inputs.
- Create expense object.
- Dispatch `addExpense`.
- Send POST request to Firebase using Axios.

---

6. Expenses Table Component

- Uses `useSelector` to get expenses array.
- Maps each expense into table rows.
- Delete button:
- Calls `deleteExpenseFirebase`.
- Redux updates state automatically when fulfilled.

---

7. Skeletons & Loading

- While `fetchExpenses` is loading:
- Show `Skeleton` components from MUI for image, inputs, and table rows.
- Avoid showing empty data.
- Example:

```
{loading ? <Skeleton variant="rectangular" height={50} /> : <TableRow> ...
</TableRow>}
```

---

8. Workflow Summary

1. **App loads** → `useEffect` dispatches `fetchExpenses`.
2. **Redux thunk** calls Firebase GET → converts data → stores in Redux state.
3. **User adds expense** → Form dispatches `addExpense` → Axios POST → expense added to Firebase.
4. **User deletes expense** → Table dispatches `deleteExpenseFirebase` → Axios DELETE → Redux state updates automatically.
5. **UI shows Skeletons** while loading.

✅This file contains a **complete line-by-line explanation** of the Expenses App using Redux Toolkit, Axios, Firebase, and MUI skeletons.