Full Stack Development BootCamp - Day 6

Build Real-World Apps from Backend to Frontend!

Trainer:

Muhammad Maaz Sheikh
Technical Team Lead

Tech Stack Covered:

- Nest Js
- PostgreSQL
- React Js
- Real world project development

Organized By:

Iqra University

Recap Day 5

Table of Content

Section A-9: State Management with RTK (Redux Toolkit)

Section A-10: CRUD Rest APIs and React Component Integration

State Management with RTK (Redux Toolkit)

State Management with RTK (Redux Toolkit)

What is State Management?

🧠 **Definition:**

- State = the data your app needs to work (e.g., a counter value, user info, list of products).
- State management = how your app reads and updates that data.

🎯 **Why it matters:**

- Keeps your app consistent.
- Makes it easier to debug and maintain.
- Avoids "prop drilling" by managing state globally.

Imagine you have data like:

- A user is logged in ✅ or not ❌
- A shopping cart has 2 items
- A counter shows number = 5

All of this is state. Redux will helps you manage this globally, so any component can use or change it.

State Management with RTK (Redux Toolkit)

What is Redux Toolkit (RTK)?

🧠 **Definition:**

Redux Toolkit, or RTK, is a node package that simplifies development by providing utility functions. It is made to simplify the creation of a redux store and provide easy state management. It can be easily installed using simple npm commands.

Before the introduction of the Redux toolkit, state management was complex in simple redux.

The Redux toolkit is a wrapper around Redux and encapsulates its necessary functions. Redux toolkit is flexible and provides a simple way to make a store for large applications. It follows the SOPE principle, which means it is Simple, Opinionated, Powerful, and Effective.

Problems solved by Redux Toolkit (RTK) in Redux:

Redux Toolkit was created to solve these three common problems that we face in **Redux**.

- Too much code to configure the store.
- Writing too much boilerplate code to dispatch actions and store the data in the reducer.
- Extra packages like Redux-Thunk and Redux-Saga for doing asynchronous actions.

Check this article to know **Why Redux Toolkit is preferred over Redux**.

State Management with RTK (Redux Toolkit)

Steps to Install and Import Redux Toolkit(RTK)

Step#01: Redux Toolkit (RTK) Installation

To install the redux toolkit in our project type the following command in the terminal.

Commands:

```
npm i @reduxjs/toolkit //The main library for Redux Toolkit
npm i react-redux      // We also need basic redux with this application
```

Step#02: Importing Redux Toolkit (RTK)

We import only the necessary functions in our code

Importing in Code:

```
// Importing
import { configureStore } from '@reduxjs/toolkit';      // Creates a store
import { createSlice } from '@reduxjs/toolkit';          // Combines and creates reducer
```

Dependencies after installing the the Redux Toolkit:

```
"dependencies": {
   "react": "^17.0.2",
   "react-dom": "^17.0.2",
   "@reduxjs/toolkit": "^1.6.1",
   "react-redux": "^7.2.5"
 }
```

State Management with RTK (Redux Toolkit)

Benefits of Redux Toolkit (RTK)

- Easier state management as compared to Redux
- Boilerplate code for the majority of functions
- Official recommended SOPE library
- Wrapper functions are provided which reduce lines of code

Important function provided by Redux Toolkit:

- The createStore function in basic Redux is wrapped by the configureStore function which automatically provides middlewares and enhancers.

- Classic reducer is replaced by createReducer function which makes the code shorter and simpler to understand.

- The **createAction()** utility that returns an action creator function.

- Redux **createSlice()** function comes in handy to replace create action and create Reducer functions with a single function.

- Redux **createAsyncThunk()** that takes Redux strings as arguments and returns a Promise.

- Redux **createEntityAdapter()** utility that helps to perform CRUD operations.

State Management with RTK (Redux Toolkit)

Setting up Redux Store with Redux Toolkit

The store is the main box that holds all your slices.

Create a file in **src/app/store.ts**

store.ts:

```
// store.ts
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';

export const store = configureStore({
 reducer: { counter: counterReducer }
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

State Management with RTK (Redux Toolkit)

RTK Slices (Reducers, Actions, Selectors)

In Redux Toolkit (RTK), a Slice is a collection of reducers, actions, and initial state that manage a specific part (or "slice") of your application's state. RTK makes Redux code easier, cleaner, and less repetitive.

A slice contains:

Element	Description
name	Unique name of the slice
initialState	The default state for this slice
reducers	Functions that define how state is updated (like methods for state updates)
actions	Auto-generated action creators based on reducers
selectors	Functions to extract slice data from the global state (written manually)

State Management with RTK (Redux Toolkit)

RTK Slices (Reducers, Actions, Selectors)

Create a file in **src/features/counter/counterSlice.ts**

counterSlice.ts:

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = { value: 0 };
const counterSlice = createSlice({
  name: 'counter',  // Name of this slice
  initialState,     // Starting state
  reducers: {
    increment: (state) => {
      state.value += 1; // Add 1
    },
    decrement: (state) => {
      state.value -= 1; // Subtract 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload; // Add custom value
    },
  },
});

// Export actions to use in components
export const { increment, decrement, incrementByAmount } = counterSlice.actions;
// Export the reducer to put in the store
export default counterSlice.reducer;
```

State Management with RTK (Redux Toolkit)

RTK Slices (Reducers, Actions, Selectors)

Selectors:

Create a file for selector in **src/features/counter/counterSelector.ts**

counterSelector.ts:

```
import { RootState } from "../../app/store";

export const selectCounterValue = (state: RootState) => state.counter.value;
```

State Management with RTK (Redux Toolkit)

Connect Store with React and Use Redux in a Component

Connect the Store to React:

Wrap your React app with the Redux <Provider>, so Redux is available to all components.
In your **main.tsx**:

main.tsx:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App'
import { Provider } from 'react-redux';
import { store } from './app/store';

createRoot(document.getElementById('root')!).render(
 <Provider store={store}>
   <App />
 </Provider>,
)
```

State Management with RTK (Redux Toolkit)

Connect Store with React and Use Redux in a Component

Use Redux in a Component:

Now let's use the counter in a real component.
Create a file for Counter in **src/features/counter/counter.tsx**

counter.tsx:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementByAmount } from '../../features/counter/counterSlice';
import { RootState } from '../../app/store';
import { selectCounterValue } from './counterSelector';

function Counter() {
 const count = useSelector(selectCounterValue);; // Get state
 const dispatch = useDispatch(); // To call actions

 return (
   <div>
     <h2>Count: {count}</h2>
     <button onClick={() => dispatch(increment())}>+1</button>
     <button onClick={() => dispatch(decrement())}>-1</button>
     <button onClick={() => dispatch(incrementByAmount(5))}>+5</button>
   </div>
 );
}

export default Counter;
```

State Management with RTK (Redux Toolkit)

Connect Store with React and Use Redux in a Component

Use Redux in a Component:

Now let's use the counter in a real component.
Create another file to see the reflection of state of counter in **src/features/counter/counterReflection.tsx**

counterReflection.tsx:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementByAmount } from '../../features/counter/counterSlice' ;
import { RootState } from '../../app/store';
import { selectCounterValue } from './counterSelector';

function CounterReflection () {
 const count = useSelector(selectCounterValue); // Get state

 return (
   <div>
     <h2>Count from other component: {count}</h2>
   </div>
 );
}
export default CounterReflection;
```

State Management with RTK (Redux Toolkit)

Connect Store with React and Use Redux in a Component

Use Counter and CounterReflection in App.tsx:

Now let's use the counter and CounterReflection in a app.tsx
App.tsx should look like below file

app.tsx:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'
import Counter from './features/counter/counter' ;
import CounterReflection from './features/counter/counterReflection' ;
function App() {
 return (
    <>
        <h1>New Counter Component</h1>
        <Counter />
        <CounterReflection />
    </>
 )
}
export default App
```

State Management with RTK (Redux Toolkit)

Connect Store with React and Use Redux in a Component

What Just Happened in a Component?

Code	Explanation
useSelector	Reads value from Redux state
useDispatch	Sends actions like increment()
increment()	Changes state by +1
selectCounterValue	Comes from the selector.ts reading state from store

State Management with RTK (Redux Toolkit)

Asynchronous State handling with RTK Query

🧠 What is Asynchronous State Handling?

In React, sometimes you need to fetch data from an API—like getting a list of products or user details. Since this data comes from a remote server, it's asynchronous, meaning it doesn't come instantly.

RTK Query is a powerful tool from Redux Toolkit that helps us handle this async data easily with built-in caching, loading, and error states.

Why use RTK Query?

RTK Query automates:

- Making API calls (GET, POST, etc.)
- Caching data
- Showing loading indicators
- Handling success and error cases
- Refetching when needed

You don’t need to manually manage **useEffect**, **useState**, or **axios** to fetch data.

State Management with RTK (Redux Toolkit)

Asynchronous State handling with RTK Query

Step-by-Step: RTK Query Asynchronous Data Handling

Step# 1: Install dependencies

Install react redux tool kit if not installed already by using below command

- ➔ npm install @reduxjs/toolkit react-redux

Step# 2: Create User Type

- ➔ Create **user** folder in features
- ➔ Create **user.type.ts** in **types** folder

Use Below Code for user.type.ts:

```
export interface User{
    id: number;
    name: string;
    email: string;
    password: string;
}
```

State Management with RTK (Redux Toolkit)

Asynchronous State handling with RTK Query

Step# 3: Create an API Slice

The **createApi** function sets up the endpoints for fetching data.
Create **userApi.ts** in **src/features/users** folder:

Code for userApi.ts:

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';
import { User } from '../product/types/user.type';

export const userApi = createApi({
 reducerPath: 'api', // optional name for the reducer
 baseQuery: fetchBaseQuery({ baseUrl: 'http://localhost:3000/' }),
 tagTypes: ['User'],
 endpoints: (builder) => ({
   getUsers: builder.query<User[], void>({
     query: () => 'users',
     providesTags: ['User'],
   }),
 }),
});

export const { useGetUsersQuery } = userApi;
```

State Management with RTK (Redux Toolkit)

Asynchronous State handling with RTK Query

Explanation of ApiSlice:

- **baseQuery** defines the base URL for your API.
- **getUsers** is a query endpoint that fetches data from /users.
- **useGetUsersQuery** is an auto-generated React hook.

Step# 4: Add User API to the store

- Integrate the RTK Query reducer and middleware to the Redux store.
- Update code in **app/store.ts** in src folder:

```typescript
// store.ts
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';
import { userApi } from '../features/services/userApi';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
    [userApi.reducerPath]: userApi.reducer
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(userApi.middleware),
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

State Management with RTK (Redux Toolkit)

Asynchronous State handling with RTK Query

Step# 5: Provide the Redux Store

- Wrap your app in the Provider if not already done check **main.tsx** file it should look like below one.

```
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App'
import { Provider } from 'react-redux';
import { store } from './app/store';

createRoot(document.getElementById('root')!).render(
  <Provider store={store}>
    <App />
  </Provider>,
)
```

Step# 6: Use RTK Query hook in your userLists.tsx component

- Add new file in user folder with name **userLists.tsx** and use code pasted in **step# 07**

Explanation of userLists.tsx:

- **isLoading**: true on first load.
- **isFetching**: true on background updates.
- **data**: the actual API response.
- **error**: error info if request fails.

State Management with RTK (Redux Toolkit)

Handling Loading, Error, and Data State

Step# 7: Code for userLists.tsx component

```
import { useGetUsersQuery } from '../services/userApi';
import { User } from './types/user.type';

const UserList = () => {

  const { data, error, isLoading, isFetching } = useGetUsersQuery();

  if (isLoading) return <p>Loading posts...</p>;
  if (error) return <p>Error fetching posts!</p>;

  return (
    <div>
      <h2>Users List</h2>
      <table>
        <thead>
          <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Email</th>
          </tr>
        </thead>
        <tbody>
          {data?.map((usr: User) => (
            <tr>
              <td>{usr.id}</td>
              <td>{usr.name}</td>
              <td>{usr.email}</td>
            </tr>
          ))}
        </tbody>
      </table>
      {isFetching && <p>Updating...</p>}
    </div>
  );
}
export default UserList;
```

State Management with RTK (Redux Toolkit)

Handling Loading, Error, and Data State

Step# 8: Usage in App.tsx

After creating **userLists.tsx**, import it into your main App.tsx to render the list:

Summary: Benefits of RTK Query Async Handling

1. Fetching Data Efficiently

How It Works:

- RTK Query generates **custom hooks** (**useGetUsersQuery**) that automatically fetch data from APIs.
- It uses **fetchBaseQuery**, which is built on top of fetch().

How to Check:

- Open your browser's Network tab (F12 → Network).
- Call **useGetUsersQuery** (), and you'll see a GET request sent.
- It sends the request once, not repeatedly.

```
const { data, error, isLoading, isFetching } = useGetUsersQuery();
```

State Management with RTK (Redux Toolkit)

Handling Loading, Error, and Data State

Summary: Benefits of RTK Query Async Handling

2. Built-In Caching

How It Works:

- RTK Query caches fetched data and doesn't refetch unless needed.
- It reuses cached results for components using the same hook (**useGetUsersQuery**), improving speed.

How to Check:

- Load the page → Check Network tab (a GET request is sent).
- Navigate away and come back → ✅ No additional request is made (cached!).
- Unless cache is invalidated (e.g., using invalidatesTags after mutation), the old data is reused.

```
const { data, error, isLoading, isFetching } = useGetUsersQuery({ refetchOnMountOrArgChange: 60 });
```

State Management with RTK (Redux Toolkit)

Handling Loading, Error, and Data State

Summary: Benefits of RTK Query Async Handling

3. Loading, Error & Success States Built-In

How It Works:

- RTK Query provides several flags for each API call:

- **isLoading ->** First time request in progress
- **isFetching ->** Any request (even after cache)
- **isError ->** Request failed
- **Error ->** Error object (status, message)
- **isSuccess ->** Data successfully fetched

```
const { data, error, isLoading, isFetching } = useGetUsersQuery();

if (isLoading) return <p>Loading posts...</p>;
if (error) return <p>Error fetching posts!</p>;
```

State Management with RTK (Redux Toolkit)

Handling Loading, Error, and Data State

Summary: Benefits of RTK Query Async Handling

4. Performance Improvements

Why RTK Query is Fast::

- ❌ No need for useEffect, useState, dispatch, axios manually.
- ✅ Auto-deduplicates network calls.
- ✅ Auto-memoizes results.
- ✅ Batches and reuses data from cache.
- ✅ Refetches only when necessary.

State Management with RTK (Redux Toolkit)

Handling Loading, Error, and Data State

Summary: Benefits of RTK Query Async Handling

5. DevTools Debugging Support

Install Redux DevTools extension:

- **Chrome**: https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd

How to Use:

- Click Redux tab → You'll see RTK Query getProducts/fulfilled, getProducts/pending etc.
- You can track each API lifecycle action and the cached state.

CRUD Rest APIs and React Component Integration

State Management with RTK (Redux Toolkit)

CRUD Rest API with Integration

Let's build out a basic Product CRUD (Create, Read, Update, Delete) implementation using RTK Query, React with TypeScript, and a simple form UI.

What we will build ?

- ✅ **Fetch user list (already done)**
- ➕ **Add new user via a form**
- ✏️ **Update user(inline or via form)**
- ❌ **Delete user**
- 📃 **Auto-update the user table using RTK Query cache invalidation**

State Management with RTK (Redux Toolkit)

Project Structure

Let's build out a basic Product CRUD (Create, Read, Update, Delete) implementation using RTK Query, React with TypeScript, and a simple form UI.

```
│
├── features/
│   ├── users/
│   │   ├── UserList.tsx
│   │   ├── UserForm.tsx
│   │   └── usersApi.ts
│
├── app/
│   └── store.ts
│
```

State Management with RTK (Redux Toolkit)

Implementation and Setup of CRUD in Toolkit

Step 1: Extend userApi.ts

Add or extend remaining code in userAPI file for Create, Update and Delete and export **useQuery** Hook.

```
addUser: builder.mutation<User, Partial<User>>({
    query: (newUser) => ({
      url: 'users',
      method: 'POST',
      body: newUser,
    }),
    invalidatesTags: ['User'],
  }),
  updateUser: builder.mutation<User, User>({
    query: (user) => ({
      url: `users/${user.id}`,
      method: 'PUT',
      body: user,
    }),
    invalidatesTags: ['User'],
  }),
  deleteUser: builder.mutation<{ success: boolean }, number>({
    query: (id) => ({
      url: `users/${id}`,
      method: 'DELETE',
    }),
    invalidatesTags: ['User'],
  }),
```

```
export const {
  useGetUsersQuery,
  useAddUserMutation,
  useUpdateUserMutation,
  useDeleteUserMutation,
} = userApi;
```

State Management with RTK (Redux Toolkit)

Implementation and Setup of CRUD in Toolkit

Step 2: Add UserForm.tsx

Add or extend remaining code in userAPI file for Create, Update and Delete and export **useQuery** Hook.

```
import React, { useState } from 'react';
import { useAddUserMutation } from './userApi';

const UserForm = () => {
 const [name, setName] = useState('');
 const [email, setEmail] = useState('');
 const [password, setPassword] = useState('');
 const [addUser] = useAddUserMutation();

 const handleSubmit = async (e: React.FormEvent) => {
   e.preventDefault();
   if (!name || !email) return;

   await addUser({ name, email, password });
   setName('');
   setEmail('');
   setPassword('');
 };
```

State Management with RTK (Redux Toolkit)

Implementation and Setup of CRUD in Toolkit

Step 2: Add UserForm.tsx (Continue)

```
return (
  <form onSubmit={handleSubmit} style={{ marginBottom: 20 }}>
    <input
      value={name}
      onChange={(e) => setName(e.target.value)}
      placeholder="Enter Name"
      required
    />
    <input
      value={email}
      onChange={(e) => setEmail(e.target.value)}
      placeholder="Enter Email"
      required
      type="Email"
    />
    <input
      value={password}
      onChange={(e) => setPassword(e.target.value)}
      placeholder="Enter Password"
      type="Password"
    />
    <button type="submit">Add User</button>
  </form>
);
};
export default UserForm;
```

State Management with RTK (Redux Toolkit)

Implementation and Setup of CRUD in Toolkit

Step 3: Enhance UserList.tsx with Delete

```
const [deleteUser] = useDeleteUserMutation();
```

- Add Columns name also and named it Action to record the actions.
- Add th column in table header

```
<th>Actions</th>
```

- Add td in table body

```
<td>
  {/* Update logic can be added here */}
  <button onClick={() => deleteUser(usr.id)}>Delete</button>
</td>
```

State Management with RTK (Redux Toolkit)

Step-by-Step: Update Logic in the Same UserForm

Step 1: Modify UserForm.tsx to handle both add and edit.

- User Form now should look like this.

```
import React, { useState, useEffect } from 'react';
import { useAddUserMutation } from '../services/userApi';
import { User } from './types/user.type';
import { useUpdateUserMutation } from '../services/userApi';
interface Props {
  selectedUser?: User | null;
  onSuccess: () => void;
}

const UserForm: React.FC<Props> = ({ selectedUser, onSuccess }) => {
 const [name, setName] = useState('');
 const [email, setEmail] = useState('');
 const [password, setPassword] = useState('');
 const [addUser] = useAddUserMutation();
 const [updateUser] = useUpdateUserMutation();

 useEffect(() => {
   if (selectedUser) {
       setName(selectedUser.name);
       setEmail(selectedUser.email);
       setPassword(selectedUser.password);
   }
 }, [selectedUser]);
```

State Management with RTK (Redux Toolkit)

Step-by-Step: Update Logic in the Same UserForm

Step 1: Modify UserForm.tsx to handle both add and edit.

- User Form now should look like this. (continue)

```
const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault();
  if (!name || !email) return;

  const payload = {
      name,
      email,
      password,
    };

  if (selectedUser) {
      await updateUser({ ...selectedUser, ...payload });
    } else {
      await addUser({ payload });
    }
  setName('');
  setEmail('');
  setPassword('');
  onSuccess();
};
```

State Management with RTK (Redux Toolkit)

Step-by-Step: Update Logic in the Same UserForm

Step 1: Modify UserForm.tsx to handle both add and edit.

- User Form now should look like this. (continue)

```
return (
  <form onSubmit={handleSubmit} style={{ marginBottom: 20 }}>
    <h3>{selectedUser ? 'Update User' : 'Add User'}</h3>
    <input
      value={name}
      onChange={(e) => setName(e.target.value)}
      placeholder="Enter Name"
      required
    />
    <input
      value={email}
      onChange={(e) => setEmail(e.target.value)}
      placeholder="Enter Email"
      required
      type="Email"
    />
    <input
      value={password}
      onChange={(e) => setPassword(e.target.value)}
      placeholder="Enter Password"
      type="Password"
    />    <button type="submit">{selectedUser ? 'Update' : 'Add'}</button>
  </form>
 );
};
export default UserForm;
```

State Management with RTK (Redux Toolkit)

Step-by-Step: Update Logic in the UserList

Step 2: Add State to UserList to Select Item for Editing

- Add below line of code to get the particular record for update..

```
const [selectedUser, setSelectedUser] = useState<User | null>(null);
```

- Call UserForm component from here now instead of app.tsx because now we need to pass props to update the records to cater this make below changes

```
<UserForm
        selectedUser={selectedUser}
        onSuccess={() => setSelectedUser(null)}
      />
```

- Add Edit Button before Delete Button in the code just like below

```
<td>
            <button onClick={() => setSelectedUser(usr)}>
              Edit
            </button>{' '}
            <button onClick={() => deleteProduct(usr.id)}>Delete</button>
          </td>
```

State Management with RTK (Redux Toolkit)

Step-by-Step: Update Logic in the Same UserForm

Final Notes

- When **Edit** is clicked, the form populates.
- On submission, **updateProduct** is triggered via RTK Mutation.
- **invalidatesTags** automatically **refetches** the list using cache invalidation.
- The form clears after submission.