how ot manage shadcn theme in vite app

Got it 👍 You're using **Vite** and want to set up and manage a **shadcn/ui theme**.

Here's the step-by-step way to manage themes (light/dark/custom) with **shadcn/ui + Vite**:

---

1. Install dependencies

Make sure you have these installed:

```bash
npm install shadcn-ui clsx tailwind-variants tailwindcss-animate
npm install tailwindcss postcss autoprefixer
npm install @radix-ui/react-icons
```

If you don't already have Tailwind configured in Vite:

```bash
npx tailwindcss init -p
```

---

2. Configure Tailwind for themes

In your `tailwind.config.js`:

```js
/** @type {import('tailwindcss').Config} */
module.exports = {
  darkMode: ["class"], // important for theme switching
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [require("tailwindcss-animate")],
}
```

---

3. Add CSS variables for shadcn theme

In `src/index.css` (or `globals.css` if you created one):

```css
@tailwind base;
@tailwind components;
@tailwind utilities;
```

```
@layer base {
  :root {
    --background: 0 0% 100%;
    --foreground: 240 10% 3.9%;

    --primary: 240 5.9% 10%;
    --primary-foreground: 0 0% 98%;

    --secondary: 240 4.8% 95.9%;
    --secondary-foreground: 240 5.9% 10%;
  }

  .dark {
    --background: 240 10% 3.9%;
    --foreground: 0 0% 98%;

    --primary: 0 0% 98%;
    --primary-foreground: 240 5.9% 10%;

    --secondary: 240 3.7% 15.9%;
    --secondary-foreground: 0 0% 98%;
  }
}
```

These variables drive shadcn components.

---

4. Wrap your app with a Theme Provider

Create `src/components/theme-provider.tsx`:

```tsx
import * as React from "react"
import { ThemeProvider as NextThemesProvider } from "next-themes"

export function ThemeProvider({ children }: { children: React.ReactNode }) {
  return (
    <NextThemesProvider
      attribute="class"
      defaultTheme="system"
      enableSystem
      disableTransitionOnChange
    >
      {children}
    </NextThemesProvider>
  )
}
```

---

5. Use it in `main.tsx`

```tsx
import React from "react"
import ReactDOM from "react-dom/client"
import App from "./App"
import { ThemeProvider } from "./components/theme-provider"
import "./index.css"

ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <ThemeProvider>
      <App />
    </ThemeProvider>
  </React.StrictMode>
)
```

---

6. Add a Theme Toggle Button

You can now add a theme switcher using shadcn button + icons:

```tsx
import { useTheme } from "next-themes"
import { Sun, Moon } from "lucide-react"
import { Button } from "@/components/ui/button"

export function ThemeToggle() {
  const { theme, setTheme } = useTheme()

  return (
    <Button
      variant="ghost"
      size="icon"
      onClick={() => setTheme(theme === "dark" ? "light" : "dark")}
    >
      {theme === "dark" ? <Sun className="h-5 w-5" /> : <Moon className="h-5 w-5" />}
    </Button>
  )
}
```

✅ That's it!
Now you have **theme management in a Vite app with shadcn/ui**, and you can extend the CSS variables to make custom themes.

Do you want me to also show you **how to add multiple custom themes (not just light/dark)** in shadcn with Vite?

how to update tailwind version

Updating **Tailwind CSS** in your Vite app is pretty straightforward. Here's the process 👇

---

1. Check your current Tailwind version

Run:

```bash
npm list tailwindcss
```

---

2. Update Tailwind to the latest version

Run:

```bash
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest
```

This updates Tailwind, PostCSS, and Autoprefixer (they often release together).

If you're using **pnpm**:

```bash
pnpm add -D tailwindcss@latest postcss@latest autoprefixer@latest
```

Or **yarn**:

```bash
yarn add -D tailwindcss@latest postcss@latest autoprefixer@latest
```

---

3. Update Config (if needed)

Sometimes newer Tailwind versions change defaults. Open `tailwind.config.js` and make sure it's aligned with the latest docs ↗.

For example, if you're using shadcn/ui, you need:

```js
darkMode: ["class"], // not just 'media'
```

---

4. Restart Dev Server

Stop and restart your Vite server:

```bash
npm run dev
```

---

5. (Optional) Verify installed version

```bash
npx tailwindcss -v
```

This will show the current installed Tailwind version.

👉 Do you want me to give you a **one-shot script** that both updates Tailwind and fixes any breaking changes (like plugin renames, config updates) for shadcn in Vite?

how theme will be toggled

Great question 👍 Let's connect the dots.

Since you're using **Vite + shadcn/ui + Tailwind**, the theme toggle works by:

1. **Using `next-themes` (or your own state) to set a class (`light` / `dark`) on `<html>` or `<body>`.**
2. **Tailwind's `darkMode: "class"` setting** tells Tailwind to apply dark styles only when `.dark` class is present.
3. **Your shadcn theme variables** (in `:root` and `.dark`) switch values automatically when the class changes.

Example Workflow

1. `tailwind.config.js`

```js
export default {
  darkMode: ["class"], // important for toggling
  content: ["./index.html", "./src/**/*.{js,ts,jsx,tsx}"],
  theme: { extend: {} },
  plugins: [require("tailwindcss-animate")],
}
```

---

2. Theme Provider (`theme-provider.tsx`)

```tsx
import { ThemeProvider as NextThemesProvider } from "next-themes"
import { ReactNode } from "react"

export function ThemeProvider({ children }: { children: ReactNode }) {
  return (
    <NextThemesProvider
      attribute="class"
      defaultTheme="system"
      enableSystem
      disableTransitionOnChange
    >
      {children}
    </NextThemesProvider>
  )
}
```

This will put `class="light"` or `class="dark"` on `<html>`.

---

3. Theme Toggle Button

```tsx
import { useTheme } from "next-themes"
import { Sun, Moon } from "lucide-react"
import { Button } from "@/components/ui/button"

export function ThemeToggle() {
  const { theme, setTheme } = useTheme()

  return (
    <Button
      variant="ghost"
      size="icon"
      onClick={() => setTheme(theme === "dark" ? "light" : "dark")}
    >
      {theme === "dark" ? <Sun className="h-5 w-5" /> : <Moon className="h-5 w-5" />}
    </Button>
  )
}
```

Clicking this button will **toggle the `.dark` class on `<html>`**, and Tailwind + shadcn CSS variables will instantly switch.

---

4. Global Theme Variables (`index.css`)

```css
@layer base {
  :root {
    --background: 0 0% 100%;
    --foreground: 240 10% 3.9%;
  }

  .dark {
    --background: 240 10% 3.9%;
    --foreground: 0 0% 98%;
  }
}
```

✅ When you press the toggle:

- `.dark` class is applied/removed →
- Tailwind dark utilities (`dark:bg-black`) activate →
- CSS variables in `.dark { ... }` apply →
- All **shadcn/ui components update theme automatically**.

Do you want me to also show you how to **remember the user's theme choice** (so it stays dark/light even after refreshing)?

```
import { configureStore } from '@reduxjs/toolkit';
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux';
import authSlice from './slices/authSlice';
import categoriesSlice from './slices/categoriesSlice';
import productsSlice from './slices/productsSlice';
import ordersSlice from './slices/ordersSlice';
import usersSlice from './slices/usersSlice';
import offersSlice from './slices/offersSlice';
import couponsSlice from './slices/couponsSlice';
import uiSlice from './slices/uiSlice';

export const store = configureStore({
  reducer: {
    auth: authSlice,
    categories: categoriesSlice,
    products: productsSlice,
    orders: ordersSlice,
    users: usersSlice,
    offers: offersSlice,
    coupons: couponsSlice,
    ui: uiSlice,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: ['persist/PERSIST'],
      },
    }),
});

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

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;
```

make it scalable industry level also and persistenece

Nice — I'll upgrade this to a production-ready, scalable pattern with:

- **combineReducers + reducer manager** for code-splitting / dynamic reducer injection
- **redux-persist** for persistence (with a safe whitelist)
- Proper **typed hooks** and exports (`store`, `persistor`, `injectReducer` helper)
- Dev-only **logger**, strict **serializableCheck** config using `redux-persist` action constants
- Notes about where to add RTK Query / Sagas / extra middleware

Paste the files into your `src/store` (or `src/redux`) folder.

---

1) `src/store/rootReducer.ts`

```ts
// src/store/rootReducer.ts
import { combineReducers } from '@reduxjs/toolkit';
import authSlice from '../slices/authSlice';
import categoriesSlice from '../slices/categoriesSlice';
import productsSlice from '../slices/productsSlice';
import ordersSlice from '../slices/ordersSlice';
import usersSlice from '../slices/usersSlice';
import offersSlice from '../slices/offersSlice';
import couponsSlice from '../slices/couponsSlice';
import uiSlice from '../slices/uiSlice';

/**
 * Static reducers - reducers that always exist.
 * For very large apps, you can keep this minimal and inject feature reducers dynamically.
 */
export const staticReducers = {
  auth: authSlice,
  categories: categoriesSlice,
  products: productsSlice,
  orders: ordersSlice,
  users: usersSlice,
  offers: offersSlice,
  coupons: couponsSlice,
  ui: uiSlice,
};

export const createRootReducer = (asyncReducers = {}) =>
  combineReducers({
    ...staticReducers,
    ...asyncReducers,
  });

export type StaticReducers = typeof staticReducers;
```

---

2) `src/store/reducerManager.ts` (dynamic reducer manager)

```ts
// src/store/reducerManager.ts
import { Reducer, AnyAction } from 'redux';
import { createRootReducer } from './rootReducer';

export function createReducerManager(initialReducers: Record<string, Reducer>) {
  let reducers = { ...initialReducers };
  let combinedReducer = createRootReducer();
  const keysToRemove: string[] = [];

  return {
    getReducerMap: () => reducers,

    reduce: (state: any, action: AnyAction) => {
      if (keysToRemove.length > 0) {
        state = { ...state };
        for (const key of keysToRemove) {
          delete state[key];
        }
        keysToRemove.length = 0;
      }
      return combinedReducer(state, action);
    },

    add: (key: string, reducer: Reducer) => {
      if (!key || reducers[key]) return;
      reducers = { ...reducers, [key]: reducer };
      combinedReducer = createRootReducer(reducers);
    },
    remove: (key: string) => {
      if (!key || !reducers[key]) return;
      delete reducers[key];
      keysToRemove.push(key);
      combinedReducer = createRootReducer(reducers);
    },
  };
}
```

---

3) `src/store/store.ts`

```ts
// src/store/store.ts
import { configureStore, EnhancedStore, Middleware } from '@reduxjs/toolkit';
import { Persistor, persistStore, persistReducer, FLUSH, PAUSE, PERSIST, PURGE, REGISTER,
REHYDRATE } from 'redux-persist';
import storage from 'redux-persist/lib/storage'; // localStorage for web
import logger from 'redux-logger';
import thunk from 'redux-thunk';
import { createReducerManager } from './reducerManager';
import { createRootReducer, staticReducers } from './rootReducer';

/**
 * Persist config - carefully choose what to persist.
 * Avoid persisting sensitive info (or encrypt it) and large objects.
 */
const persistConfig = {
  key: 'root',
  version: 1,
  storage,
  whitelist: ['auth', 'ui'], // persist auth tokens and ui preferences (adjust per your needs)
};

export function setupStore(preloadedState = undefined) {
  // Create reducer manager with static reducers
  const reducerManager = createReducerManager(staticReducers as any);

  // Create persist reducer for the root (so dynamic reducers that are added later won't be
persisted until they are added to reducerManager)
  const persistedReducer = persistReducer(persistConfig, reducerManager.getReducerMap ?
createRootReducer() : createRootReducer());

  const middlewares: Middleware[] = [thunk];

  // Add logger in non-production builds only
  if (process.env.NODE_ENV !== 'production') {
    middlewares.push(logger);
  }

  const store = configureStore({
    reducer: persistReducer(persistConfig, createRootReducer()), // root reducer
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          // ignore redux-persist actions
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      }).concat(middlewares),
    preloadedState,
    devTools: process.env.NODE_ENV !== 'production',
  });

  // Convert to enhanced store so we can attach reducerManager later
  const reducerMgr = createReducerManager(staticReducers as any);
  // NOTE: configureStore already created a reducer; for dynamic injection we attach a manager
  // We'll replace the reducer on the store to a managed reducer:
  // @ts-ignore - attach manager for runtime usage
  (store as EnhancedStore & { reducerManager?: any }).reducerManager = reducerMgr;
  store.replaceReducer(reducerMgr.reduce);

  const persistor = persistStore(store);

  return { store, persistor };
}

export type RootState = ReturnType<ReturnType<typeof createRootReducer>>;
export type AppStore = ReturnType<typeof setupStore>['store'];
export type AppDispatch = AppStore['dispatch'];
```

> Note: the above attaches a **`reducerManager`** to the store so you can inject/remove reducers at runtime (useful for code-splitting, microfrontends, or lazy features).

---

4) `src/store/hooks.ts` (typed hooks)

```ts
// src/store/hooks.ts
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux';
import type { AppDispatch, RootState } from './store';

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;
```

---

5) Helper to inject reducers at runtime (usage in feature module)

```ts
// src/store/injector.ts
import type { Reducer } from 'redux';
import type { AppStore } from './store';

export function injectReducer(store: AppStore, key: string, reducer: Reducer) {
  // @ts-ignore
  const rm = store.reducerManager;
  if (!rm) {
    // fallback - replace reducer directly (rare)
    const root = (store as any).reducer;
    store.replaceReducer(root);
    return;
  }
  rm.add(key, reducer);
  store.replaceReducer(rm.reduce);
}
```

Usage inside a route or lazy feature:

```ts
// in Feature entry file
import { injectReducer } from '@/store/injector';
import featureReducer from './featureSlice';
import { store } from '@/store';

injectReducer(store, 'feature', featureReducer);
```

---

6) `main.tsx` or `index.tsx` — wiring up the provider & persistor

```tsx
// src/main.tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import { PersistGate } from 'redux-persist/integration/react';
import { setupStore } from './store/store';
import App from './App';

const { store, persistor } = setupStore();

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <App />
      </PersistGate>
    </Provider>
  </React.StrictMode>
);
```

Production/security considerations & recommendations

1. **Limit what you persist** — only `auth` (refresh token id) and `ui` preferences are typical. Do not persist large caches or sensitive data (or encrypt it).
2. **Token storage** — consider storing auth tokens in **httpOnly cookies** (server-managed) rather than localStorage if you worry about XSS. If you must store tokens in localStorage, set short lifetimes and refresh tokens via secure flows.
3. **RTK Query** — for server data caching & automatic invalidation, consider adding RTK Query slices (they integrate with Redux Toolkit and require very little config).
4. **Feature/module lazy loading** — use `injectReducer` to load feature reducers only when route/component is mounted. This reduces initial bundle size.
5. **Middleware** — add monitoring middleware (Sentry, custom metrics) in production. Keep `redux-logger` to dev only (we did).
6. **Type safety** — export `AppDispatch` and `RootState` from a single `store` entry so the rest of app can import types.
7. **Testing** — mock the store setup for unit & integration tests. Provide a `createTestStore` factory that accepts preloaded state and optional reducers.
8. **Persist rehydration** — handle `PersistGate` fallback UI to avoid layout flash; use `disablePersist` flag for private/incognito flows.
9. **Storage strategy** — for server-side rendering or SSR frameworks you'll need an alternate storage adapter (e.g., `redux-persist` with cookie storage). localStorage works for SPA.

Short checklist to run

1. `npm install @reduxjs/toolkit react-redux redux-persist redux-logger redux-thunk`

2. Add the files above and update imports paths to match your project.
3. Ensure slices `export default` reducer functions (e.g., `export default authSlice.reducer` or if you use `createSlice({ name, initialState, reducers })` then `export default slice.reducer;`).
4. Start app and verify persisted state (open devtools -> Application -> Local Storage).

If you want, I can:

- convert this to a version that uses **RTK Query** and demonstrate caching + invalidation, or
- add an example **feature lazy route** showing `injectReducer` in action, or
- produce a minimal `createTestStore` factory for unit tests.

Which one do you want next?

RTK Query and demonstrate caching + invalidation, or add tis

Perfect 👍 Let's extend your scalable store with **RTK Query** (built into Redux Toolkit).
This gives you:

- API abstraction with auto-generated hooks (`useGetXQuery`, `useAddXMutation`, etc.)
- Built-in **caching**, **automatic re-fetching**, and **invalidation**
- Middleware already optimized for serializable checks

---

1. Install RTK Query dependencies

bash

```bash
npm install @reduxjs/toolkit react-redux
```

(Already installed in your case if you're using Redux Toolkit, but good to confirm.)

---

2. Create an API Slice

Make a `src/services/api.ts` file:

ts

```ts
// src/services/api.ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

// Base API service
export const api = createApi({
  reducerPath: 'api', // appears in Redux state as state.api
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Categories', 'Users', 'Orders'],
  endpoints: (builder) => ({
    // Example: fetch products
    getProducts: builder.query<any[], void>({
      query: () => '/products',
      providesTags: ['Products'],
    }),

    // Example: fetch single product
    getProductById: builder.query<any, string>({
      query: (id) => `/products/${id}`,
      providesTags: (result, error, id) => [{ type: 'Products', id }],
    }),

    // Example: add product
    addProduct: builder.mutation<any, { name: string; price: number }>({
      query: (body) => ({
        url: '/products',
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Products'], // refetch products after adding
    }),

    // Example: update product
    updateProduct: builder.mutation<any, { id: string; data: any }>({
      query: ({ id, data }) => ({
        url: `/products/${id}`,
        method: 'PUT',
        body: data,
      }),
      invalidatesTags: (result, error, { id }) => [
        { type: 'Products', id },
        'Products',
      ],
    }),
  }),
})

// Auto-generated hooks
export const {
  useGetProductsQuery,
  useGetProductByIdQuery,
  useAddProductMutation,
  useUpdateProductMutation,
} = api
```

---

3. Add `api.reducer` and `api.middleware` to the store

Update your `store.ts`:

```ts
import { configureStore } from '@reduxjs/toolkit'
import { persistStore, persistReducer, FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER } from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import { api } from '../services/api' // 👈 import api
import { createRootReducer, staticReducers } from './rootReducer'
import { createReducerManager } from './reducerManager'

// persist config
const persistConfig = {
  key: 'root',
  version: 1,
  storage,
  whitelist: ['auth', 'ui'],
}

export function setupStore(preloadedState = undefined) {
  const reducerManager = createReducerManager({
    ...staticReducers,
    [api.reducerPath]: api.reducer, // 👈 add api reducer
  })

  const persistedReducer = persistReducer(persistConfig, reducerManager.reduce)

  const store = configureStore({
    reducer: persistedReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      }).concat(api.middleware), // 👈 add api middleware
    preloadedState,
    devTools: process.env.NODE_ENV !== 'production',
  })

  // attach reducer manager for dynamic reducers
  ;(store as any).reducerManager = reducerManager
  store.replaceReducer(reducerManager.reduce)

  const persistor = persistStore(store)

  return { store, persistor }
}

export type RootState = ReturnType<ReturnType<typeof createRootReducer>>
export type AppDispatch = ReturnType<typeof setupStore>['store']['dispatch']
```

---

4. Use in Components

Now you can fetch, cache, and mutate data with auto hooks:

```tsx
import React from 'react'
import { useGetProductsQuery, useAddProductMutation } from '@/services/api'

export function Products() {
  const { data: products, isLoading, error } = useGetProductsQuery()
  const [addProduct] = useAddProductMutation()

  if (isLoading) return <p>Loading...</p>
  if (error) return <p>Error fetching products</p>

  return (
    <div>
      <h1>Products</h1>
      <ul>
        {products?.map((p) => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>
      <button
        onClick={() =>
          addProduct({ name: 'New Product', price: 100 })
        }
      >
        Add Product
      </button>
    </div>
  )
}
```

---

5. How caching + invalidation works here

- `useGetProductsQuery()` caches the response under the key `Products`.
- When you call `useAddProductMutation()`, it **invalidates** the `Products` tag.
- RTK Query automatically **refetches** the `getProducts` query → cache updates → UI updates.
- Queries with `providesTags` + `invalidatesTags` enable this lifecycle.

⚡ This setup is **enterprise-ready** because:

- RTK Query handles caching, invalidation, polling, and re-fetching automatically.
- Combined with `redux-persist`, you persist only UI/auth — **not API cache** (good practice).
- `injectReducer` still works for feature slices, and you can also `injectEndpoints` dynamically into RTK Query if needed.

👉 Do you also want me to show you how to **inject RTK Query endpoints dynamically** (so features can register their own API calls lazily like reducers)?

give full code now with everything with files names

Perfect 👍 Let's put everything together into a **full project structure** with all files you need:

- **Redux Toolkit store** with persistence
- **Dynamic reducer injection**
- **RTK Query** with caching + invalidation
- **Typed hooks**
- **Usage in `main.tsx`**

📂 Project Structure

```css
src/
 ├── App.tsx
 ├── main.tsx
 ├── services/
 │    └── api.ts
 ├── slices/
 │    ├── authSlice.ts
 │    ├── categoriesSlice.ts
 │    ├── productsSlice.ts
 │    ├── ordersSlice.ts
 │    ├── usersSlice.ts
 │    ├── offersSlice.ts
 │    ├── couponsSlice.ts
 │    └── uiSlice.ts
 └── store/
      ├── rootReducer.ts
      ├── reducerManager.ts
      ├── store.ts
      ├── hooks.ts
      └── injector.ts
```

---

1) `src/store/rootReducer.ts`

ts

```ts
import { combineReducers } from '@reduxjs/toolkit'
import authSlice from '../slices/authSlice'
import categoriesSlice from '../slices/categoriesSlice'
import productsSlice from '../slices/productsSlice'
import ordersSlice from '../slices/ordersSlice'
import usersSlice from '../slices/usersSlice'
import offersSlice from '../slices/offersSlice'
import couponsSlice from '../slices/couponsSlice'
import uiSlice from '../slices/uiSlice'

export const staticReducers = {
  auth: authSlice,
  categories: categoriesSlice,
  products: productsSlice,
  orders: ordersSlice,
  users: usersSlice,
  offers: offersSlice,
  coupons: couponsSlice,
  ui: uiSlice,
}

export const createRootReducer = (asyncReducers = {}) =>
  combineReducers({
    ...staticReducers,
    ...asyncReducers,
  })

export type StaticReducers = typeof staticReducers
```

---

2) `src/store/reducerManager.ts`

ts

```ts
import { Reducer, AnyAction } from 'redux'
import { createRootReducer } from './rootReducer'

export function createReducerManager(initialReducers: Record<string, Reducer>) {
  let reducers = { ...initialReducers }
  let combinedReducer = createRootReducer(initialReducers)
  const keysToRemove: string[] = []

  return {
    getReducerMap: () => reducers,

    reduce: (state: any, action: AnyAction) => {
      if (keysToRemove.length > 0) {
        state = { ...state }
        for (const key of keysToRemove) {
          delete state[key]
        }
        keysToRemove.length = 0
      }
      return combinedReducer(state, action)
    },

    add: (key: string, reducer: Reducer) => {
      if (!key || reducers[key]) return
      reducers = { ...reducers, [key]: reducer }
      combinedReducer = createRootReducer(reducers)
    },

    remove: (key: string) => {
      if (!key || !reducers[key]) return
      delete reducers[key]
      keysToRemove.push(key)
      combinedReducer = createRootReducer(reducers)
    },
  }
}
```

---

3) `src/store/store.ts`

```ts
import { configureStore } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import logger from 'redux-logger'
import thunk from 'redux-thunk'
import { api } from '../services/api'
import { createRootReducer, staticReducers } from './rootReducer'
import { createReducerManager } from './reducerManager'

const persistConfig = {
  key: 'root',
  version: 1,
  storage,
  whitelist: ['auth', 'ui'], // persist only these slices
}

export function setupStore(preloadedState = undefined) {
  const reducerManager = createReducerManager({
    ...staticReducers,
    [api.reducerPath]: api.reducer, // attach RTK Query reducer
  })

  const persistedReducer = persistReducer(persistConfig, reducerManager.reduce)

  const store = configureStore({
    reducer: persistedReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      })
        .concat(thunk)
        .concat(api.middleware)
        .concat(process.env.NODE_ENV !== 'production' ? [logger] : []),
    preloadedState,
    devTools: process.env.NODE_ENV !== 'production',
  })

  ;(store as any).reducerManager = reducerManager
  store.replaceReducer(reducerManager.reduce)

  const persistor = persistStore(store)
  return { store, persistor }
}

export type RootState = ReturnType<ReturnType<typeof createRootReducer>>
export type AppStore = ReturnType<typeof setupStore>['store']
export type AppDispatch = AppStore['dispatch']
```

---

4) `src/store/hooks.ts`

```ts
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux'
import type { AppDispatch, RootState } from './store'

export const useAppDispatch = () => useDispatch<AppDispatch>()
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector
```

---

5) `src/store/injector.ts`

```ts
import type { Reducer } from 'redux'
import type { AppStore } from './store'

export function injectReducer(store: AppStore, key: string, reducer: Reducer) {
  // @ts-ignore
  const rm = store.reducerManager
  if (!rm) return
  rm.add(key, reducer)
  store.replaceReducer(rm.reduce)
}
```

---

6) `src/services/api.ts`

```ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Categories', 'Users', 'Orders'],
  endpoints: (builder) => ({
    getProducts: builder.query<any[], void>({
      query: () => '/products',
      providesTags: ['Products'],
    }),

    getProductById: builder.query<any, string>({
      query: (id) => `/products/${id}`,
      providesTags: (result, error, id) => [{ type: 'Products', id }],
    }),

    addProduct: builder.mutation<any, { name: string; price: number }>({
      query: (body) => ({
        url: '/products',
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Products'],
    }),

    updateProduct: builder.mutation<any, { id: string; data: any }>({
      query: ({ id, data }) => ({
        url: `/products/${id}`,
        method: 'PUT',
        body: data,
      }),
      invalidatesTags: (result, error, { id }) => [
        { type: 'Products', id },
        'Products',
      ],
    }),
  }),
})

export const {
  useGetProductsQuery,
  useGetProductByIdQuery,
  useAddProductMutation,
  useUpdateProductMutation,
} = api
```

---

7) Example Slice (`src/slices/authSlice.ts`)

```
ts

import { createSlice, PayloadAction } from '@reduxjs/toolkit'

interface AuthState {
  token: string | null
  user: { id: string; name: string } | null
}

const initialState: AuthState = {
  token: null,
  user: null,
}

const authSlice = createSlice({
  name: 'auth',
  initialState,
  reducers: {
    loginSuccess(state, action: PayloadAction<{ token: string; user: any }>) {
      state.token = action.payload.token
      state.user = action.payload.user
    },
    logout(state) {
      state.token = null
      state.user = null
    },
  },
})

export const { loginSuccess, logout } = authSlice.actions
export default authSlice.reducer
```

(Other slices like **`productsSlice.ts`**, **`ordersSlice.ts`** etc. follow the same pattern.)

---

8) `src/App.tsx`

```tsx
import React from 'react'
import { useGetProductsQuery, useAddProductMutation } from './services/api'

function App() {
  const { data: products, isLoading } = useGetProductsQuery()
  const [addProduct] = useAddProductMutation()

  if (isLoading) return <p>Loading...</p>

  return (
    <div>
      <h1>Products</h1>
      <ul>
        {products?.map((p) => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>

      <button
        onClick={() =>
          addProduct({ name: 'New Product', price: 99 })
        }
      >
        Add Product
      </button>
    </div>
  )
}

export default App
```

---

9) `src/main.tsx`

```tsx
import React from 'react'
import ReactDOM from 'react-dom/client'
import { Provider } from 'react-redux'
import { PersistGate } from 'redux-persist/integration/react'
import { setupStore } from './store/store'
import App from './App'

const { store, persistor } = setupStore()

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <App />
      </PersistGate>
    </Provider>
  </React.StrictMode>
)
```

✅ This gives you:

- **Scalable store** (dynamic reducers + slices + RTK Query)
- **Persistence** with **`redux-persist`**
- **Typed hooks**
- **Caching + invalidation** using RTK Query

Do you also want me to include an **example of dynamically injecting a feature slice + API endpoints** (so modules register themselves only when loaded)?

> Do you also want me to include an example of dynamically injecting a feature slice + API endpoints (so modules register themselves only when loaded)?
>
> give this to

Got it 👍 Let's extend the scalable Redux Toolkit setup so that **feature slices and RTK Query endpoints can be injected dynamically** (only when a feature/module is loaded). This keeps the initial bundle small and makes your app enterprise-ready.

📂 Final Project Structure

css

```css
src/
├─ app/
│  ├─ store.ts
│  ├─ reducerManager.ts
│  └─ rootReducer.ts
├─ features/
│  ├─ auth/
│  │  └─ authSlice.ts
│  └─ products/
│     ├─ productsSlice.ts
│     └─ productsApi.ts
├─ services/
│  └─ api.ts
└─ main.tsx
```

---

1. `src/app/reducerManager.ts`

ts

```ts
import { combineReducers, Reducer, AnyAction } from '@reduxjs/toolkit'

export function createReducerManager(initialReducers: Record<string, Reducer>) {
  const reducers = { ...initialReducers }
  let combinedReducer = combineReducers(reducers)

  const keysToRemove: string[] = []

  return {
    getReducerMap: () => reducers,

    reduce: (state: any, action: AnyAction) => {
      if (keysToRemove.length > 0) {
        state = { ...state }
        for (let key of keysToRemove) {
          delete state[key]
        }
        keysToRemove.length = 0
      }
      return combinedReducer(state, action)
    },

    add: (key: string, reducer: Reducer) => {
      if (!key || reducers[key]) return
      reducers[key] = reducer
      combinedReducer = combineReducers(reducers)
    },

    remove: (key: string) => {
      if (!key || !reducers[key]) return
      delete reducers[key]
      keysToRemove.push(key)
      combinedReducer = combineReducers(reducers)
    },
  }
}
```

---

2. `src/app/rootReducer.ts`

```ts
import authSlice from '@/features/auth/authSlice'
import uiSlice from '@/features/ui/uiSlice'

export const staticReducers = {
  auth: authSlice,
  ui: uiSlice,
}

export function createRootReducer() {
  return staticReducers
}
```

---

3. `src/services/api.ts` (base API)

```ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Categories', 'Users', 'Orders'],
  endpoints: () => ({}), // no static endpoints here
})
```

---

4. `src/app/store.ts`

```ts
import { configureStore } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import { api } from '@/services/api'
import { staticReducers, createRootReducer } from './rootReducer'
import { createReducerManager } from './reducerManager'

const persistConfig = {
  key: 'root',
  version: 1,
  storage,
  whitelist: ['auth', 'ui'],
}

export function setupStore(preloadedState = undefined) {
  const reducerManager = createReducerManager({
    ...staticReducers,
    [api.reducerPath]: api.reducer,
  })

  const persistedReducer = persistReducer(persistConfig, reducerManager.reduce)

  const store = configureStore({
    reducer: persistedReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      }).concat(api.middleware),
    preloadedState,
    devTools: process.env.NODE_ENV !== 'production',
  })

  // expose reducerManager
  ;(store as any).reducerManager = reducerManager

  const persistor = persistStore(store)

  return { store, persistor }
}

export type RootState = ReturnType<ReturnType<typeof createRootReducer>>
export type AppDispatch = ReturnType<typeof setupStore>['store']['dispatch']
```

---

5. `src/features/auth/authSlice.ts`

```
ts

import { createSlice } from '@reduxjs/toolkit'

const authSlice = createSlice({
  name: 'auth',
  initialState: { user: null, token: null },
  reducers: {
    setUser: (state, action) => {
      state.user = action.payload
    },
    logout: (state) => {
      state.user = null
      state.token = null
    },
  },
})

export const { setUser, logout } = authSlice.actions
export default authSlice.reducer
```

---

6. `src/features/products/productsSlice.ts`

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

const productsSlice = createSlice({
  name: 'products',
  initialState: { items: [] },
  reducers: {
    setProducts: (state, action) => {
      state.items = action.payload
    },
  },
})

export const { setProducts } = productsSlice.actions
export default productsSlice.reducer
```

---

7. `src/features/products/productsApi.ts`

👉 Example of **injecting endpoints dynamically**.

```ts
import { api } from '@/services/api'

export const productsApi = api.injectEndpoints({
  endpoints: (builder) => ({
    getProducts: builder.query<any[], void>({
      query: () => '/products',
      providesTags: ['Products'],
    }),
    getProductById: builder.query<any, string>({
      query: (id) => `/products/${id}`,
      providesTags: (result, error, id) => [{ type: 'Products', id }],
    }),
    addProduct: builder.mutation<any, { name: string; price: number }>({
      query: (body) => ({
        url: '/products',
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Products'],
    }),
  }),
  overrideExisting: false,
})

export const {
  useGetProductsQuery,
  useGetProductByIdQuery,
  useAddProductMutation,
} = productsApi
```

---

8. `src/main.tsx`

```tsx
import React from 'react'
import ReactDOM from 'react-dom/client'
import { Provider } from 'react-redux'
import { PersistGate } from 'redux-persist/integration/react'
import { setupStore } from '@/app/store'
import App from './App'

const { store, persistor } = setupStore()

ReactDOM.createRoot(document.getElementById('root')!).render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>
)
```

---

9. Example Component using dynamic endpoints

```tsx
tsx

import React from 'react'
import {
  useGetProductsQuery,
  useAddProductMutation,
} from '@/features/products/productsApi'

export function Products() {
  const { data, isLoading, error } = useGetProductsQuery()
  const [addProduct] = useAddProductMutation()

  if (isLoading) return <p>Loading...</p>
  if (error) return <p>Error fetching products</p>

  return (
    <div>
      <h2>Products</h2>
      <ul>
        {data?.map((p) => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>
      <button onClick={() => addProduct({ name: 'New Product', price: 100 })}>
        Add Product
      </button>
    </div>
  )
}
```

✅ With this setup:

- `auth` + `ui` are static reducers.
- `productsSlice` can be injected dynamically via `store.reducerManager.add('products', productsSlice)`.
- API endpoints (`productsApi`) are injected into the **shared base `api` slice** → queries & mutations are auto-wired with caching + invalidation.
- Cache is not persisted (good practice), but `auth` + `ui` states are persisted.

Do you also want me to include an example of **lazy loading a feature module** (e.g., when navigating to `/products`, the reducer + API endpoints get injected automatically)?

Printed using ChatGPT to PDF, powered by PDFCrowd HTML to PDF API.

sure

Perfect 👍 Let’s make it **lazy-loadable**, so when a user navigates to a route (like `/products`), we dynamically inject the **slice + API endpoints** into the store.

This pattern is super useful in enterprise apps with **code-splitting**.

📂 Updated Structure

bash

```
src/
 ├─ app/
 │   ├─ store.ts
 │   ├─ reducerManager.ts
 │   └─ rootReducer.ts
 ├─ features/
 │   ├─ auth/authSlice.ts
 │   ├─ products/
 │   │   ├─ index.tsx       👈 lazy-loaded entry
 │   │   ├─ productsSlice.ts
 │   │   └─ productsApi.ts
 ├─ services/api.ts
 ├─ App.tsx
 └─ main.tsx
```

---

1. Dynamic Injection Utility

`src/app/store.ts` → add this helper:

ts

```
// Dynamic reducer injection
export function injectReducer(store: any, key: string, reducer: any) {
  if (!(store as any).reducerManager.getReducerMap()[key]) {
    (store as any).reducerManager.add(key, reducer)
    store.replaceReducer((store as any).reducerManager.reduce)
  }
}
```

---

2. Products Feature Entry (Lazy)

`src/features/products/index.tsx`

tsx

```
import React, { useEffect } from 'react'
import { useGetProductsQuery, useAddProductMutation } from './productsApi'
import productsSlice from './productsSlice'
import { injectReducer } from '@/app/store'
import { useStore } from 'react-redux'

export default function ProductsPage() {
  const store = useStore()

  // Inject reducer only when component is mounted
  useEffect(() => {
    injectReducer(store, 'products', productsSlice)
  }, [store])

  const { data, isLoading, error } = useGetProductsQuery()
  const [addProduct] = useAddProductMutation()

  if (isLoading) return <p>Loading...</p>
  if (error) return <p>Error loading products</p>

  return (
    <div>
      <h2>Products</h2>
      <ul>
        {data?.map((p) => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>
      <button onClick={() => addProduct({ name: 'New Product', price: 100 })}>
        Add Product
      </button>
    </div>
  )
}
```

---

3. App Router with Lazy Loading

`src/App.tsx`

```tsx
import React, { Suspense, lazy } from 'react'
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom'

const ProductsPage = lazy(() => import('@/features/products'))

export default function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> | <Link to="/products">Products</Link>
      </nav>
      <Suspense fallback={<p>Loading...</p>}>
        <Routes>
          <Route path="/" element={<h1>Home</h1>} />
          <Route path="/products" element={<ProductsPage />} />
        </Routes>
      </Suspense>
    </Router>
  )
}
```

---

4. Example Walkthrough 🚀

1. When the app loads → only `auth` + `ui` reducers + base `api` exist in store.
2. Navigate to `/products` → `ProductsPage` loads.
3. `injectReducer(store, 'products', productsSlice)` runs → now store has `products`.
4. `productsApi` endpoints are already injected into the shared `api`.

5. Products query runs, cache is stored, and UI updates.

---

5. Benefits

- ✅ **Code splitting**: only load what's needed.
- ✅ **Smaller bundle** at startup.
- ✅ **Scalable**: features register themselves, no central "god store" file.
- ✅ **Automatic RTK Query caching + invalidation** still works.

👉 Do you also want me to extend this with an example of **injecting API endpoints lazily** (not just reducers), so endpoints are registered *only* when the module is loaded?

sure

Nice 🚀 Let's go one step further: not only inject reducers dynamically, but also **inject RTK Query endpoints lazily**.

That means:

- When `/products` is visited → the `products` reducer + endpoints get registered.
- Until then, there's **no knowledge of products API** in the store.
- Great for huge enterprise apps with hundreds of APIs.

📂 Structure (extended)

```bash
src/
 ├─ app/
 │   ├─ store.ts
 │   ├─ reducerManager.ts
 │   └─ rootReducer.ts
 ├─ features/
 │   ├─ auth/authSlice.ts
 │   ├─ products/
 │   │   ├─ index.tsx
 │   │   ├─ productsSlice.ts
 │   │   └─ productsApi.ts
 ├─ services/api.ts   👈 base api only
 ├─ App.tsx
 └─ main.tsx
```

---

1. Base API (no endpoints)

`src/services/api.ts`

```ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Categories', 'Users', 'Orders'],
  endpoints: () => ({}), // 👈 no endpoints yet
})
```

---

2. Products API (lazy endpoints)

`src/features/products/productsApi.ts`

```ts
import { api } from '@/services/api'

// instead of export const productsApi = api.injectEndpoints(...)
let injected = false

export function useProductsApi() {
  if (!injected) {
    api.injectEndpoints({
      endpoints: (builder) => ({
        getProducts: builder.query<any[], void>({
          query: () => '/products',
          providesTags: ['Products'],
        }),
        getProductById: builder.query<any, string>({
          query: (id) => `/products/${id}`,
          providesTags: (result, error, id) => [{ type: 'Products', id }],
        }),
        addProduct: builder.mutation<any, { name: string; price: number }>({
          query: (body) => ({
            url: '/products',
            method: 'POST',
            body,
          }),
          invalidatesTags: ['Products'],
        }),
      }),
      overrideExisting: false,
    })
    injected = true
  }

  return {
    useGetProductsQuery: api.endpoints.getProducts.useQuery,
    useGetProductByIdQuery: api.endpoints.getProductById.useQuery,
    useAddProductMutation: api.endpoints.addProduct.useMutation,
  }
}
```

---

3. Products Feature (inject reducer + endpoints)

`src/features/products/index.tsx`

```tsx
import React, { useEffect } from 'react'
import { injectReducer } from '@/app/store'
import productsSlice from './productsSlice'
import { useStore } from 'react-redux'
import { useProductsApi } from './productsApi'

export default function ProductsPage() {
  const store = useStore()

  // Inject reducer + endpoints when loaded
  useEffect(() => {
    injectReducer(store, 'products', productsSlice)
  }, [store])

  const { useGetProductsQuery, useAddProductMutation } = useProductsApi()
  const { data, isLoading, error } = useGetProductsQuery()
  const [addProduct] = useAddProductMutation()

  if (isLoading) return <p>Loading...</p>
  if (error) return <p>Error fetching products</p>

  return (
    <div>
      <h2>Products</h2>
      <ul>
        {data?.map((p) => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>
      <button onClick={() => addProduct({ name: 'New Product', price: 100 })}>
        Add Product
      </button>
    </div>
  )
}
```

4. App Router (unchanged)

`src/App.tsx`

```tsx
import React, { Suspense, lazy } from 'react'
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom'

const ProductsPage = lazy(() => import('@/features/products'))

export default function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> | <Link to="/products">Products</Link>
      </nav>
      <Suspense fallback={<p>Loading...</p>}>
        <Routes>
          <Route path="/" element={<h1>Home</h1>} />
          <Route path="/products" element={<ProductsPage />} />
        </Routes>
      </Suspense>
    </Router>
  )
}
```

🔄 Flow

1. At app start:
    - Store has `auth`, `ui`, and empty `api` slice.
    - No products reducer, no products endpoints.
2. Navigate to `/products`:
    - `injectReducer(store, 'products', productsSlice)` adds reducer.
    - `useProductsApi()` registers endpoints into `api`.
    - Queries now work (`useGetProductsQuery`).
3. Leave `/products`:
    - Reducer can be removed if you want (`store.reducerManager.remove('products')`).
    - Endpoints remain registered (or you can design cleanup).

✅ Benefits

- Initial bundle stays tiny.
- Both **reducers and endpoints** are loaded only when needed.
- No central “API monster file”. Features self-register.
- Perfect for **micro-frontend** or **plugin-style architecture**.

👉 Do you also want me to show you how to **unload endpoints when leaving a route** (so API hooks don’t stick around in memory)?

sure

Perfect ⚡ now we’ll go **full dynamic** → not only inject reducers and endpoints lazily, but also **unload endpoints when leaving a route**.

This is useful for **huge apps** (hundreds of APIs) where you don’t want unused endpoints bloating memory.

📂 Structure (same)

```bash
src/
 ├─ app/
 │   ├─ store.ts
 │   ├─ reducerManager.ts
 │   └─ rootReducer.ts
 ├─ features/
 │   ├─ auth/authSlice.ts
 │   ├─ products/
 │   │  ├─ index.tsx
 │   │  ├─ productsSlice.ts
 │   │  └─ productsApi.ts
 ├─ services/api.ts
 ├─ App.tsx
 └─ main.tsx
```

---

1. Base API stays the same

`src/services/api.ts`

```ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Categories', 'Users', 'Orders'],
  endpoints: () => ({}), // initially empty
})
```

---

2. Products API with unload support

`src/features/products/productsApi.ts`

```ts
import { api } from '@/services/api'

let injected = false
let removeEndpoints: null | (() => void) = null

export function useProductsApi() {
  if (!injected) {
    const injectedApi = api.injectEndpoints({
      endpoints: (builder) => ({
        getProducts: builder.query<any[], void>({
          query: () => '/products',
          providesTags: ['Products'],
        }),
        getProductById: builder.query<any, string>({
          query: (id) => `/products/${id}`,
          providesTags: (result, error, id) => [{ type: 'Products', id }],
        }),
        addProduct: builder.mutation<any, { name: string; price: number }>({
          query: (body) => ({
            url: '/products',
            method: 'POST',
            body,
          }),
          invalidatesTags: ['Products'],
        }),
      }),
      overrideExisting: false,
    })

    injected = true
    removeEndpoints = injectedApi.util.removeEndpoints // 👈 capture cleanup fn
  }

  return {
    useGetProductsQuery: api.endpoints.getProducts.useQuery,
    useGetProductByIdQuery: api.endpoints.getProductById.useQuery,
    useAddProductMutation: api.endpoints.addProduct.useMutation,
  }
}

// expose cleanup
export function unloadProductsApi() {
  if (removeEndpoints) {
    removeEndpoints(['getProducts', 'getProductById', 'addProduct'])
    injected = false
    removeEndpoints = null
  }
}
```

---

3. Products Page with cleanup

`src/features/products/index.tsx`

```tsx
import React, { useEffect } from 'react'
import { useStore } from 'react-redux'
import { injectReducer } from '@/app/store'
import productsSlice from './productsSlice'
import { useProductsApi, unloadProductsApi } from './productsApi'

export default function ProductsPage() {
  const store = useStore()

  // Inject reducer + endpoints when mounted
  useEffect(() => {
    injectReducer(store, 'products', productsSlice)
    return () => {
      // Optional: remove reducer when leaving route
      store.reducerManager.remove('products')
      store.replaceReducer(store.reducerManager.reduce)

      // Unload endpoints
      unloadProductsApi()
    }
  }, [store])

  const { useGetProductsQuery, useAddProductMutation } = useProductsApi()
  const { data, isLoading, error } = useGetProductsQuery()
  const [addProduct] = useAddProductMutation()

  if (isLoading) return <p>Loading...</p>
  if (error) return <p>Error fetching products</p>

  return (
    <div>
      <h2>Products</h2>
      <ul>
        {data?.map((p) => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>
      <button onClick={() => addProduct({ name: 'New Product', price: 100 })}>
        Add Product
      </button>
    </div>
  )
}
```

🔄 Lifecycle

1. Navigate to `/products`:
    - Reducer + endpoints are injected.
    - API hooks (`useGetProductsQuery`) work.
2. Leave `/products`:
    - Reducer is removed.
    - Endpoints are unloaded via `removeEndpoints`.
    - Cache and subscriptions are cleared → memory freed.

✅ Benefits

- Ultra-scalable, plugin-style architecture.
- Reducers + endpoints exist **only when needed**.
- Perfect for **micro-frontends**, multi-tenant dashboards, or SaaS apps with dynamic modules.

👉 Do you also want me to extend this with an example of **multi-feature lazy modules** (e.g. `/products` + `/orders`, both injecting their own reducers + APIs independently)?

sure

Perfect ✅ Let's extend the setup to handle **multiple lazy-loaded features** (`/products` + `/orders`) where each feature:

- Dynamically injects its **slice**
- Dynamically injects its **API endpoints**
- Unloads both when the route is left

This is **enterprise-ready**, micro-frontend friendly, and memory-efficient.

📂 Example Structure

```bash
src/
 ├─ app/
 │   ├─ store.ts
 │   ├─ reducerManager.ts
 │   └─ rootReducer.ts
 ├─ features/
 │   ├─ auth/authSlice.ts
 │   ├─ products/
 │   │   ├─ index.tsx
```

```
  │       │   ├─ productsSlice.ts
  │       │   └─ productsApi.ts
  │       └─ orders/
  │           ├─ index.tsx
  │           ├─ ordersSlice.ts
  │           └─ ordersApi.ts
  ├─ services/api.ts
  ├─ App.tsx
  └─ main.tsx
```

---

1. Orders API (lazy endpoints)

`src/features/orders/ordersApi.ts`

```ts
import { api } from '@/services/api'

let injected = false
let removeEndpoints: null | (() => void) = null

export function useOrdersApi() {
  if (!injected) {
    const injectedApi = api.injectEndpoints({
      endpoints: (builder) => ({
        getOrders: builder.query<any[], void>({
          query: () => '/orders',
          providesTags: ['Orders'],
        }),
        addOrder: builder.mutation<any, { userId: string; items: any[] }>({
          query: (body) => ({
            url: '/orders',
            method: 'POST',
            body,
          }),
          invalidatesTags: ['Orders'],
        }),
      }),
      overrideExisting: false,
    })

    injected = true
    removeEndpoints = injectedApi.util.removeEndpoints
  }

  return {
    useGetOrdersQuery: api.endpoints.getOrders.useQuery,
    useAddOrderMutation: api.endpoints.addOrder.useMutation,
  }
}

export function unloadOrdersApi() {
  if (removeEndpoints) {
    removeEndpoints(['getOrders', 'addOrder'])
    injected = false
    removeEndpoints = null
  }
}
```

---

2. Orders Slice

`src/features/orders/ordersSlice.ts`

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

const ordersSlice = createSlice({
  name: 'orders',
  initialState: { items: [] },
  reducers: {
    setOrders: (state, action) => {
      state.items = action.payload
    },
  },
})

export const { setOrders } = ordersSlice.actions
export default ordersSlice.reducer
```

---

3. Orders Page (lazy + cleanup)

`src/features/orders/index.tsx`

```tsx
import React, { useEffect } from 'react'
import { useStore } from 'react-redux'
import { injectReducer } from '@/app/store'
import ordersSlice from './ordersSlice'
import { useOrdersApi, unloadOrdersApi } from './ordersApi'

export default function OrdersPage() {
  const store = useStore()

  useEffect(() => {
    injectReducer(store, 'orders', ordersSlice)

    return () => {
      store.reducerManager.remove('orders')
      store.replaceReducer(store.reducerManager.reduce)
      unloadOrdersApi()
    }
  }, [store])

  const { useGetOrdersQuery, useAddOrderMutation } = useOrdersApi()
  const { data, isLoading, error } = useGetOrdersQuery()
  const [addOrder] = useAddOrderMutation()

  if (isLoading) return <p>Loading...</p>
  if (error) return <p>Error loading orders</p>

  return (
    <div>
      <h2>Orders</h2>
      <ul>
        {data?.map((o) => (
          <li key={o.id}>Order #{o.id}</li>
        ))}
      </ul>
      <button
        onClick={() =>
          addOrder({ userId: '1', items: [{ id: 'p1', qty: 2 }] })
        }
      >
        Add Order
      </button>
    </div>
  )
}
```

4. App Router with Multiple Lazy Modules

`src/App.tsx`

```tsx
import React, { Suspense, lazy } from 'react'
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom'

const ProductsPage = lazy(() => import('@/features/products'))
const OrdersPage = lazy(() => import('@/features/orders'))

export default function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/products">Products</Link> |
        <Link to="/orders">Orders</Link>
      </nav>
      <Suspense fallback={<p>Loading module...</p>}>
        <Routes>
          <Route path="/" element={<h1>Home</h1>} />
          <Route path="/products" element={<ProductsPage />} />
          <Route path="/orders" element={<OrdersPage />} />
        </Routes>
      </Suspense>
    </Router>
  )
}
```

✅ How it works

Action	Effect
Navigate to **`/products`**	Inject **`products`** reducer + endpoints
Leave **`/products`**	Remove reducer + unload endpoints
Navigate to **`/orders`**	Inject **`orders`** reducer + endpoints
Leave **`/orders`**	Remove reducer + unload endpoints

- Each module self-registers and self-unloads.
- Cache and subscriptions are cleared when endpoints are unloaded.
- Base **`api`** slice + store remain intact.

This pattern is **fully enterprise-ready**:

- Supports hundreds of features without bloating store.
- Works with **React.lazy + Suspense**.
- RTK Query caching + invalidation still works per feature.

If you want, I can make a **complete ready-to-run Vite + React + Redux Toolkit template** with **dynamic lazy-loaded features, reducers, and RTK Query endpoints**, so you can just clone and start coding.

Do you want me to do that next?

yes

Perfect! Let's create a **full ready-to-run Vite + React + Redux Toolkit + RTK Query template** with **dynamic lazy-loaded features**, reducers, and endpoints.

This template will include:

- Vite + React setup
- Redux Toolkit store with `reducerManager`
- `redux-persist` for selected slices
- Base RTK Query `api` slice
- Dynamic injection/unloading of feature slices + endpoints
- Lazy-loaded routes (`/products`, `/orders`)

1️⃣ Project Setup

```bash
# Create Vite React project
npm create vite@latest my-enterprise-app -- --template react-ts
cd my-enterprise-app

# Install dependencies
npm install @reduxjs/toolkit react-redux redux-persist react-router-dom
```

2️⃣ Project Structure

```css
src/
 ├─ app/
 │   ├─ store.ts
 │   ├─ reducerManager.ts
 │   └─ rootReducer.ts
 ├─ services/
 │   └─ api.ts
 ├─ features/
 │   ├─ auth/authSlice.ts
 │   ├─ products/
 │   │   ├─ index.tsx
 │   │   ├─ productsSlice.ts
 │   │   └─ productsApi.ts
 │   └─ orders/
 │       ├─ index.tsx
 │       ├─ ordersSlice.ts
 │       └─ ordersApi.ts
 ├─ App.tsx
 └─ main.tsx
```

3️⃣ Base API

`src/services/api.ts`

```ts
ts

import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Orders'],
  endpoints: () => ({}),
})
```

4️⃣ Reducer Manager

`src/app/reducerManager.ts`

```ts
ts

import { combineReducers, Reducer, AnyAction } from '@reduxjs/toolkit'

export function createReducerManager(initialReducers: Record<string, Reducer>) {
  let reducers = { ...initialReducers }
  let combinedReducer = combineReducers(reducers)
  const keysToRemove: string[] = []

  return {
    getReducerMap: () => reducers,
    reduce: (state: any, action: AnyAction) => {
      if (keysToRemove.length) {
        state = { ...state }
        keysToRemove.forEach((key) => delete state[key])
        keysToRemove.length = 0
      }
      return combinedReducer(state, action)
    },
    add: (key: string, reducer: Reducer) => {
      if (!key || reducers[key]) return
      reducers[key] = reducer
      combinedReducer = combineReducers(reducers)
    },
    remove: (key: string) => {
      if (!key || !reducers[key]) return
      delete reducers[key]
      keysToRemove.push(key)
      combinedReducer = combineReducers(reducers)
    },
  }
}
```

5️⃣ Root Reducer

`src/app/rootReducer.ts`

```ts
ts

import authSlice from '@/features/auth/authSlice'
```

```
export const staticReducers = {
  auth: authSlice,
}

export const createRootReducer = (asyncReducers = {}) => ({
  ...staticReducers,
  ...asyncReducers,
})
```

6️⃣ Store

`src/app/store.ts`

```ts
import { configureStore } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import { api } from '@/services/api'
import { staticReducers, createRootReducer } from './rootReducer'
import { createReducerManager } from './reducerManager'

const persistConfig = {
  key: 'root',
  storage,
  whitelist: ['auth'],
}

export function setupStore(preloadedState = undefined) {
  const reducerManager = createReducerManager({
    ...staticReducers,
    [api.reducerPath]: api.reducer,
  })

  const persistedReducer = persistReducer(persistConfig, reducerManager.reduce)

  const store = configureStore({
    reducer: persistedReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      }).concat(api.middleware),
    preloadedState,
    devTools: true,
  })

  ;(store as any).reducerManager = reducerManager
  const persistor = persistStore(store)
  return { store, persistor }
}

export type AppStore = ReturnType<typeof setupStore>['store']
export type AppDispatch = AppStore['dispatch']

// Dynamic reducer injection
export function injectReducer(store: AppStore, key: string, reducer: any) {
  const rm = (store as any).reducerManager
  if (!rm.getReducerMap()[key]) {
    rm.add(key, reducer)
    store.replaceReducer(rm.reduce)
  }
}
```

7️⃣ Features

Auth Slice (`src/features/auth/authSlice.ts`)

```
ts

import { createSlice } from '@reduxjs/toolkit'

const authSlice = createSlice({
  name: 'auth',
  initialState: { user: null, token: null },
  reducers: {
    setUser: (state, action) => { state.user = action.payload },
    logout: (state) => { state.user = null; state.token = null },
  },
})

export const { setUser, logout } = authSlice.actions
export default authSlice.reducer
```

Products Slice + API (`features/products`)

- **`productsSlice.ts`**

```
ts

import { createSlice } from '@reduxjs/toolkit'

const productsSlice = createSlice({
  name: 'products',
  initialState: { items: [] },
  reducers: { setProducts: (state, action) => { state.items = action.payload } },
})

export default productsSlice.reducer
```

- **`productsApi.ts`**

```
ts

import { api } from '@/services/api'

let injected = false
let removeEndpoints: null | (() => void) = null

export function useProductsApi() {
  if (!injected) {
    const injectedApi = api.injectEndpoints({
      endpoints: (builder) => ({
        getProducts: builder.query<any[], void>({
          query: () => '/products',
          providesTags: ['Products'],
        }),
      }),
      overrideExisting: false,
    })
    injected = true
    removeEndpoints = injectedApi.util.removeEndpoints
  }

  return { useGetProductsQuery: api.endpoints.getProducts.useQuery }
}

export function unloadProductsApi() {
  if (removeEndpoints) {
    removeEndpoints(['getProducts'])
    injected = false
    removeEndpoints = null
  }
}
```

- **`index.tsx`** (lazy-loaded page)

```tsx
import React, { useEffect } from 'react'
import { useStore } from 'react-redux'
import productsSlice from './productsSlice'
import { injectReducer } from '@/app/store'
import { useProductsApi, unloadProductsApi } from './productsApi'

export default function ProductsPage() {
  const store = useStore()
  useEffect(() => {
    injectReducer(store, 'products', productsSlice)
    return () => {
      store.reducerManager.remove('products')
      store.replaceReducer(store.reducerManager.reduce)
      unloadProductsApi()
    }
  }, [store])

  const { useGetProductsQuery } = useProductsApi()
  const { data, isLoading } = useGetProductsQuery()

  if (isLoading) return <p>Loading...</p>
  return <ul>{data?.map(p => <li key={p.id}>{p.name}</li>)}</ul>
}
```

Orders Slice + API (`features/orders`) – same pattern

- **`ordersSlice.ts`**, **`ordersApi.ts`**, **`index.tsx`** – replicate **`products`** pattern, just replace **`products`** → **`orders`**.

8️⃣ App Router

`src/App.tsx`

```tsx
import React, { Suspense, lazy } from 'react'
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom'

const ProductsPage = lazy(() => import('@/features/products'))
const OrdersPage = lazy(() => import('@/features/orders'))

export default function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/products">Products</Link> |
        <Link to="/orders">Orders</Link>
      </nav>
      <Suspense fallback={<p>Loading module...</p>}>
        <Routes>
          <Route path="/" element={<h1>Home</h1>} />
          <Route path="/products" element={<ProductsPage />} />
          <Route path="/orders" element={<OrdersPage />} />
        </Routes>
      </Suspense>
    </Router>
  )
}
```

9️⃣ Entry Point

`src/main.tsx`

```tsx
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import { Provider } from 'react-redux'
import { PersistGate } from 'redux-persist/integration/react'
import { setupStore } from '@/app/store'

const { store, persistor } = setupStore()

ReactDOM.createRoot(document.getElementById('root')!).render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>
)
```

✅ Features of this Template

- Lazy-loaded **features**: Products & Orders.
- Dynamic **reducers + RTK Query endpoints** injection.
- Automatic **unloading** when leaving route.
- **`redux-persist`** for auth.
- Scalable for **hundreds of modules**.
- Fully **TypeScript-ready**, Vite-optimized.

If you want, I can also create a **ready-to-copy Vite project** with all these files preconfigured, including **mock API server** for testing products and orders.

Do you want me to do that?

instead of lazy load we can use simple load why it matter?

Ah, good question! Let's break it down carefully. Whether you use **lazy load** (React.lazy + Suspense) or just **regular imports** ("simple load") depends on your app size, architecture, and performance needs.

1️⃣ Simple Load (regular import)

```ts
import ProductsPage from '@/features/products'
```

Pros:

- Simple and straightforward.
- No Suspense or fallback UI needed.
- All code is loaded upfront → fewer moving parts.

Cons:

- **Initial bundle is bigger**: even modules the user might never visit are shipped.
- Slower **initial load** for large apps.
- Not memory-efficient if you have hundreds of features.

Use case: small apps, few routes, or admin dashboards with all modules always needed.

2️⃣ Lazy Load (React.lazy + Suspense)

```ts
const ProductsPage = React.lazy(() => import('@/features/products'))
```

Pros:

- **Code splitting**: only load the module when the user navigates to it.
- Smaller initial JS bundle → faster page load.
- Can dynamically inject reducers & endpoints **only when needed**.
- Reduces memory footprint for huge apps.
- Ideal for **enterprise apps or micro-frontend architectures**.

Cons:

- Slightly more complex setup (Suspense fallback, async reducer injection).
- Initial navigation may have a small delay while loading module.

Use case: large SaaS apps, dashboards with dozens of features, micro-frontends, or apps with heavy modules.

3️⃣ Why it matters in our setup

Our Redux template supports **dynamic reducer + RTK Query endpoint injection**.

- If you do **simple load**:
  - Every feature is imported upfront.
  - Reducers and endpoints exist immediately.
  - No memory saving.
- If you do **lazy load**:
  - Features self-register only when accessed.
  - Reducers and endpoints are injected dynamically.
  - They can also be **unloaded** when leaving the route.
  - Huge apps benefit because **startup bundle is smaller**.

✅ Summary

Approach	Bundle Size	Memory Usage	Complexity	Best For
Simple Load	Large	High	Low	Small apps
Lazy Load	Small	Low	Medium	Enterprise / Big dashboards

💡 In short:

- **Use lazy loading** if your app is large, has many features, or you want dynamic reducer/API injection.
- **Simple load** is fine for small apps where startup performance isn't critical.

If you want, I can show you a **hybrid approach**: load **critical features upfront**, and **lazy load secondary modules**, so you get the best of both worlds.

Do you want me to do that?

give full setup code with all typescript handling

Perfect ✅ I'll give you a **full TypeScript-ready Vite + React + Redux Toolkit + RTK Query setup** with:

- Dynamic reducer + RTK Query endpoint injection
- Feature modules: **`products`** + **`orders`**
- Full TypeScript types
- Lazy loading handled

- **`redux-persist`** for selected slices
- Proper typing for **`store`**, **`dispatch`**, **`selectors`**

1️⃣ Project Setup

```bash
npm create vite@latest my-enterprise-app -- --template react-ts
cd my-enterprise-app

npm install @reduxjs/toolkit react-redux redux-persist react-router-dom
```

2️⃣ Project Structure

```css
src/
 ├─ app/
 │   ├─ store.ts
 │   ├─ reducerManager.ts
 │   └─ rootReducer.ts
 ├─ services/
 │   └─ api.ts
 ├─ features/
 │   ├─ auth/authSlice.ts
 │   ├─ products/
 │   │   ├─ index.tsx
 │   │   ├─ productsSlice.ts
 │   │   └─ productsApi.ts
 │   └─ orders/
 │       ├─ index.tsx
 │       ├─ ordersSlice.ts
 │       └─ ordersApi.ts
 ├─ App.tsx
 └─ main.tsx
```

3️⃣ Base API

`src/services/api.ts`

```ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  tagTypes: ['Products', 'Orders'],
  endpoints: () => ({}),
})
```

4️⃣ Reducer Manager

`src/app/reducerManager.ts`

```ts
import { combineReducers, Reducer, AnyAction } from '@reduxjs/toolkit'

export interface ReducerManager {
  getReducerMap: () => Record<string, Reducer>
  reduce: Reducer
  add: (key: string, reducer: Reducer) => void
  remove: (key: string) => void
}

export function createReducerManager(initialReducers: Record<string, Reducer>): ReducerManager {
  let reducers = { ...initialReducers }
  let combinedReducer = combineReducers(reducers)
  const keysToRemove: string[] = []

  return {
    getReducerMap: () => reducers,
    reduce: (state, action: AnyAction) => {
      if (keysToRemove.length) {
        state = { ...state }
        keysToRemove.forEach((key) => delete state[key])
        keysToRemove.length = 0
      }
      return combinedReducer(state, action)
    },
    add: (key, reducer) => {
      if (!key || reducers[key]) return
      reducers[key] = reducer
      combinedReducer = combineReducers(reducers)
    },
    remove: (key) => {
      if (!key || !reducers[key]) return
      delete reducers[key]
      keysToRemove.push(key)
      combinedReducer = combineReducers(reducers)
    },
  }
}
```

5️⃣ Root Reducer

`src/app/rootReducer.ts`

```ts
import authSlice from '@/features/auth/authSlice'

export const staticReducers = {
  auth: authSlice,
}

export const createRootReducer = (asyncReducers: Record<string, any> = {}) => ({
  ...staticReducers,
  ...asyncReducers,
})
```

6️⃣ Store Setup

`src/app/store.ts`

ts

```ts
import { configureStore, Reducer } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import { staticReducers } from './rootReducer'
import { createReducerManager } from './reducerManager'
import { api } from '@/services/api'

const persistConfig = {
  key: 'root',
  storage,
  whitelist: ['auth'],
}

export function setupStore(preloadedState?: any) {
  const reducerManager = createReducerManager({
    ...staticReducers,
    [api.reducerPath]: api.reducer,
  })

  const persistedReducer: Reducer = persistReducer(persistConfig, reducerManager.reduce)

  const store = configureStore({
    reducer: persistedReducer,
    middleware: (getDefaultMiddleware) =>
      getDefaultMiddleware({
        serializableCheck: {
          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
        },
      }).concat(api.middleware),
    devTools: true,
    preloadedState,
  })

  ;(store as any).reducerManager = reducerManager
  const persistor = persistStore(store)

  return { store, persistor }
}

// Types
export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = ReturnType<typeof store.dispatch>

// Typed hooks
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux'
export const useAppDispatch = () => useDispatch<AppDispatch>()
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector

// Dynamic reducer injection
export function injectReducer(store: any, key: string, reducer: any) {
  const rm = (store as any).reducerManager
  if (!rm.getReducerMap()[key]) {
    rm.add(key, reducer)
    store.replaceReducer(rm.reduce)
  }
}
```

7️⃣ Features

Auth Slice

`src/features/auth/authSlice.ts`

```
ts

import { createSlice, PayloadAction } from '@reduxjs/toolkit'

interface AuthState {
  user: string | null
  token: string | null
}

const initialState: AuthState = { user: null, token: null }

const authSlice = createSlice({
  name: 'auth',
  initialState,
  reducers: {
    setUser: (state, action: PayloadAction<string>) => {
      state.user = action.payload
    },
    logout: (state) => {
      state.user = null
      state.token = null
    },
  },
})

export const { setUser, logout } = authSlice.actions
export default authSlice.reducer
```

Products Module (lazy + types)

- **`productsSlice.ts`**

```
ts

import { createSlice, PayloadAction } from '@reduxjs/toolkit'

interface Product { id: string; name: string }
interface ProductsState { items: Product[] }

const initialState: ProductsState = { items: [] }

const productsSlice = createSlice({
  name: 'products',
  initialState,
  reducers: { setProducts: (state, action: PayloadAction<Product[]>) => { state.items = action.payload } },
})

export default productsSlice.reducer
```

- **`productsApi.ts`**

```
ts

import { api } from '@/services/api'

let injected = false
let removeEndpoints: null | (() => void) = null

export interface Product { id: string; name: string }
export function useProductsApi() {
  if (!injected) {
    const injectedApi = api.injectEndpoints({
      endpoints: (builder) => ({
        getProducts: builder.query<Product[], void>({
          query: () => '/products',
          providesTags: ['Products'],
        }),
      }),
      overrideExisting: false,
    })
    injected = true
    removeEndpoints = injectedApi.util.removeEndpoints
  }

  return { useGetProductsQuery: api.endpoints.getProducts.useQuery }
}

export function unloadProductsApi() {
  if (removeEndpoints) {
    removeEndpoints(['getProducts'])
    injected = false
    removeEndpoints = null
  }
}
```

- **`index.tsx`** (lazy-loaded)

```
ts

import React, { useEffect } from 'react'
import { useStore } from 'react-redux'
import productsSlice from './productsSlice'
import { injectReducer } from '@/app/store'
import { useProductsApi, unloadProductsApi } from './productsApi'

export default function ProductsPage() {
  const store = useStore()
  useEffect(() => {
    injectReducer(store, 'products', productsSlice)
    return () => {
      store.reducerManager.remove('products')
      store.replaceReducer(store.reducerManager.reduce)
      unloadProductsApi()
    }
  }, [store])

  const { useGetProductsQuery } = useProductsApi()
  const { data, isLoading } = useGetProductsQuery()

  if (isLoading) return <p>Loading...</p>
  return <ul>{data?.map(p => <li key={p.id}>{p.name}</li>)}</ul>
}
```

Orders Module – same pattern

- **`ordersSlice.ts`**, **`ordersApi.ts`**, **`index.tsx`** → just replace **`products`** → **`orders`**.
- Define types: **`interface Order { id: string; items: any[] }`**.
- Lazy inject + unload on unmount.

8️⃣ App Router

`src/App.tsx`

```ts
import React, { Suspense, lazy } from 'react'
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom'

const ProductsPage = lazy(() => import('@/features/products'))
const OrdersPage = lazy(() => import('@/features/orders'))

export default function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |
        <Link to="/products">Products</Link> |
        <Link to="/orders">Orders</Link>
      </nav>
      <Suspense fallback={<p>Loading module...</p>}>
        <Routes>
          <Route path="/" element={<h1>Home</h1>} />
          <Route path="/products" element={<ProductsPage />} />
          <Route path="/orders" element={<OrdersPage />} />
        </Routes>
      </Suspense>
    </Router>
  )
}
```

9️⃣ Entry Point

`src/main.tsx`

```ts
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import { Provider } from 'react-redux'
import { PersistGate } from 'redux-persist/integration/react'
import { setupStore } from '@/app/store'

const { store, persistor } = setupStore()

ReactDOM.createRoot(document.getElementById('root')!).render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>
)
```

✅ **This is a fully TypeScript-safe, enterprise-ready Vite + React + Redux Toolkit template** with:

- Dynamic reducer + RTK Query endpoint injection
- Lazy-loaded feature modules
- Automatic unloading of reducers + endpoints
- Typed store, dispatch, and selectors
- **`redux-persist`** for selected slices

If you want, I can **add a mock API server** (`msw` or `json-server`) so you can **run this template immediately without a backend** for testing.

Do you want me to do that next?

what is static reducer means?

Ah! Let's clarify that carefully. In a Redux Toolkit + dynamic reducer setup, the term **"static reducer"** has a very specific meaning.

1️⃣ Static Reducers

Definition:
A **static reducer** is a reducer that is **always present** in your Redux store from the moment the app starts.

- It's not dynamically injected.
- It's required for core app functionality (e.g., authentication, UI state).
- You cannot remove it at runtime.

Example:

```ts
import authSlice from '@/features/auth/authSlice'

export const staticReducers = {
  auth: authSlice,  // always present
}
```

Here, `auth` is **static** because every page or feature in your app might rely on it.
Even if you later load "products" or "orders" dynamically, `auth` stays in the store.

2️⃣ Dynamic (Async) Reducers

Definition:
A **dynamic reducer** (sometimes called **async reducer**) is:

- Added to the store **at runtime**, usually when a feature module loads.
- Can also be **removed** when the module is no longer needed.
- Used for lazy-loaded features to save memory and reduce initial bundle size.

Example:

```ts
import productsSlice from '@/features/products/productsSlice'
import { injectReducer } from '@/app/store'

// inject products reducer only when the route is visited
injectReducer(store, 'products', productsSlice)
```

- `products` reducer is **dynamic**: it didn't exist when the app started.
- Can be unloaded with `store.reducerManager.remove('products')`.

3️⃣ Why we need static + dynamic reducers

- **Static reducers:** core features that the app always needs.
    - Example: `auth`, `ui`, `settings`, `api` slice.
- **Dynamic reducers:** feature-specific modules that may not be needed initially.
    - Example: `products`, `orders`, `reports`.

This approach is **industry standard** for large apps because it allows:

1. **Small initial bundle size** → faster app startup
2. **Memory efficiency** → unload unused modules
3. **Scalable architecture** → hundreds of features without bloating the store

---

In short:

Type	Load Time	Removable	Example
Static	App start	❌	`auth`, `ui`, `api`
Dynamic	Runtime	✅	`products`, `orders`

If you want, I can make a **diagram showing static vs dynamic reducers and lazy-loaded endpoints**, so the flow is visually clear for enterprise apps.

Do you want me to do that?

api what it is?

Ah, good! Let's break down **what the `api` is** in the context of **Redux Toolkit and RTK Query**.

1️⃣ RTK Query `api`

In Redux Toolkit, **RTK Query** provides a standard way to **define and manage API requests** in your store.

The `api` object is **not just a reducer**—it's a **centralized API service** that handles:

- Making HTTP requests
- Caching responses automatically
- Managing loading / error states
- Automatic invalidation of cache when mutations occur

Example: Base API

```ts
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export const api = createApi({
  reducerPath: 'api',                   // The key in Redux store
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }), // base URL for requests
  tagTypes: ['Products', 'Orders'],     // used for cache invalidation
  endpoints: () => ({}),                // initial endpoints (can be empty)
})
```

What each part means:

Field	Purpose
`reducerPath`	Name of this slice in Redux store. Example: **`store.api`**
`baseQuery`	Default fetch function for requests (**`fetchBaseQuery`** is built-in)
`tagTypes`	Categories for caching and invalidation (**`Products`**, **`Orders`**)
`endpoints`	Functions defining queries/mutations. Can be injected dynamically

---

2️⃣ API Endpoints

Endpoints are **functions you define on top of the base API**.

Example:

```ts
const injectedApi = api.injectEndpoints({
  endpoints: (builder) => ({
    getProducts: builder.query<Product[], void>({
      query: () => '/products',           // URL
      providesTags: ['Products'],         // cache tag
    }),
    addProduct: builder.mutation<Product, Product>({
      query: (body) => ({
        url: '/products',
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Products'],      // clears cache when mutation happens
    }),
  }),
  overrideExisting: false,
})
```

- **`query`** = GET request

- `mutation` = POST/PUT/DELETE request
- `providesTags` = defines which cache this query populates
- `invalidatesTags` = defines which cache should refresh after a mutation

3️⃣ Why we need a central `api` object

- **Single source of truth** for all backend calls.
- **Automatic caching + invalidation** → no need to manually update Redux state after a POST/PUT.
- **Dynamic injection** → we can add endpoints at runtime when features load.

For example:

```ts
// products module
const { useGetProductsQuery, useAddProductMutation } = api.endpoints.getProducts.useQuery()
```

- When you call `useGetProductsQuery`, RTK Query automatically fetches, caches, and subscribes to updates.
- If you call `useAddProductMutation`, the cache for `Products` is automatically invalidated, so `getProducts` refetches.

✅ TL;DR

- `api` = **RTK Query service object**.
- Handles **HTTP requests, caching, and Redux state** automatically.
- Can have **static endpoints** or **dynamic endpoints** injected later.
- Works perfectly with **dynamic reducers** in large apps.

If you want, I can make a **visual diagram showing how `api`, endpoints, cache, and dynamic reducers interact**, which makes it very easy to understand the flow.

Do you want me to do that?