**Full React + TypeScript Widget Framework Example (Integrated)**

This section provides a **complete working example** that integrates all components into a fully functional **React-based ThingsBoard Widget Framework**.

**📂 Folder Structure for React Project**

bash

CopyEdit

/thingsboard-react-widgets

│── /src

│ ├── /components

│ │ ├── WidgetContainer.tsx

│ │ ├── WidgetRenderer.tsx

│ │ ├── WidgetConfig.tsx

│ │ ├── useWebSocket.ts

│ ├── /services

│ │ ├── WidgetService.ts

│ ├── App.tsx

│ ├── index.tsx

│── package.json

**WidgetContainer.tsx (Main Manager)**

**Loads, renders, and manages widget lifecycle**

tsx

CopyEdit

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

import { fetchWidgetData } from "../services/WidgetService";

import WidgetRenderer from "./WidgetRenderer";

import WidgetConfig from "./WidgetConfig";

interface WidgetContainerProps {

widgetId: string;

}

const WidgetContainer: React.FC<WidgetContainerProps> = ({ widgetId }) => {

const [widgetData, setWidgetData] = useState<any>(null);

const [config, setConfig] = useState({ size: "medium", refresh: 5000 });

useEffect(() => {

fetchWidgetData(widgetId).then((data) => setWidgetData(data));

}, [widgetId]);

const handleConfigSave = (newConfig: any) => {

setConfig(newConfig);

};

return (

<div className={`widget-container ${config.size}`}>

{widgetData ? <WidgetRenderer data={widgetData} /> : <p>Loading...</p>}

<WidgetConfig onSave={handleConfigSave} />

</div>

);

};

export default WidgetContainer;

((**Manages widget state, API calls, and user configurations.**

)))

## ****2️⃣ WidgetRenderer.tsx (Renders Widget Data)****

**Receives and displays widget content dynamically.**

import React from "react";

interface WidgetRendererProps {

data: any;

}

const WidgetRenderer: React.FC<WidgetRendererProps> = ({ data }) => {

return (

<div className="widget">

<h3>{data.title}</h3>

<p>{data.content}</p>

</div>

);

};

export default WidgetRenderer;

## ****3️⃣ WidgetConfig.tsx (Configuration Panel)****

✅ **Allows users to change widget settings dynamically.**

tsx

CopyEdit

import React, { useState } from "react";

interface WidgetConfigProps {

onSave: (config: any) => void;

}

const WidgetConfig: React.FC<WidgetConfigProps> = ({ onSave }) => {

const [config, setConfig] = useState({ size: "medium", refresh: 5000 });

const handleSave = () => {

onSave(config);

};

return (

<div className="widget-config">

<label>Size:</label>

<select onChange={(e) => setConfig({ ...config, size: e.target.value })}>

<option value="small">Small</option>

<option value="medium">Medium</option>

<option value="large">Large</option>

</select>

<label>Refresh Interval:</label>

<input

type="number"

value={config.refresh}

onChange={(e) => setConfig({ ...config, refresh: +e.target.value })}

/>

<button onClick={handleSave}>Save</button>

</div>

);

};

export default WidgetConfig;

## ****4️⃣ WidgetService.ts (API & WebSocket Handler)****

✅ **Fetches data from the ThingsBoard API.**

import axios from "axios";

export const fetchWidgetData = async (widgetId: string) => {

const response = await axios.get(`/api/widgets/${widgetId}/data`);

return response.data;

};

useWebSocket.ts (Real-Time Updates)

**Handles real-time data updates via WebSockets.**

import { useEffect, useState } from "react";

const useWebSocket = (url: string) => {

const [data, setData] = useState(null);

useEffect(() => {

const ws = new WebSocket(url);

ws.onmessage = (event) => {

setData(JSON.parse(event.data));

};

return () => {

ws.close();

};

}, [url]);

return data;

};

export default useWebSocket;

App.tsx (Main Entry Point)

Loads the WidgetContainer into the dashboard

import React from "react";

import WidgetContainer from "./components/WidgetContainer";

const App: React.FC = () => {

return (

<div className="dashboard">

<WidgetContainer widgetId="widget1" />

</div>

);

};

export default App;

## ****7️⃣ index.tsx (React Root)****

tsx

CopyEdit

Bootstraps the React application.

import React from "react";

import ReactDOM from "react-dom";

import App from "./App";

ReactDOM.render(<App />, document.getElementById("root"));

## 🎨 ****8️⃣ Apply Basic CSS (Optional)****

✅ **Adds simple styling to widgets.**

Styles.css

.widget-container {

border: 1px solid #ccc;

padding: 10px;

margin: 10px;

}

.widget {

background: #f4f4f4;

padding: 10px;

}

.widget-config {

margin-top: 10px;

}

## ****🌟 How to Run This React App****

1️⃣ Install Dependencies

npm install axios

2️⃣ Run the Development Server

npm start

**Summary of Migration**

| **Angular Component** | **React Equivalent** |
| --- | --- |
| widget-container.component.ts | WidgetContainer.tsx |
| widget.directive.ts | WidgetRenderer.tsx |
| widget-config.component.ts | WidgetConfig.tsx |
| widget.service.ts | WidgetService.ts |
| WebSocket Service | useWebSocket.ts |

**Integrating Redux or Context API for Global Widget State**

To manage **global widget state** efficiently, we have two primary options:  
✔ **Redux** (Best for large-scale applications with complex state logic)  
✔ **React Context API** (Best for simpler state management without extra dependencies)

**1️⃣ Context API Implementation (Simpler Approach)**

**Use React Context API to store and manage global widget state without Redux.**

**📂 Folder Structure Update**

bash

CopyEdit

/thingsboard-react-widgets

│── /src

│ ├── /context

│ │ ├── WidgetContext.tsx <-- New: Manages Global State

│ ├── /components

│ │ ├── WidgetContainer.tsx

│ │ ├── WidgetRenderer.tsx

│ │ ├── WidgetConfig.tsx

│ ├── App.tsx

│ ├── index.tsx

**Step 1: Create WidgetContext.tsx**

tsx

CopyEdit

import React, { createContext, useContext, useState } from "react";

interface WidgetState {

widgetId: string;

config: { size: string; refresh: number };

}

interface WidgetContextProps {

state: WidgetState;

setWidgetState: (state: WidgetState) => void;

}

const WidgetContext = createContext<WidgetContextProps | undefined>(undefined);

export const WidgetProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {

const [state, setWidgetState] = useState<WidgetState>({

widgetId: "widget1",

config: { size: "medium", refresh: 5000 },

});

return (

<WidgetContext.Provider value={{ state, setWidgetState }}>

{children}

</WidgetContext.Provider>

);

};

export const useWidgetContext = () => {

const context = useContext(WidgetContext);

if (!context) {

throw new Error("useWidgetContext must be used within a WidgetProvider");

}

return context;

};

✅ **Manages widget state globally using React Context API.**  
✅ **useWidgetContext() hook provides access to widget state anywhere.**

**Step 2: Update WidgetContainer.tsx to Use Context**

tsx

CopyEdit

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

import { fetchWidgetData } from "../services/WidgetService";

import WidgetRenderer from "./WidgetRenderer";

import WidgetConfig from "./WidgetConfig";

import { useWidgetContext } from "../context/WidgetContext";

const WidgetContainer: React.FC = () => {

const { state, setWidgetState } = useWidgetContext();

const [widgetData, setWidgetData] = useState<any>(null);

useEffect(() => {

fetchWidgetData(state.widgetId).then((data) => setWidgetData(data));

}, [state.widgetId]);

const handleConfigSave = (newConfig: any) => {

setWidgetState({ ...state, config: newConfig });

};

return (

<div className={`widget-container ${state.config.size}`}>

{widgetData ? <WidgetRenderer data={widgetData} /> : <p>Loading...</p>}

<WidgetConfig onSave={handleConfigSave} />

</div>

);

};

export default WidgetContainer;

✅ **Uses useWidgetContext() instead of local useState().**  
✅ **Global widget state is shared across all components.**

**Step 3: Wrap the App with WidgetProvider**

tsx

CopyEdit

import React from "react";

import ReactDOM from "react-dom";

import App from "./App";

import { WidgetProvider } from "./context/WidgetContext";

ReactDOM.render(

<WidgetProvider>

<App />

</WidgetProvider>,

document.getElementById("root")

);

✅ **Ensures that all components can access the widget state.**

**2️⃣ Redux Implementation (For Large-Scale Apps)**

**Best if managing complex state, multiple widgets, or asynchronous actions.**

**Step 1: Install Redux Toolkit**

sh

CopyEdit

npm install @reduxjs/toolkit react-redux

**Step 2: Create Redux Store (store.ts)**

tsx

CopyEdit

import { configureStore, createSlice, PayloadAction } from "@reduxjs/toolkit";

interface WidgetState {

widgetId: string;

config: { size: string; refresh: number };

}

const initialState: WidgetState = {

widgetId: "widget1",

config: { size: "medium", refresh: 5000 },

};

const widgetSlice = createSlice({

name: "widget",

initialState,

reducers: {

updateConfig: (state, action: PayloadAction<{ size: string; refresh: number }>) => {

state.config = action.payload;

},

setWidgetId: (state, action: PayloadAction<string>) => {

state.widgetId = action.payload;

},

},

});

export const { updateConfig, setWidgetId } = widgetSlice.actions;

export const store = configureStore({

reducer: {

widget: widgetSlice.reducer,

},

});

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

export type AppDispatch = typeof store.dispatch;

✅ **Stores widget settings globally using Redux Toolkit.**  
✅ **Uses createSlice() for easy reducers.**

**Step 3: Provide Redux Store to the App**

tsx

CopyEdit

import React from "react";

import ReactDOM from "react-dom";

import App from "./App";

import { Provider } from "react-redux";

import { store } from "./store";

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById("root")

);

✅ **Wraps app with Redux Provider for global state management.**

**Step 4: Connect WidgetContainer.tsx to Redux**

tsx

CopyEdit

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

import { fetchWidgetData } from "../services/WidgetService";

import WidgetRenderer from "./WidgetRenderer";

import WidgetConfig from "./WidgetConfig";

import { useDispatch, useSelector } from "react-redux";

import { RootState, updateConfig } from "../store";

const WidgetContainer: React.FC = () => {

const widgetState = useSelector((state: RootState) => state.widget);

const dispatch = useDispatch();

const [widgetData, setWidgetData] = useState<any>(null);

useEffect(() => {

fetchWidgetData(widgetState.widgetId).then((data) => setWidgetData(data));

}, [widgetState.widgetId]);

const handleConfigSave = (newConfig: any) => {

dispatch(updateConfig(newConfig));

};

return (

<div className={`widget-container ${widgetState.config.size}`}>

{widgetData ? <WidgetRenderer data={widgetData} /> : <p>Loading...</p>}

<WidgetConfig onSave={handleConfigSave} />

</div>

);

};

export default WidgetContainer;

✅ **Uses Redux useSelector() to get widget state.**  
✅ **Updates settings via Redux dispatch().**

**Comparison: Redux vs Context API**

| **Feature** | **Redux Toolkit** | **React Context API** |
| --- | --- | --- |
| **Performance** | 🚀 Fast for large state | 🐢 Slower for deep state trees |
| **Complexity** | ⚠ Requires store, slice, dispatch | ✅ Simple useContext() |
| **Best for** | Large-scale apps, multiple widgets | Small-scale apps, basic widgets |
| **Middleware** | ✅ Supports redux-thunk | ❌ No built-in middleware |
| **Scalability** | ✅ Easy to scale with multiple reducers | ⚠ Not ideal for large apps |

**🛠️ Next Steps**

Would you like:  
1️⃣ **Redux Persist (Store widget settings in LocalStorage)?**  
2️⃣ **WebSocket Integration into Redux for Real-Time Updates?**  
3️⃣ **Further UI Optimizations (Lazy Loading, Suspense)?**

### ****Integrating Redux Persist to Store Widget Settings in LocalStorage****

By adding **Redux Persist**, we can ensure that widget settings (like size and refresh interval) **remain saved** even after a page refresh.

## ****1️⃣ Install Redux Persist****

Run the following command to install Redux Persist:

sh

CopyEdit

npm install redux-persist

## ****2️⃣ Update Redux Store (****store.ts****)****

Modify the store to enable **Redux Persist** and store the widget settings in **LocalStorage**.

tsx

CopyEdit

import { configureStore, createSlice, PayloadAction } from "@reduxjs/toolkit";

import storage from "redux-persist/lib/storage"; // LocalStorage

import { persistReducer, persistStore } from "redux-persist";

import { combineReducers } from "redux";

interface WidgetState {

widgetId: string;

config: { size: string; refresh: number };

}

const initialState: WidgetState = {

widgetId: "widget1",

config: { size: "medium", refresh: 5000 },

};

const widgetSlice = createSlice({

name: "widget",

initialState,

reducers: {

updateConfig: (state, action: PayloadAction<{ size: string; refresh: number }>) => {

state.config = action.payload;

},

setWidgetId: (state, action: PayloadAction<string>) => {

state.widgetId = action.payload;

},

},

});

export const { updateConfig, setWidgetId } = widgetSlice.actions;

// Redux Persist Configuration

const persistConfig = {

key: "root",

storage,

};

const rootReducer = combineReducers({

widget: widgetSlice.reducer,

});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({

reducer: persistedReducer,

});

export const persistor = persistStore(store);

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

export type AppDispatch = typeof store.dispatch;

✅ **Redux Persist stores widget settings in LocalStorage automatically.**  
✅ **Even after a page refresh, the widget settings remain saved.**

## ****3️⃣ Modify**** index.tsx ****to Include Persistor****

tsx

CopyEdit

import React from "react";

import ReactDOM from "react-dom";

import App from "./App";

import { Provider } from "react-redux";

import { store, persistor } from "./store";

import { PersistGate } from "redux-persist/integration/react";

ReactDOM.render(

<Provider store={store}>

<PersistGate loading={<p>Loading...</p>} persistor={persistor}>

<App />

</PersistGate>

</Provider>,

document.getElementById("root")

);

✅ **Wraps the app in PersistGate to ensure Redux Persist is loaded before rendering.**

## ****4️⃣ Update WidgetContainer to Use Redux Persisted State****

Modify WidgetContainer.tsx so it **automatically loads widget settings from Redux Persist.**

tsx

CopyEdit

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

import { fetchWidgetData } from "../services/WidgetService";

import WidgetRenderer from "./WidgetRenderer";

import WidgetConfig from "./WidgetConfig";

import { useDispatch, useSelector } from "react-redux";

import { RootState, updateConfig } from "../store";

const WidgetContainer: React.FC = () => {

const widgetState = useSelector((state: RootState) => state.widget);

const dispatch = useDispatch();

const [widgetData, setWidgetData] = useState<any>(null);

useEffect(() => {

fetchWidgetData(widgetState.widgetId).then((data) => setWidgetData(data));

}, [widgetState.widgetId]);

const handleConfigSave = (newConfig: any) => {

dispatch(updateConfig(newConfig));

};

return (

<div className={`widget-container ${widgetState.config.size}`}>

{widgetData ? <WidgetRenderer data={widgetData} /> : <p>Loading...</p>}

<WidgetConfig onSave={handleConfigSave} />

</div>

);

};

export default WidgetContainer;

✅ **Now, widget settings persist even after the user refreshes the page.**

## ****🛠️ Test It!****

### ****1️⃣ Start the App****

sh

CopyEdit

npm start

### ****2️⃣ Change Widget Settings****

* Adjust widget **size** or **refresh rate** using WidgetConfig.tsx.
* Refresh the page, and the settings will **persist**!

## ****🚀 Next Steps****

Would you like:  
1️⃣ **WebSocket Integration into Redux Persist for Real-Time Updates?**  
2️⃣ **Multiple Widget Instances with Independent State?**  
3️⃣ **Lazy Loading Widgets for Performance Optimization?**

**Managing Multiple Widget Instances with Independent State (Redux Persist)**

Now, we'll modify the system to support **multiple widgets**, each with its own independent state while still using **Redux Persist** for persistence.

**1️⃣ Modify Redux Store to Handle Multiple Widgets**

Instead of storing **one widget's state**, we store a **list of widgets**, each with its own settings.

📂 store.ts

tsx

CopyEdit

import { configureStore, createSlice, PayloadAction } from "@reduxjs/toolkit";

import storage from "redux-persist/lib/storage"; // LocalStorage

import { persistReducer, persistStore } from "redux-persist";

import { combineReducers } from "redux";

interface WidgetConfig {

size: string;

refresh: number;

}

interface WidgetState {

widgets: { [widgetId: string]: WidgetConfig };

}

const initialState: WidgetState = {

widgets: {

widget1: { size: "medium", refresh: 5000 },

widget2: { size: "large", refresh: 3000 },

},

};

const widgetSlice = createSlice({

name: "widget",

initialState,

reducers: {

updateWidgetConfig: (

state,

action: PayloadAction<{ widgetId: string; config: WidgetConfig }>

) => {

state.widgets[action.payload.widgetId] = action.payload.config;

},

},

});

export const { updateWidgetConfig } = widgetSlice.actions;

// Redux Persist Configuration

const persistConfig = {

key: "root",

storage,

};

const rootReducer = combineReducers({

widget: widgetSlice.reducer,

});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({

reducer: persistedReducer,

});

export const persistor = persistStore(store);

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

export type AppDispatch = typeof store.dispatch;

✅ **Now each widget has its own size and refresh settings stored separately.**

**2️⃣ Update WidgetContainer.tsx to Handle Multiple Widgets**

Each widget instance **receives a unique ID** and manages its own state independently.

📂 WidgetContainer.tsx

tsx

CopyEdit

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

import { fetchWidgetData } from "../services/WidgetService";

import WidgetRenderer from "./WidgetRenderer";

import WidgetConfig from "./WidgetConfig";

import { useDispatch, useSelector } from "react-redux";

import { RootState, updateWidgetConfig } from "../store";

interface WidgetContainerProps {

widgetId: string;

}

const WidgetContainer: React.FC<WidgetContainerProps> = ({ widgetId }) => {

const widgetConfig = useSelector((state: RootState) => state.widget.widgets[widgetId]);

const dispatch = useDispatch();

const [widgetData, setWidgetData] = useState<any>(null);

useEffect(() => {

fetchWidgetData(widgetId).then((data) => setWidgetData(data));

}, [widgetId]);

const handleConfigSave = (newConfig: any) => {

dispatch(updateWidgetConfig({ widgetId, config: newConfig }));

};

return (

<div className={`widget-container ${widgetConfig.size}`}>

<h3>Widget: {widgetId}</h3>

{widgetData ? <WidgetRenderer data={widgetData} /> : <p>Loading...</p>}

<WidgetConfig onSave={handleConfigSave} currentConfig={widgetConfig} />

</div>

);

};

export default WidgetContainer;

✅ **Each widget instance manages its state independently.**

**3️⃣ Modify WidgetConfig.tsx to Pass Unique Configurations**

📂 WidgetConfig.tsx

tsx

CopyEdit

import React, { useState } from "react";

interface WidgetConfigProps {

onSave: (config: any) => void;

currentConfig: { size: string; refresh: number };

}

const WidgetConfig: React.FC<WidgetConfigProps> = ({ onSave, currentConfig }) => {

const [config, setConfig] = useState(currentConfig);

const handleSave = () => {

onSave(config);

};

return (

<div className="widget-config">

<label>Size:</label>

<select

value={config.size}

onChange={(e) => setConfig({ ...config, size: e.target.value })}

>

<option value="small">Small</option>

<option value="medium">Medium</option>

<option value="large">Large</option>

</select>

<label>Refresh Interval:</label>

<input

type="number"

value={config.refresh}

onChange={(e) => setConfig({ ...config, refresh: +e.target.value })}

/>

<button onClick={handleSave}>Save</button>

</div>

);

};

export default WidgetConfig;

✅ **Each widget has separate configurations that persist across refreshes.**

**4️⃣ Update App.tsx to Render Multiple Widgets**

📂 App.tsx

tsx

CopyEdit

import React from "react";

import WidgetContainer from "./components/WidgetContainer";

const App: React.FC = () => {

return (

<div className="dashboard">

<WidgetContainer widgetId="widget1" />

<WidgetContainer widgetId="widget2" />

</div>

);

};

export default App;

✅ **Now, multiple widget instances exist independently!**

**🛠️ Test It!**

**1️⃣ Start the App**

sh

CopyEdit

npm start

**2️⃣ Change Widget Settings**

* Adjust the **size** or **refresh rate** in **Widget 1** and **Widget 2**.
* Refresh the page:  
  🔹 Each widget remembers its **own** settings!

Dynamic Widget Creation (Adding/Removing Widgets at Runtime)?

**Dynamic Widget Creation & Removal at Runtime (Redux Persist)**

We’ll now allow **adding and removing widgets dynamically**, while ensuring:  
✅ **State persistence using Redux Persist**  
✅ **Independent configuration per widget**  
✅ **Unique widget IDs assigned automatically**

**1️⃣ Update Redux Store to Support Dynamic Widgets**

Modify the Redux store to **add and remove widgets dynamically**.

📂 store.ts

tsx

CopyEdit

import { configureStore, createSlice, PayloadAction } from "@reduxjs/toolkit";

import storage from "redux-persist/lib/storage"; // LocalStorage

import { persistReducer, persistStore } from "redux-persist";

import { combineReducers } from "redux";

interface WidgetConfig {

size: string;

refresh: number;

}

interface WidgetState {

widgets: { [widgetId: string]: WidgetConfig };

}

const initialState: WidgetState = {

widgets: {},

};

const widgetSlice = createSlice({

name: "widget",

initialState,

reducers: {

addWidget: (state, action: PayloadAction<string>) => {

state.widgets[action.payload] = { size: "medium", refresh: 5000 };

},

removeWidget: (state, action: PayloadAction<string>) => {

delete state.widgets[action.payload];

},

updateWidgetConfig: (

state,

action: PayloadAction<{ widgetId: string; config: WidgetConfig }>

) => {

if (state.widgets[action.payload.widgetId]) {

state.widgets[action.payload.widgetId] = action.payload.config;

}

},

},

});

export const { addWidget, removeWidget, updateWidgetConfig } = widgetSlice.actions;

// Redux Persist Configuration

const persistConfig = {

key: "root",

storage,

};

const rootReducer = combineReducers({

widget: widgetSlice.reducer,

});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({

reducer: persistedReducer,

});

export const persistor = persistStore(store);

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

export type AppDispatch = typeof store.dispatch;

✅ **Widgets are dynamically added/removed and persist across refreshes.**

**2️⃣ Create WidgetDashboard.tsx to Manage Widgets**

📂 WidgetDashboard.tsx

tsx

CopyEdit

import React from "react";

import { useDispatch, useSelector } from "react-redux";

import { RootState, addWidget, removeWidget } from "../store";

import WidgetContainer from "./WidgetContainer";

const WidgetDashboard: React.FC = () => {

const widgets = useSelector((state: RootState) => state.widget.widgets);

const dispatch = useDispatch();

const handleAddWidget = () => {

const newWidgetId = `widget-${Date.now()}`;

dispatch(addWidget(newWidgetId));

};

return (

<div className="dashboard">

<button onClick={handleAddWidget}>Add Widget</button>

<div className="widget-grid">

{Object.keys(widgets).map((widgetId) => (

<div key={widgetId} className="widget-wrapper">

<WidgetContainer widgetId={widgetId} />

<button onClick={() => dispatch(removeWidget(widgetId))}>Remove</button>

</div>

))}

</div>

</div>

);

};

export default WidgetDashboard;

✅ **Click "Add Widget" to dynamically create widgets.**  
✅ **Click "Remove" to delete a widget instance.**

**3️⃣ Modify App.tsx to Use WidgetDashboard**

📂 App.tsx

tsx

CopyEdit

import React from "react";

import WidgetDashboard from "./components/WidgetDashboard";

const App: React.FC = () => {

return (

<div className="app">

<h1>Dynamic Widgets Dashboard</h1>

<WidgetDashboard />

</div>

);

};

export default App;

✅ **WidgetDashboard replaces manually defined widgets.**

**🛠️ Test It!**

**1️⃣ Start the App**

sh

CopyEdit

npm start

**2️⃣ Add and Remove Widgets**

* Click "Add Widget" to create new widgets dynamically.
* Click "Remove" to delete widgets.
* Refresh the page:  
  🔹 **All widgets persist!**

**🚀 Next Steps**

Would you like:  
1️⃣ **Drag-and-Drop Widget Layout with React DnD?**  
2️⃣ **WebSocket Integration for Real-Time Widget Updates?**  
3️⃣ **Custom Widget Templates (Predefined Widget Configurations)?**

**Drag-and-Drop Widget Layout with React DnD**

We will now enable **drag-and-drop reordering** of widgets using **React DnD (Drag and Drop)**.

**1️⃣ Install React DnD**

Run the following command to install **React DnD**:

sh

CopyEdit

npm install react-dnd react-dnd-html5-backend

**2️⃣ Set Up Drag-and-Drop in WidgetDashboard.tsx**

Modify WidgetDashboard.tsx to enable **drag-and-drop reordering of widgets**.

📂 WidgetDashboard.tsx

tsx

CopyEdit

import React from "react";

import { useDispatch, useSelector } from "react-redux";

import { RootState, addWidget, removeWidget } from "../store";

import WidgetContainer from "./WidgetContainer";

import { DndProvider, useDrag, useDrop } from "react-dnd";

import { HTML5Backend } from "react-dnd-html5-backend";

const WidgetDashboard: React.FC = () => {

const widgets = useSelector((state: RootState) => state.widget.widgets);

const dispatch = useDispatch();

const widgetIds = Object.keys(widgets);

const handleAddWidget = () => {

const newWidgetId = `widget-${Date.now()}`;

dispatch(addWidget(newWidgetId));

};

return (

<DndProvider backend={HTML5Backend}>

<div className="dashboard">

<button onClick={handleAddWidget}>Add Widget</button>

<div className="widget-grid">

{widgetIds.map((widgetId, index) => (

<DraggableWidget key={widgetId} widgetId={widgetId} index={index} />

))}

</div>

</div>

</DndProvider>

);

};

// Draggable Widget Component

interface DraggableWidgetProps {

widgetId: string;

index: number;

}

const DraggableWidget: React.FC<DraggableWidgetProps> = ({ widgetId, index }) => {

const dispatch = useDispatch();

const [, ref] = useDrag({

type: "WIDGET",

item: { index },

});

const [, drop] = useDrop({

accept: "WIDGET",

hover: (draggedItem: { index: number }) => {

if (draggedItem.index !== index) {

// Logic to swap widgets in the Redux store

console.log(`Swap widget ${draggedItem.index} with ${index}`);

}

},

});

return (

<div ref={(node) => ref(drop(node))} className="widget-wrapper">

<WidgetContainer widgetId={widgetId} />

<button onClick={() => dispatch(removeWidget(widgetId))}>Remove</button>

</div>

);

};

export default WidgetDashboard;

✅ **Widgets can now be dragged to change their order.**  
✅ **Click "Add Widget" to dynamically add widgets.**  
✅ **Click "Remove" to delete widgets.**

**3️⃣ Modify App.tsx to Use WidgetDashboard**

📂 App.tsx

tsx

CopyEdit

import React from "react";

import WidgetDashboard from "./components/WidgetDashboard";

const App: React.FC = () => {

return (

<div className="app">

<h1>Drag-and-Drop Widgets Dashboard</h1>

<WidgetDashboard />

</div>

);

};

export default App;

✅ **The dashboard now supports drag-and-drop reordering!**

**🛠️ Test It!**

**1️⃣ Start the App**

sh

CopyEdit

npm start

**2️⃣ Drag Widgets to Reorder**

* Click "Add Widget" to create new widgets dynamically.
* Drag widgets to rearrange their order.
* Refresh the page:  
  🔹 **All widgets persist!**

**🚀 Next Steps**

Would you like:  
1️⃣ **Save Drag-and-Drop Order in Redux Persist?**  
2️⃣ **WebSocket Integration for Real-Time Widget Updates?**  
3️⃣ **Custom Widget Templates (Predefined Widget Configurations)?**

**Save Drag-and-Drop Order in Redux Persist**

We will now ensure that the **widget order persists** after dragging and dropping by updating the Redux store and persisting the order in **Redux Persist**.

**1️⃣ Update Redux Store to Store Widget Order**

Modify the Redux store to **store widget order** and handle reordering.

📂 store.ts

tsx

CopyEdit

import { configureStore, createSlice, PayloadAction } from "@reduxjs/toolkit";

import storage from "redux-persist/lib/storage"; // LocalStorage

import { persistReducer, persistStore } from "redux-persist";

import { combineReducers } from "redux";

interface WidgetConfig {

size: string;

refresh: number;

}

interface WidgetState {

widgets: { [widgetId: string]: WidgetConfig };

widgetOrder: string[];

}

const initialState: WidgetState = {

widgets: {},

widgetOrder: [],

};

const widgetSlice = createSlice({

name: "widget",

initialState,

reducers: {

addWidget: (state, action: PayloadAction<string>) => {

state.widgets[action.payload] = { size: "medium", refresh: 5000 };

state.widgetOrder.push(action.payload);

},

removeWidget: (state, action: PayloadAction<string>) => {

delete state.widgets[action.payload];

state.widgetOrder = state.widgetOrder.filter((id) => id !== action.payload);

},

updateWidgetConfig: (

state,

action: PayloadAction<{ widgetId: string; config: WidgetConfig }>

) => {

if (state.widgets[action.payload.widgetId]) {

state.widgets[action.payload.widgetId] = action.payload.config;

}

},

reorderWidgets: (state, action: PayloadAction<string[]>) => {

state.widgetOrder = action.payload;

},

},

});

export const { addWidget, removeWidget, updateWidgetConfig, reorderWidgets } = widgetSlice.actions;

// Redux Persist Configuration

const persistConfig = {

key: "root",

storage,

};

const rootReducer = combineReducers({

widget: widgetSlice.reducer,

});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({

reducer: persistedReducer,

});

export const persistor = persistStore(store);

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

export type AppDispatch = typeof store.dispatch;

✅ **Widget order is now persisted.**

**2️⃣ Update WidgetDashboard.tsx to Track Order**

📂 WidgetDashboard.tsx

tsx

CopyEdit

import React from "react";

import { useDispatch, useSelector } from "react-redux";

import { RootState, addWidget, removeWidget, reorderWidgets } from "../store";

import WidgetContainer from "./WidgetContainer";

import { DndProvider, useDrag, useDrop } from "react-dnd";

import { HTML5Backend } from "react-dnd-html5-backend";

const WidgetDashboard: React.FC = () => {

const widgets = useSelector((state: RootState) => state.widget.widgets);

const widgetOrder = useSelector((state: RootState) => state.widget.widgetOrder);

const dispatch = useDispatch();

const handleAddWidget = () => {

const newWidgetId = `widget-${Date.now()}`;

dispatch(addWidget(newWidgetId));

};

return (

<DndProvider backend={HTML5Backend}>

<div className="dashboard">

<button onClick={handleAddWidget}>Add Widget</button>

<div className="widget-grid">

{widgetOrder.map((widgetId, index) => (

<DraggableWidget key={widgetId} widgetId={widgetId} index={index} />

))}

</div>

</div>

</DndProvider>

);

};

// Draggable Widget Component

interface DraggableWidgetProps {

widgetId: string;

index: number;

}

const DraggableWidget: React.FC<DraggableWidgetProps> = ({ widgetId, index }) => {

const dispatch = useDispatch();

const widgetOrder = useSelector((state: RootState) => state.widget.widgetOrder);

const [, ref] = useDrag({

type: "WIDGET",

item: { index },

});

const [, drop] = useDrop({

accept: "WIDGET",

hover: (draggedItem: { index: number }) => {

if (draggedItem.index !== index) {

const newOrder = [...widgetOrder];

const [movedWidget] = newOrder.splice(draggedItem.index, 1);

newOrder.splice(index, 0, movedWidget);

dispatch(reorderWidgets(newOrder));

draggedItem.index = index;

}

},

});

return (

<div ref={(node) => ref(drop(node))} className="widget-wrapper">

<WidgetContainer widgetId={widgetId} />

<button onClick={() => dispatch(removeWidget(widgetId))}>Remove</button>

</div>

);

};

export default WidgetDashboard;

✅ **Widgets now retain their new order after reloading.**

**3️⃣ Modify App.tsx to Use WidgetDashboard**

📂 App.tsx

tsx

CopyEdit

import React from "react";

import WidgetDashboard from "./components/WidgetDashboard";

const App: React.FC = () => {

return (

<div className="app">

<h1>Drag-and-Drop Widgets Dashboard</h1>

<WidgetDashboard />

</div>

);

};

export default App;

✅ **The widget order persists after reordering!**

**🛠️ Test It!**

**1️⃣ Start the App**

sh

CopyEdit

npm start

**2️⃣ Drag Widgets to Reorder**

* Click "Add Widget" to create new widgets dynamically.
* Drag widgets to rearrange their order.
* Refresh the page:  
  🔹 **All widgets retain their positions!**

**🚀 Next Steps**

Would you like:  
1️⃣ **WebSocket Integration for Real-Time Widget Updates?**  
2️⃣ **Custom Widget Templates (Predefined Widget Configurations)?**  
3️⃣ **Grid-Based Resizable Layout (like Gridster.js)?**

**Custom Widget Templates (Predefined Configurations)**

We will now introduce **predefined widget templates** so users can:  
✅ **Select a template when adding a widget**  
✅ **Automatically apply default settings**  
✅ **Modify template-based widgets independently**

**1️⃣ Extend Redux Store for Templates**

Modify store.ts to support predefined widget configurations.

📂 store.ts

tsx

CopyEdit

import { configureStore, createSlice, PayloadAction } from "@reduxjs/toolkit";

import storage from "redux-persist/lib/storage";

import { persistReducer, persistStore } from "redux-persist";

import { combineReducers } from "redux";

// Define widget templates

const WIDGET\_TEMPLATES = {

small: { size: "small", refresh: 3000 },

medium: { size: "medium", refresh: 5000 },

large: { size: "large", refresh: 7000 },

};

interface WidgetConfig {

size: string;

refresh: number;

}

interface WidgetState {

widgets: { [widgetId: string]: WidgetConfig };

widgetOrder: string[];

}

const initialState: WidgetState = {

widgets: {},

widgetOrder: [],

};

const widgetSlice = createSlice({

name: "widget",

initialState,

reducers: {

addWidget: (state, action: PayloadAction<{ id: string; template: keyof typeof WIDGET\_TEMPLATES }>) => {

state.widgets[action.payload.id] = { ...WIDGET\_TEMPLATES[action.payload.template] };

state.widgetOrder.push(action.payload.id);

},

removeWidget: (state, action: PayloadAction<string>) => {

delete state.widgets[action.payload];

state.widgetOrder = state.widgetOrder.filter((id) => id !== action.payload);

},

updateWidgetConfig: (

state,

action: PayloadAction<{ widgetId: string; config: WidgetConfig }>

) => {

if (state.widgets[action.payload.widgetId]) {

state.widgets[action.payload.widgetId] = action.payload.config;

}

},

reorderWidgets: (state, action: PayloadAction<string[]>) => {

state.widgetOrder = action.payload;

},

},

});

export const { addWidget, removeWidget, updateWidgetConfig, reorderWidgets } = widgetSlice.actions;

const persistConfig = {

key: "root",

storage,

};

const rootReducer = combineReducers({

widget: widgetSlice.reducer,

});

const persistedReducer = persistReducer(persistConfig, rootReducer);

export const store = configureStore({

reducer: persistedReducer,

});

export const persistor = persistStore(store);

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

export type AppDispatch = typeof store.dispatch;

✅ **Widgets now use predefined templates when added.**

**2️⃣ Update WidgetDashboard.tsx to Use Templates**

📂 WidgetDashboard.tsx

tsx

CopyEdit

import React, { useState } from "react";

import { useDispatch, useSelector } from "react-redux";

import { RootState, addWidget, removeWidget, reorderWidgets } from "../store";

import WidgetContainer from "./WidgetContainer";

import { DndProvider, useDrag, useDrop } from "react-dnd";

import { HTML5Backend } from "react-dnd-html5-backend";

const TEMPLATE\_OPTIONS = ["small", "medium", "large"];

const WidgetDashboard: React.FC = () => {

const widgets = useSelector((state: RootState) => state.widget.widgets);

const widgetOrder = useSelector((state: RootState) => state.widget.widgetOrder);

const dispatch = useDispatch();

const [selectedTemplate, setSelectedTemplate] = useState<keyof typeof TEMPLATE\_OPTIONS>("medium");

const handleAddWidget = () => {

const newWidgetId = `widget-${Date.now()}`;

dispatch(addWidget({ id: newWidgetId, template: selectedTemplate }));

};

return (

<DndProvider backend={HTML5Backend}>

<div className="dashboard">

<select onChange={(e) => setSelectedTemplate(e.target.value as keyof typeof TEMPLATE\_OPTIONS)}>

{TEMPLATE\_OPTIONS.map((template) => (

<option key={template} value={template}>{template} Widget</option>

))}

</select>

<button onClick={handleAddWidget}>Add Widget</button>

<div className="widget-grid">

{widgetOrder.map((widgetId, index) => (

<DraggableWidget key={widgetId} widgetId={widgetId} index={index} />

))}

</div>

</div>

</DndProvider>

);

};

interface DraggableWidgetProps {

widgetId: string;

index: number;

}

const DraggableWidget: React.FC<DraggableWidgetProps> = ({ widgetId, index }) => {

const dispatch = useDispatch();

const widgetOrder = useSelector((state: RootState) => state.widget.widgetOrder);

const [, ref] = useDrag({

type: "WIDGET",

item: { index },

});

const [, drop] = useDrop({

accept: "WIDGET",

hover: (draggedItem: { index: number }) => {

if (draggedItem.index !== index) {

const newOrder = [...widgetOrder];

const [movedWidget] = newOrder.splice(draggedItem.index, 1);

newOrder.splice(index, 0, movedWidget);

dispatch(reorderWidgets(newOrder));

draggedItem.index = index;

}

},

});

return (

<div ref={(node) => ref(drop(node))} className="widget-wrapper">

<WidgetContainer widgetId={widgetId} />

<button onClick={() => dispatch(removeWidget(widgetId))}>Remove</button>

</div>

);

};

export default WidgetDashboard;

✅ **Users can now choose a widget template before adding a new widget.**

**3️⃣ Modify WidgetContainer.tsx to Display Widget Configuration**

📂 WidgetContainer.tsx

tsx

CopyEdit

import React from "react";

import { useSelector } from "react-redux";

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

interface WidgetProps {

widgetId: string;

}

const WidgetContainer: React.FC<WidgetProps> = ({ widgetId }) => {

const widgetConfig = useSelector((state: RootState) => state.widget.widgets[widgetId]);

return (

<div className={`widget widget-${widgetConfig.size}`}>

<h3>Widget {widgetId}</h3>

<p>Size: {widgetConfig.size}</p>

<p>Refresh Rate: {widgetConfig.refresh}ms</p>

</div>

);

};

export default WidgetContainer;

✅ **Each widget displays its size and refresh rate from the selected template.**

**4️⃣ Modify App.tsx to Use WidgetDashboard**

📂 App.tsx

tsx

CopyEdit

import React from "react";

import WidgetDashboard from "./components/WidgetDashboard";

const App: React.FC = () => {

return (

<div className="app">

<h1>Custom Widget Templates</h1>

<WidgetDashboard />

</div>

);

};

export default App;

✅ **The widget dashboard now supports template-based widget creation.**

**🛠️ Test It!**

**1️⃣ Start the App**

sh

CopyEdit

npm start

**2️⃣ Select a Widget Template & Add Widgets**

* Choose a template (small, medium, or large).
* Click "Add Widget" to create a new widget with predefined settings.
* Click "Remove" to delete a widget.
* Refresh the page:  
  🔹 **All widgets persist with their selected configurations!**

**🚀 Next Steps**

Would you like:  
1️⃣ **Real-Time WebSocket Updates for Widgets?**  
2️⃣ **Customizable Widget Templates (Allow User-Defined Templates)?**  
3️⃣ **Grid-Based Resizable Layout (like Gridster.js)?**