

actually what the thing is I will tell you. I'm facing one issue of handeling UI by manually each time so what I want to do is I will create the dinamic library and will use in my layouts. as it will have custom buttons with some custom fields. inbuild permission management. inbuild theme management. extensible flexible theme architecture as strech sidebar vagera and all dinamic layouts so that the developoment will get more easier. and for applying handler to that components we will just create the flexible parameters to there so that events will also be handeled.

aditionally a seperate library for integrating and using redux and dispatchers combinely using the library. so the purpose of library is very clear to implement redux in auto mode instead of manual things and to integrate API's too. I do have clear Idea regarding both just want to know the optimized process to go with there.

tell me how I can create library there. also it should be flexible for jsx and tsx so that it will be used for both the technologies.

FlxStruct: for using flexible structures

Integrest: for integrating api's and redux dispatchers

both the libraries are different-different by the use and implementation as well. it should be like crete library and use npm install to install it as dependency and directly import it and use it. this seems intresting. let's move:

Cool — you're basically describing **two reusable npm packages**:

1. **FlxStruct** (UI framework layer): layout system + theme engine + permissions + shared components + event handling via config/props
2. **Integrest** (data layer): "auto" Redux setup + dispatchers + API integration (and ideally RTK Query) so devs stop wiring things manually

Below is an optimized, real-world process to build them so they're **flexible for JSX + TSX**, installable via **npm**, and easy to maintain.

1) Choose the architecture: Monorepo (recommended)

Use a monorepo so both libraries share tooling, lint rules, build pipeline, release process, and types.

Recommended stack

- **pnpm workspaces** (fast + good monorepo support)
- **Turborepo** (optional, speeds builds)
- **tsup** for packaging (simple + generates types)
- **changesets** for versioning + publishing
- React 18+ as peerDependency

Folder idea:

```
go  
  
adl-libs/  
  packages/  
    flxstruct/  
    integrest/  
  configs/  
  package.json  
  pnpm-workspace.yaml
```

 Copy code

Why monorepo helps you:

- Same “design tokens” / permission rules can be shared later
- Easier cross-testing between packages
- Publishing both feels consistent (same commands)

2) FlxStruct design: “Config-driven UI kit”

What FlxStruct should export

Think in **three layers**:

A) Core engines

- `ThemeProvider`, `createTheme()`, `extendTheme()`
- `PermissionProvider`, `usePermission()`, `Can` component
- `LayoutProvider`, `createLayout()`

B) Layout primitives (the flexible structure)

- <Shell layout="stretchSidebar" />
- <Sidebar /> <Header /> <Content />
- Config allows:
 - collapsible sidebar
 - stretch behavior
 - dynamic menu based on permissions
 - theme switching

C) Component kit

- Buttons, inputs, cards, dialogs, etc.
- All components accept:
 - variant, size, intent, etc.
 - permission?: string | string[]
 - onAction?: (actionId, payload) => void (central event bridge)

Event handling (your “handlers via flexible parameters” idea)

Instead of wiring `onClick` everywhere, support:

- actionId: "user:create" | "sidebar:toggle" etc.
- payload?: any
- onAction(actionId, payload) at app level

Example usage:

```
tsx

<Shell
  theme={myTheme}
  permissions={userPerms}
  onAction={(id, payload) => {
    if (id === "sidebar:toggle") toggleSidebar()
    if (id === "user:create") navigate("/users/new")
  }}
/>

<FlxButton actionId="user:create">Create User</FlxButton>
```

 Copy code

This makes the UI “dynamic + declarative”.

Permissions (inbuild)

Provide 2 patterns:

- **Component gating**
 - `<Can do="user.edit">...</Can>`
- **Prop gating**
 - `<FlexButton permission="user.create" />`

Rules:

- If not allowed: hide OR disable (configurable)
-

3) Integrest design: “Auto Redux + API layer”

Strong recommendation: build on Redux Toolkit

Don't reinvent Redux boilerplate. Use:

- `@reduxjs/toolkit`
- optionally `RTK Query` for API calls

Integrest should provide:

- `createIntegrestStore(config)`
 - auto combines slices
 - auto adds middleware
 - auto sets up devtools
- `createModule({ name, slice, api })`
 - modules can plug in
- `createApiClient()` wrapper (fetch/axios) with interceptors
- Optional:
 - `createDispatcher()` to auto-generate typed action dispatchers

Key idea: app dev should do minimal code:

ts

```
const { store, hooks } = createIntegrestStore({
  modules: [userModule, productModule],
  apiBaseUrl: "/api",
})
```

 Copy code

And then:

- `hooks.useUser()` or `hooks.useAppDispatch()`
- prebuilt API hooks if using RTK Query

"Auto mode" Redux

You can auto-register:

- reducers/slices
- thunks/async actions
- API endpoints
- selectors

So dev only "adds module" rather than wiring store manually.

4) JSX + TSX compatibility

To support **JSX users** and **TSX users**, do this:

- Write library source in **TypeScript** always (best for long-term).
- Publish:
 - compiled JS (for JS users)
 - `.d.ts` types (for TS users)

So JS users can just install + import normally.

TS users get intellisense and type safety.

5) Packaging rules (important for npm libraries)

Use `peerDependencies` correctly

In both packages:

- `react`, `react-dom` should be **peerDependencies**
- so you don't bundle React into your library

Example:

json

```
"peerDependencies": {  
  "react": ">=18",  
  "react-dom": ">=18"  
}
```

 Copy code

For Integrst:

json

```
"peerDependencies": {  
  "@reduxjs/toolkit": ">=2",  
  "react-redux": ">=9"  
}
```

 Copy code

Build output: ESM + CJS + Types

Your `package.json` should expose:

- `"main"` (CJS)
 - `"module"` (ESM)
 - `"types"` (d.ts)
 - `"exports"` map (modern bundlers love this)
-

6) Step-by-step process (optimized)

Step A: Bootstrap monorepo

- `pnpm init`
- `setup pnpm-workspace.yaml`
- `create packages/flxstruct , packages/integrest`

Step B: Build system

Use `tsup` in each package:

- `outputs dist/index.js` (esm)
- `outputs dist/index.cjs` (cjs)
- `outputs dist/index.d.ts`

Step C: Local testing before publish

Use `pnpm -r build`

Then in a separate test app:

- `pnpm add ../packages/flxstruct`
or use `pnpm link / workspace:*`

Step D: Versioning + Publishing

Use `changesets`:

- `pnpm changeset` to generate version note
- `pnpm changeset version`

- `pnpm -r publish`

Publish order usually doesn't matter unless one depends on the other.

7) What to build first (so you don't overbuild)

FlxStruct MVP

1. ThemeProvider + tokens
2. Shell layout (stretch sidebar)
3. PermissionProvider + Can
4. Button/Input with permission + actionId support
5. One sample "dynamic layout config" JSON

Integrest MVP

1. createStore wrapper (RTK)
2. module registration (slice + optional thunks)
3. api client wrapper
4. example module with auto exports

Once MVP works, then extend:

- more layout types
 - plugin system for new layouts
 - codegen for API endpoints (optional)
-

— ChatGPT 5.2 ▾



FlxStruct plugins

- layout plugins: `registerLayout("stretchSidebar", Component)`
- theme plugins: `registerTheme("darkPro", themeObj)`
- component overrides: `components.Button = CustomButton`

Integrest plugins

- auth plugin adds interceptors + auth slice
- logging plugin adds middleware
- error handling plugin adds standardized toasts/events

9) Concrete naming & exports

FlxStruct exports

- `ThemeProvider`, `createTheme`, `extendTheme`, `useTheme`
- `PermissionProvider`, `Can`, `usePermission`
- `Shell`, `createLayout`, `registerLayout`
- `FlxButton`, `FlxInput`, etc.

Integrest exports

- `createIntegrestStore`
 - `createModule`
 - `createApiClient`
 - `createDispatcher`
-

If you want, I can also give you:

- a clean **starter folder structure** for both packages,
- the exact `package.json` + `tsup.config.ts` template for dual ESM/CJS/types,
- and a minimal example showing FlxStruct + Integrest working together in a demo app.

But even without that, if you follow the plan above you'll end up with proper npm-installable libraries that work in **JSX + TSX** and remove most manual UI/Redux wiring.

