

Component Package Extraction Guide

This document describes the current state of [SyncfusionThemeStudio/src/components](#) and what is required to extract them into a standalone, shareable npm package for use across UI projects.

Two extraction paths are covered:

1. **Native-only package** - Zero third-party UI dependencies, CSS-variable-driven
 2. **Syncfusion-only package** - Themed wrappers around Syncfusion EJ2 React components
-

Table of Contents

- [Current Architecture Overview](#)
 - [Path A: Native Components Package](#)
 - [Path B: Syncfusion Components Package](#)
 - [Shared Infrastructure \(Both Paths\)](#)
 - [Recommended Package Structure](#)
 - [Consumer Setup](#)
 - [Migration Checklist](#)
 - [Why Syncfusion Extraction is Fundamentally Harder Than Native](#)
 - [What Must Exist at the Project Root \(and Why\)](#)
 - [How the Project Was Built and Why It Needs Rework](#)
 - [Critical: CSS Layer Ordering and Style Loading](#)
-

Current Architecture Overview

Component Inventory

Category	Location	Files	Description
Native UI	src/components/ui/native/	126	Zero-dependency components using CSS variables + Tailwind
Syncfusion wrappers	src/components/ui/syncfusion/	39	Theme-aware wrappers around Syncfusion EJ2 React
Shared types	src/components/ui/shared/	22	Const enums, interfaces, and type definitions
Form field adapters	src/components/ui/form-fields/	9	React Hook Form integrations for both native and Syncfusion
Icons	src/components/icons/	30+	Feather icon wrappers and custom icon sets

Style System (Critical Dependency)

All components are styled through a **CSS custom property system**. Styles live in [src/styles/layers/](#):

File	Lines	Used By	Purpose
base.css	395	Both	CSS variables: colors, typography, radius, transitions, component defaults, dark mode
components.css	1,395	Native	Component classes: <code>.native-btn</code> , <code>.native-select</code> , <code>.native-input</code> , etc.
native-overrides.css	148	Native	Native-specific style corrections
syncfusion-overrides.css	1,946	Syncfusion	Overrides for Syncfusion default styling to match theme

File	Lines	Used By	Purpose
syncfusion-themed.css	281	Syncfusion	Theme-aware Syncfusion class definitions (<code>.sf-themed</code> , <code>.sf-light</code> , <code>.sf-dark</code>)

Layer order (increasing specificity):

```
base -> syncfusion-base -> components -> syncfusion-overrides -> utilities
```

CSS Variable Categories

The `base.css` file defines the following variable groups that components consume:

- **Primary colors:** `--color-primary-50` through `--color-primary-900` (RGB triplets for Tailwind alpha support)
- **Status colors:** `--color-success-*`, `--color-warning-*`, `--color-error-*`, `--color-info-*`
- **Semantic surfaces:** `--color-background`, `--color-surface`, `--color-surface-elevated`, `--color-border`
- **Text colors:** `--color-text-primary`, `--color-text-secondary`, `--color-text-muted`
- **Typography:** `--font-sans`, `--font-mono`, `--font-size-*`, `--font-weight-*`, `--line-height-*`
- **Border radius:** `--radius-none` through `--radius-full`
- **Transitions:** `--transition-fast`, `--transition-normal`, `--transition-slow`
- **Animations:** `--animation-enabled`, `--animation-default-*`
- **Per-component defaults:** `--component-button-*`, `--component-input-*`, `--component-datagrid-*`, `--component-card-*`, `--component-modal-*`, `--component-badge-*`, `--component-select-*`, etc.
- **Dark mode overrides:** All of the above are re-declared under `.dark {}` selector

Utility Dependencies

Every component imports these two tiny utility files:

`src/utils/cn.ts` (7 lines) - className joiner:

```
export function cn(...classes: Array<string | boolean | undefined | null>): string {
  return classes.filter((c): c is string => typeof c === 'string' && c.length > 0).join(' ');
}
```

`src/utils/is.ts` (37 lines) - Type guards:

```
export function isValueDefined<T>(value: T | null | undefined): value is T
export function isNullOrUndefined(value: unknown): value is null | undefined
export function isNotEmptyString(value: string | null | undefined): value is string
export function isNotEmptyArray<T>(value: T[] | null | undefined): value is T[]
```

Localization Dependency

16 components (mostly native) use `FM()` from `src/localization/helpers.ts` for translated strings (e.g., table pagination labels, dialog button text). This function wraps `i18next.t()`.

Path Aliases

All imports use `@/` aliases resolved in `tsconfig.json`:

```
{
  "@/*": ["./src/*"],
  "@components/*": ["./src/components/*"],
```

```

"@stores/*": ["./src/stores/*"],
"@utils/*": ["./src/utils/*"],
"@styles/*": ["./src/styles/*"]
}

```

Path A: Native Components Package

What You Get

32 components with zero third-party UI dependencies:

Component	Category
ButtonNative, IconButtonNative, FabNative, SplitButtonNative	Actions
InputNative, SelectNative, CheckboxNative, DatePickerNative, RadioNative, ToggleNative	Forms
TableNative (with pagination, filtering, sorting, editing, grouping)	Data display
TabsNative, TimelineNative, BadgeNative, TagNative, ChipNative	Data display
AvatarNative, CardNative, ProgressBarNative	Data display
AccordionNative, MenuNative, BreadcrumbNative, ToolbarNative	Navigation
AlertNative, DialogNative, ToastNative (with context provider), TooltipNative	Feedback
HeadingNative, TextNative, DescriptionNative	Typography
ThemeToggleNative	Theme

What Must Ship With It

Artifact	Source	Why
Component source	src/components/ui/native/**	The components themselves
Shared types	src/components/ui/shared/**	Enums & interfaces all components import
cn() utility	src/utils/cn.ts	Every component uses this
is.*() guards	src/utils/is.ts	Every component uses this
Grid library	src/lib/grid/ (16 files)	Required by TableNative
Base CSS variables	src/styles/layers/base.css	All components consume CSS variables
Component CSS classes	src/styles/layers/components.css	Native component class definitions
Native overrides	src/styles/layers/native-overrides.css	Native-specific corrections
Tailwind preset	tailwind.config.ts (theme.extend portion)	Maps CSS variables to Tailwind classes

Current State - What Works Today

- All components are already self-contained in `src/components/ui/native/`
- Clean barrel export via `src/components/ui/native/index.ts` (104 lines)
- No dependency on Zustand, Syncfusion, or any state management
- Styling is 100% CSS-variable-driven with sensible defaults in `base.css`
- Dark mode works via `.dark` class on a parent element

Current State - What Blocks Extraction

Issue	Severity	Details

Issue	Severity	Details
Path aliases	High	All imports use <code>@/</code> which won't resolve outside the monorepo. Must be resolved at build time.
i18n coupling	Medium	16 components import <code>FM()</code> which requires a configured i18next instance. Cannot work without it.
No build pipeline	High	No library build config exists. The project builds as an app (Vite), not a library.
Tailwind requirement	Medium	Components use Tailwind utility classes alongside CSS classes. Consumers must have Tailwind.
CSS not independently importable	Medium	Styles are bundled as a monolithic <code>index.css</code> with Syncfusion imports. Native styles need isolation.
SearchInput in shared	Low	<code>src/components/ui/shared/SearchInput.tsx</code> is a React component in the types barrel. Should move.

Changes Required for Native Package

1. Create library build config (`tsup.config.ts` or `vite.config.lib.ts`)

- Externalize `react`, `react-dom`
- Resolve `@/` path aliases at build time
- Output ESM + CJS + type declarations
- Tree-shake unused components

2. Decouple `FM()` from `i18next` - Two options:

- **Option A** (recommended): Accept an optional `t` function prop, fall back to English defaults
- **Option B**: Export a `configureLocalization()` function consumers call once

3. Create isolated CSS entry point

- New file: `styles/native.css` that imports only `base.css` + `components.css` + `native-overrides.css`
- Exclude all Syncfusion CSS imports

4. Extract Tailwind preset

- New file: `tailwind-preset.js` containing only the `theme.extend` values from `tailwind.config.ts`
- Remove app-specific values (`sidebar`, `header` spacing)

5. `Package.json` with subpath exports

```
{
  "exports": {
    ".": "./dist/index.js",
    "./styles": "./dist/styles/native.css",
    "./tailwind-preset": "./dist/tailwind-preset.js"
  }
}
```

Peer Dependencies (Native)

```
{
  "peerDependencies": {
    "react": "^18.0.0 || ^19.0.0",
    "react-dom": "^18.0.0 || ^19.0.0"
  },
  "peerDependenciesMeta": {
```

```

        "tailwindcss": { "optional": true }
    }
}

```

Tailwind is optional because consumers can use the shipped CSS classes directly without Tailwind, but they lose the Tailwind utility classes used in some component styles.

Path B: Syncfusion Components Package

What You Get

17 themed wrapper components:

Component	Wraps
Button	@syncfusion/ej2-react-buttons
Input	@syncfusion/ej2-react-inputs
Select	@syncfusion/ej2-react-dropdowns
DataGrid	@syncfusion/ej2-react-grids
DatePicker	@syncfusion/ej2-react-calendars
Dialog	@syncfusion/ej2-react-popups
Alert	Custom (uses Syncfusion styling)
Tabs	@syncfusion/ej2-react-navigations
Timeline	@syncfusion/ej2-react-navigations
Tag, Badge, Avatar, Card, ProgressBar, Tooltip, Description	Various

Plus hooks: `useSyncfusionTheme`, `getButtonClasses`

What Must Ship With It

Everything from the Native path, **plus**:

Artifact	Source	Why
Syncfusion wrappers	<code>src/components/ui/syncfusion/**</code>	The components themselves
Syncfusion types	<code>src/components/ui/syncfusion/types.ts</code>	<code>SF_THEME_PREFIX</code> , <code>SF_LIGHT_CLASS</code> , theme interfaces
Theme hook	<code>src/components/ui/syncfusion/hooks/</code>	<code>useSyncfusionTheme</code> - reads mode from zustand store
Mode enum	<code>src/stores/mode.ts</code>	<code>Mode.Light</code> / <code>Mode.Dark</code>
Theme store	<code>src/stores/useThemeStore.ts</code>	Zustand store that holds current mode
Theme system	<code>src/stores/theme/</code> (120 files)	Default theme, injectors, presets, actions, types
Syncfusion override CSS	<code>src/styles/layers/syncfusion-overrides.css</code>	1,946 lines of Syncfusion styling overrides
Syncfusion themed CSS	<code>src/styles/layers/syncfusion-themed.css</code>	281 lines of theme-aware class definitions
Syncfusion base CSS	<code>@syncfusion/ej2-*/styles/tailwind.css</code>	Syncfusion's own base stylesheets

Current State - What Works Today

- All wrappers are self-contained in `src/components/ui/syncfusion/`
- Clean barrel export via `src/components/ui/syncfusion/index.ts`
- Theme-aware: automatically switches CSS classes based on light/dark mode
- `useSyncfusionTheme()` hook provides all theme classes to consumers

Current State - What Blocks Extraction

Issue	Severity	Details
Zustand store coupling	Critical	Every Syncfusion component calls <code>useSyncfusionTheme()</code> which reads from a zustand store (<code>useThemeStore</code>). This store is deeply integrated into the app.
Theme system size	High	The theme store spans 120 files. It manages colors, typography, layout, animations, and per-component configs. Extracting it means shipping a massive dependency.
Runtime CSS injection	High	<code>themeInjector.ts</code> writes CSS variables to <code>:root</code> at runtime based on store state. Consumers must either use this system or provide their own.
Syncfusion license	Medium	Syncfusion EJ2 requires a commercial license. Consumers must have their own.
Path aliases	High	Same as native - all <code>@/</code> imports must be resolved.
i18n coupling	Medium	Some wrappers use <code>FM()</code> .
No build pipeline	High	Same as native.
CSS not independently importable	Medium	Syncfusion styles are mixed with native styles in <code>index.css</code> .

Changes Required for Syncfusion Package

Everything from the native path, **plus**:

1. Refactor theme consumption to React Context (the biggest change)

Currently: Components call `useThemeStore()` directly (zustand global store).

Target: Components consume a `<ThemeProvider>` context.

```
// Package exports a provider + hook
export function ThemeProvider({ mode, children }: { mode: 'light' | 'dark'; children: ReactNode }) {
    // inject CSS variables, provide context
}

// Components consume via hook
export function useSyncfusionTheme(): SyncfusionTheme {
    return useContext(ThemeContext); // instead of useThemeStore()
}
```

This decouples components from the zustand store and lets consumers control mode however they want.

2. Extract a minimal theme injector

- Current `themeInjector.ts` injects 100+ CSS variables based on full `ThemeConfig`
- Package only needs: mode (light/dark) and optional color overrides
- Ship a slim injector that sets CSS variables based on a simple config object

3. Isolate Syncfusion CSS entry point

```
/* styles/syncfusion.css */
@import '@syncfusion/ej2-base/styles/tailwind.css';
@import '@syncfusion/ej2-react-inputs/styles/tailwind.css';
@import '@syncfusion/ej2-react-buttons/styles/tailwind.css';
@import './layers/base.css';
@import './layers/syncfusion-overrides.css';
@import './layers/syncfusion-themed.css';
```

4. Externalize all Syncfusion packages as peer dependencies

5. Decide on theme store shipping strategy:

- **Option A** (recommended): Ship components + `ThemeProvider` + slim injector. No `zustand` dependency. Consumers provide `mode` prop.
- **Option B**: Ship the full theme store (120 files + `zustand`). Consumers get the full theme editor capability. Much larger bundle.

Peer Dependencies (Syncfusion)

```
{
  "peerDependencies": {
    "react": "^18.0.0 || ^19.0.0",
    "react-dom": "^18.0.0 || ^19.0.0",
    "@syncfusion/ej2-react-buttons": "^32.0.0",
    "@syncfusion/ej2-react-inputs": "^32.0.0",
    "@syncfusion/ej2-react-dropdowns": "^32.0.0",
    "@syncfusion/ej2-react-grids": "^32.0.0",
    "@syncfusion/ej2-react-calendars": "^32.0.0",
    "@syncfusion/ej2-react-navigations": "^32.0.0",
    "@syncfusion/ej2-react-popups": "^32.0.0",
    "@syncfusion/ej2-react-layouts": "^32.0.0"
  }
}
```

Shared Infrastructure (Both Paths)

Form Field Adapters (Optional Add-On)

`src/components/ui/form-fields/` provides React Hook Form integrations:

Component	Wraps	Variant
<code>FormInput</code>	<code>Syncfusion Input</code>	<code>Syncfusion</code>
<code>FormSelect</code>	<code>Syncfusion Select</code>	<code>Syncfusion</code>
<code>FormDatePicker</code>	<code>Syncfusion DatePicker</code>	<code>Syncfusion</code>
<code>FormCheckbox</code>	<code>Syncfusion Checkbox</code>	<code>Syncfusion</code>
<code>FormNativeInput</code>	<code>InputNative</code>	<code>Native</code>
<code>FormNativeSelect</code>	<code>SelectNative</code>	<code>Native</code>
<code>FormNativeDatePicker</code>	<code>DatePickerNative</code>	<code>Native</code>
<code>FieldError</code>	Shared error display	<code>Both</code>

These depend on `react-hook-form` and optionally `zod` / `@hookform/resolvers`.

Recommendation: Ship as a separate subpath export ([/form-fields](#)) since they introduce additional peer dependencies.

Icons

[src/components/icons/](#) contains Feather icon wrappers and custom icon sets. These are standalone React components with no external dependencies. They can be included in either package or shipped separately.

Recommended Package Structure

Single Package with Subpath Exports

```
@your-scope/ui/
dist/
  native/          # Native components (tree-shakeable)
  syncfusion/      # Syncfusion wrappers (tree-shakeable)
  form-fields/    # React Hook Form adapters
  utils/           # cn(), is.*() utilities
  icons/           # Icon components
  styles/
    native.css     # Native-only styles
    syncfusion.css # Syncfusion styles (includes native base)
    variables.css  # CSS variables only (for custom implementations)
  tailwind-preset.js
```

```
{
  "name": "@your-scope/ui",
  "version": "1.0.0",
  "type": "module",
  "exports": {
    "./native": { "import": "./dist/native/index.js", "types": "./dist/native/index.d.ts" },
    "./syncfusion": { "import": "./dist/syncfusion/index.js", "types": "./dist/syncfusion/index.d.ts" },
    "./form-fields": { "import": "./dist/form-fields/index.js", "types": "./dist/form-fields/index.d.ts" },
    "./icons": { "import": "./dist/icons/index.js", "types": "./dist/icons/index.d.ts" },
    "./utils": { "import": "./dist/utils/index.js", "types": "./dist/utils/index.d.ts" },
    "./styles/native": "./dist/styles/native.css",
    "./styles/syncfusion": "./dist/styles/syncfusion.css",
    "./styles/variables": "./dist/styles/variables.css",
    "./tailwind-preset": "./dist/tailwind-preset.js"
  },
  "peerDependencies": {
    "react": "^18.0.0 || ^19.0.0",
    "react-dom": "^18.0.0 || ^19.0.0"
  },
  "peerDependenciesMeta": {
    "tailwindcss": { "optional": true },
    "@syncfusion/ej2-react-buttons": { "optional": true },
    "@syncfusion/ej2-react-inputs": { "optional": true },
    "@syncfusion/ej2-react-dropdowns": { "optional": true },
    "@syncfusion/ej2-react-grids": { "optional": true },
    "@syncfusion/ej2-react-calendars": { "optional": true },
    "@syncfusion/ej2-react-navigations": { "optional": true },
    "@syncfusion/ej2-react-popups": { "optional": true },
    "@syncfusion/ej2-react-layouts": { "optional": true },
    "react-hook-form": { "optional": true }
  }
}
```

```
    "zod": { "optional": true }
  }
}
```

Consumer Setup

Native Components Only

```
npm install @your-scope/ui
```

```
// tailwind.config.ts
import uiPreset from '@your-scope/ui/tailwind-preset';

export default {
  presets: [uiPreset],
  content: [
    './src/**/*.{js,ts,jsx,tsx}',
    './node_modules/@your-scope/ui/dist/**/*.js', // so Tailwind scans component classes
  ],
};
```

```
/* app.css */
@import '@your-scope/ui/styles/native';
@tailwind base;
@tailwind components;
@tailwind utilities;
```

```
// App.tsx
import { ButtonNative, AlertNative, TableNative } from '@your-scope/ui/native';

function App() {
  return <ButtonNative variant="primary" size="md">Click me</ButtonNative>;
}
```

Dark mode: Add class `dark` to a parent element (e.g., `<html class="dark">`). All CSS variables swap automatically.

Syncfusion Components

```
npm install @your-scope/ui @syncfusion/ej2-react-buttons @syncfusion/ej2-react-inputs # ... etc
```

```
/* app.css */
@import '@your-scope/ui/styles/syncfusion';
@tailwind base;
@tailwind components;
@tailwind utilities;
```

```
// App.tsx (after refactor to context-based)
import { ThemeProvider } from '@your-scope/ui/syncfusion';
import { Button, DataGrid } from '@your-scope/ui/syncfusion';

function App() {
  return (
    <ThemeProvider mode="light">
      <Button variant="primary" size="md">Click me</Button>
    </ThemeProvider>
  );
}
```

Migration Checklist

Phase 1: Foundation (Both Paths)

- Set up build tooling (`tsup` or Vite library mode)
- Configure path alias resolution for `@/` imports
- Extract `cn()` and `is.*()` into a `utils/` entry point
- Extract shared types from `src/components/ui/shared/` into package
- Create `tailwind-preset.js` from `tailwind.config.ts` theme.extend
- Set up automated type declaration generation

Phase 2: Native Package

- Create `styles/native.css` entry point (base + components + native-overrides only)
- Decouple `FM()` - add optional `t` prop to 16 components, default to English
- Move `SearchInput.tsx` out of shared types barrel
- Verify all native components build and tree-shake correctly
- Write package.json with subpath exports
- Test in a fresh consumer project

Phase 3: Syncfusion Package

- Refactor `useSyncfusionTheme()` from zustand store to React Context
- Create `ThemeProvider` that accepts `mode` prop and injects CSS variables
- Extract minimal theme injector (mode-only, no full `ThemeConfig`)
- Create `styles/syncfusion.css` entry point
- Externalize all `@syncfusion/ej2-react-*` as peer dependencies
- Verify wrappers build with Syncfusion packages externalized
- Test in a fresh consumer project with Syncfusion license

Phase 4: Optional Add-Ons

- Package form field adapters as `/form-fields` subpath
- Package icons as `/icons` subpath
- Add Storybook or docs site for component showcase

Why Syncfusion Extraction is Fundamentally Harder Than Native

The native and Syncfusion components look similar on the surface - both live in `src/components/ui/`, both use CSS variables, both share the same types. But the extraction difficulty is radically different because of **how they consume their styling**.

Native: Passive CSS Consumer (Easy)

A native component like `ButtonNative` is **completely passive** about its styling. Here is its entire dependency chain:

```

ButtonNative
└── imports: cn(), isValueDefined()           (2 tiny utility files)
└── imports: ButtonVariant, ButtonSize        (2 const enums from shared/)
└── reads:   CSS variables via class names    (passive - whatever :root has, it uses)
└── runtime: NOTHING                         (no JavaScript at mount time)

```

The component renders a `<button>` with CSS class names like `native-btn native-btn-primary native-btn-md`. Those classes are defined in `components.css` and reference CSS variables like `--component-button-primary-bg`. The variables have default values in `base.css`. **That's the entire story.** The component never touches the DOM directly, never reads from a store, never injects anything at runtime.

This means extraction is pure configuration work:

1. Bundle the source files
2. Ship the CSS
3. Done - it works in any React project that loads the CSS

Syncfusion: Active Runtime Theme System (Hard)

A Syncfusion component like `Button` has a completely different dependency chain:

```

Button (Syncfusion)
└── imports: ButtonComponent                  (@syncfusion/ej2-react-buttons - external)
└── imports: cn(), isValueDefined()           (2 tiny utility files)
└── imports: ButtonVariant, ButtonSize        (2 const enums from shared/)
└── imports: Mode                            (const enum from stores/mode.ts)
└── imports: useThemeStore                   (zustand global store)
    └── useThemeStore
        ├── zustand + devtools + persist middleware
        ├── DEFAULT_THEME (from stores/theme/defaultTheme.ts)
            └── FREMEN_THEME preset (200+ lines of color/config values)
                └── DEFAULT_* from stores/theme/defaults/ (27 files)
        ├── createThemeActions (from stores/theme/storeActions.ts)
            └── 10 action creator modules (from stores/theme/actions/ - 12 files)
        ├── injectThemeVariables (from stores/theme/themeInjector.ts)
            └── 12 specialized injector functions (from stores/theme/injectors/ - 16 files)
        ├── ThemeState interface (70+ actions defined)
            └── stores/theme/types/ (36 type definition files)
        └── Schema migration + deep merge + rehydration logic
└── runtime: calls useThemeStore() to read `mode` on every render

```

The critical difference: **the Syncfusion Button calls `useThemeStore()` directly at render time.** That single import transitively pulls in:

Dependency	Files	Why It Exists
<code>stores/theme/types/</code>	36 files	TypeScript interfaces for every configurable property (colors, spacing, typography, buttons, inputs, grids, cards, dialogs, alerts, badges, pagination, etc.)
<code>stores/theme/defaults/</code>	27 files	Default values for every component config, split by mode (light/dark) and category (buttons, forms, navigation, data display, feedback, pagination)
<code>stores/theme/actions/</code>	12 files	70+ store actions: <code>updateButtonConfig()</code> , <code>updateDataGridConfig()</code> , <code>updatePrimaryPalette()</code> , etc. Each action deep-merges partial updates and re-injects CSS variables
<code>stores/theme/injectors/</code>	16 files	Functions that call <code>document.documentElement.style.setProperty()</code> for every CSS variable. Specialized per domain: buttons, data grid, forms, navigation, data display, pagination, feedback, typography, colors, layout, animations

Dependency	Files	Why It Exists
<code>stores/theme/presets/</code>	20 files	Full theme presets (Fremen, Arctic, Copper, Emerald, Gold, Midnight, etc.) each defining 200+ config values
<code>useThemeStore.ts</code>	1 file	Zustand <code>create()</code> with <code>persist</code> middleware (<code>localStorage</code> , <code>devtools</code> , schema versioning (v1-v13), deep merge migration, and <code>onRehydrateStorage</code> that calls <code>injectThemeVariables()</code>)

Total: 120 files pulled in by a single `useThemeStore()` call.

The Core Problem: Singleton Global State

The theme store is a **zustand singleton** - one global instance created at module load time. It immediately:

1. Reads persisted state from `localStorage` (key: `theme-storage`)
2. Deep-merges it with `DEFAULT_THEME` to handle schema migrations (13 versions so far)
3. Calls `injectThemeVariables(theme, mode)` which uses `requestAnimationFrame` to write 100+ CSS variables to `document.documentElement.style`
4. Subscribes all connected components to re-render on any state change

This design is perfect for an app with a single theme store, but it creates three fundamental problems for a shared package:

Problem 1: Store collision. If two different UI projects in the same page (e.g., micro-frontends, iframes, or even just the consuming app using its own zustand) both import this package, they share the same singleton store, stomping on each other's theme.

Problem 2: Mandatory side effects. The store boots on import (zustand `create()` runs at module scope). Even importing a single `Button` triggers `localStorage` reads, schema migration logic, and DOM manipulation. A consumer cannot opt out.

Problem 3: Tight coupling to the full ThemeConfig. The `ThemeState` interface exposes 70+ update actions (`updateButtonConfig`, `updateDataGridConfig`, `updateSidebarConfig`, etc.). A consumer who just wants light/dark mode must ship the entire config system, all 20 presets, all 27 default files, and the full injection pipeline.

Why Native Avoids All of This

Native components never import the theme store. They use CSS classes that reference CSS variables:

```
/* components.css */
.native-btn-primary {
  background-color: var(--component-button-primary-bg);
  color: var(--component-button-primary-text);
}
```

The variables get their values from two possible sources:

1. **Static CSS defaults** in `base.css` (e.g., `--component-button-primary-bg: rgb(59 130 246)`) - always present
2. **Runtime injection** from the theme store's `injectThemeVariables()` - only if the app uses the theme editor

Source #1 is a plain CSS file. Source #2 is the app's concern, not the component's. The component never knows or cares where the CSS variable values come from. This is why native extraction is "just configuration" - you ship the CSS, the consumer loads it, done.

Syncfusion components bypass this clean separation. They reach directly into the JavaScript store to read `mode` and compute CSS class names like `sf-dark` or `sf-light` at render time. The styling is not purely CSS-driven - it has a mandatory JavaScript runtime dependency.

Summary of Difficulty Difference

Dimension	Native	Syncfusion
-----------	--------	------------

Dimension	Native	Syncfusion
Runtime dependencies	None beyond React	zustand + 120-file theme system
Side effects on import	None	Store creation, localStorage read, DOM injection
State management	None (pure CSS)	Global singleton zustand store
Styling mechanism	CSS classes read CSS variables passively	JavaScript reads store, computes class names at render
Dark mode	.dark CSS class on parent (consumer controls)	useThemeStore().mode read at render + CSS class toggle
Files to ship	~170 (components + types + utils + CSS)	~330 (everything above + 120 theme store files + Syncfusion CSS)
Refactoring needed	i18n decoupling (16 components)	Full store-to-context refactor (all 17 wrappers + theme system)
Consumer setup	Import CSS, use components	Import CSS, wrap app in <ThemeProvider>, install 8 Syncfusion packages
Risk of breaking the app	Low - CSS-only changes	High - touching the store/injector affects the entire theme editor

What Must Exist at the Project Root (and Why)

Today, [SyncfusionThemeStudio/](#) is an **application** that Vite builds into a [dist/](#) folder served as a website. There is no concept of "exporting components for external consumption". The following new root-level infrastructure is required to turn it into (or coexist with) a **library package**.

Root Files Needed

```
SyncfusionThemeStudio/
├── package.json
├── tsconfig.json
├── tsconfig.lib.json
├── tsup.config.ts
├── tailwind-preset.js
├── vite.config.ts
└── src/
    └── styles/
        ├── native.css
        └── syncfusion.css
```

← MODIFY (add exports, peerDependencies, files, sideEffects)
 ← MODIFY (add declarationDir, composite, emitDeclarationOnly)
 ← NEW (separate TS config for library compilation)
 ← NEW (library bundler config)
 ← NEW (extracted Tailwind theme for consumers)
 ← KEEP (app build remains unchanged)
 ← NEW (isolated entry point for native CSS only)
 ← NEW (isolated entry point for Syncfusion CSS only)

Why Each Root File is Needed

1. [tsup.config.ts](#) (NEW) - Library Bundler

Why: The project currently uses Vite to build a **single-page application** (`vite build` produces `dist/index.html` + chunked JS bundles). This is fundamentally different from building a **library** (which produces importable modules with no HTML entry point).

Vite can do library mode (`build.lib` in `vite.config.ts`), but `tsup` is better suited because:

- It handles multiple entry points natively (one per subpath export: `native/index.ts`, `syncfusion/index.ts`, etc.)
- It generates `.d.ts` type declarations automatically
- It resolves path aliases (`@/` → `./src/`) at build time without additional plugins

- It externalizes peer dependencies by default
- It outputs both ESM and CJS from a single config

What it does: Takes the `src/` source and produces a `dist/` folder with compiled JavaScript + type declarations, with `react`, `react-dom`, and all Syncfusion packages marked as external (not bundled).

```
// tsup.config.ts
import { defineConfig } from 'tsup';

export default defineConfig({
  entry: {
    'native/index': 'src/components/ui/native/index.ts',
    'syncfusion/index': 'src/components/ui/syncfusion/index.ts',
    'form-fields/index': 'src/components/ui/form-fields/index.ts',
    'utils/index': 'src/lib-exports/utils.ts',
    'icons/index': 'src/components/icons/index.ts',
  },
  format: ['esm'],
  dts: true,
  splitting: true,
  treeshake: true,
  external: [
    'react', 'react-dom',
    /^@syncfusion\/$/,
    'zustand',
    'react-hook-form', '@hookform/resolvers', 'zod',
    'i18next', 'react-i18next',
  ],
  esbuildOptions(options) {
    // Resolve @/ path aliases
    options.alias = { '@': './src' };
  },
});
```

Why it can't be the existing `vite.config.ts`: The Vite config is tuned for app mode - it has PWA plugins, manual chunk splitting for Syncfusion code-splitting, PostCSS with Tailwind, HTML entry point, dev server config, etc. Library builds have completely different requirements (no HTML, no code splitting into chunks, externalize everything). Mixing both in one config creates conflicts. Keeping them separate means `npm run build` continues to build the app, and a new `npm run build:lib` builds the library.

2. `tsconfig.lib.json` (NEW) - Library TypeScript Config

Why: The existing `tsconfig.json` is configured for app development - it includes test files, dev utilities, and compiles for Vite's module system. Library compilation needs different settings:

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "declaration": true,
    "declarationDir": "./dist/types",
    "emitDeclarationOnly": true,
    "rootDir": "./src"
  },
  "include": ["src/components/**/*", "src/utils/**/*", "src/lib/grid/**/*"],
  "exclude": ["**/*.test.*", "**/*.spec.*", "src/**/*stories.*"]
}
```

Why it can't be the existing `tsconfig.json`: The current config includes everything in `src/` (pages, layout, app entry point, stores, dev tools). Library builds must only include the files that are part of the package. Compiling the theme editor UI or the dev server

into `.d.ts` files would bloat the package and expose internal types consumers shouldn't see.

3. `package.json` (MODIFY) - Package Metadata

Why the current one doesn't work: The existing `package.json` has:

- No `exports` field (Node/bundlers don't know how to resolve subpath imports)
- No `files` field (an `npm publish` would include everything - 120 theme files, test fixtures, Playwright config, screenshots)
- No `peerDependencies` (React and Syncfusion would be bundled into the package, causing version conflicts)
- No `sideEffects` flag (bundlers can't tree-shake unused components)
- `"type": "module"` is already set (good)

What must change:

```
{
  "name": "@your-scope/ui",
  "exports": {
    "./native": { "import": "./dist/native/index.js", "types": "./dist/native/index.d.ts" },
    "./syncfusion": { "import": "./dist/syncfusion/index.js", "types": "./dist/syncfusion/index.d.ts" },
    "./form-fields": { "import": "./dist/form-fields/index.js", "types": "./dist/form-fields/index.d.ts" },
    "./styles/native": "./dist/styles/native.css",
    "./styles/syncfusion": "./dist/styles/syncfusion.css",
    "./styles/variables": "./dist/styles/variables.css",
    "./tailwind-preset": "./dist/tailwind-preset.js"
  },
  "files": ["dist/"],
  "sideEffects": ["*.css"],
  "peerDependencies": { "react": "^18 || ^19", "react-dom": "^18 || ^19" },
  "scripts": {
    "build:lib": "tsup && node scripts/copy-styles.js"
  }
}
```

Why `exports` matters: Without it, consumers can't do `import { ButtonNative } from '@your-scope/ui/native'`. Node's module resolution won't know what file to load. The `exports` map is how modern packages tell bundlers "this import path resolves to this file".

Why `files` matters: Without it, `npm pack` / `npm publish` includes the entire project directory. With `"files": ["dist/"]`, only the compiled output ships. This keeps the package size from 50MB (with `node_modules`, test fixtures, screenshots) down to < 1MB.

Why `sideEffects` matters: Without it, bundlers (webpack, Vite, Rollup) assume every module has side effects and cannot be tree-shaken. With `"sideEffects": ["*.css"]`, the bundler knows that JS modules are pure and can safely eliminate unused component code. Only CSS files are marked as having side effects (because they modify global styles when imported).

Why `peerDependencies` matters: Without it, the package bundles its own copy of React. If the consumer's app also uses React, there are now two React instances, which breaks hooks (`"Cannot update a component from inside the function body of a different component"`) and doubles bundle size. Peer dependencies tell npm "the consumer must provide React - don't install a second copy".

4. `tailwind-preset.js` (NEW) - Tailwind Theme Preset

Why: Components use Tailwind utility classes like `bg-primary-500`, `text-error-700`, `rounded-lg` that rely on custom theme mappings from `tailwind.config.ts`. These mappings connect Tailwind class names to CSS variables:

```
// tailwind.config.ts (current)
colors: {
```

```

primary: {
  500: 'rgb(var(--color-primary-500) / <alpha-value>)',
}
}

```

Without this mapping, `bg-primary-500` means nothing to a consumer's Tailwind instance. The consumer's Tailwind would generate no CSS for it, and components using those classes would have missing styles.

Why it must be a separate file: The current `tailwind.config.ts` includes app-specific values (`sidebar`, `header` spacing, `safelist` for the theme editor UI). Consumers should not get those. A preset contains only the theme mappings needed for components to render correctly.

```

// tailwind-preset.js
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: { 50: 'rgb(var(--color-primary-50) / <alpha-value>)', /* ...through 900 */ },
        success: { 50: '...', 500: '...', 700: '...' },
        warning: { 50: '...', 500: '...', 700: '...' },
        error: { 50: '...', 500: '...', 700: '...' },
        info: { 50: '...', 500: '...', 700: '...' },
        background: 'rgb(var(--color-background) / <alpha-value>)',
        surface: 'rgb(var(--color-surface) / <alpha-value>)',
        border: 'rgb(var(--color-border) / <alpha-value>)',
        'text-primary': 'rgb(var(--color-text-primary) / <alpha-value>)',
        'text-secondary': 'rgb(var(--color-text-secondary) / <alpha-value>)',
        'text-muted': 'rgb(var(--color-text-muted) / <alpha-value>)',
      },
      borderRadius: { /* maps to --radius-* vars */ },
      fontSize: { /* maps to --font-size-* vars */ },
      fontWeight: { /* maps to --font-weight-* vars */ },
    }
  }
};

```

5. `src/styles/native.css` and `src/styles/syncfusion.css` (NEW) - Isolated CSS Entry Points

Why: The current `src/styles/index.css` imports everything together:

```

/* Current index.css - monolithic */
@import '@syncfusion/ej2-base/styles/tailwind.css';           /* Syncfusion-only */
@import '@syncfusion/ej2-react-inputs/styles/tailwind.css';   /* Syncfusion-only */
@import '@syncfusion/ej2-react-buttons/styles/tailwind.css'; /* Syncfusion-only */
@import './layers/base.css';                                 /* Both need */
@import './layers/components.css';                           /* Native needs */
@import './layers/native-overrides.css';                     /* Native needs */
@import './layers/syncfusion-overrides.css';                /* Syncfusion needs */
@import './layers/syncfusion-themed.css';                   /* Syncfusion needs */

```

A consumer who only uses native components should not be forced to download 2,227 lines of Syncfusion CSS overrides (or install `@syncfusion/ej2-base` just for its base CSS). Conversely, a consumer who only uses Syncfusion wrappers doesn't need the 1,395 lines of `.native-btn-*` classes.

Separate entry points let consumers import exactly what they need:

```
/* src/styles/native.css */
@import './layers/base.css';
@import './layers/components.css';
@import './layers/native-overrides.css';
```

```
/* src/styles/syncfusion.css */
@import '@syncfusion/ej2-base/styles/tailwind.css';
@import '@syncfusion/ej2-react-inputs/styles/tailwind.css';
@import '@syncfusion/ej2-react-buttons/styles/tailwind.css';
@import './layers/base.css';
@import './layers/syncfusion-overrides.css';
@import './layers/syncfusion-themed.css';
```

6. Syncfusion-Only Additional Root Requirements

For the Syncfusion path, two more things are needed beyond what native requires:

`src/providers/ThemeProvider.tsx` (NEW) - Because the zustand store must be replaced with a React Context. This provider:

- Accepts a `mode` prop ('`light`' | '`dark`')
- Optionally accepts color overrides
- Calls a slim version of `injectThemeVariables()` to set CSS vars on mount/update
- Provides the mode value via React Context so `useSyncfusionTheme()` can read it without zustand

Slim injector extraction - The current `injectThemeVariables()` in `themeInjector.ts` accepts a full `ThemeConfig` (the interface with 13 top-level fields and hundreds of nested properties). For the package, a minimal injector is needed that only handles what consumers can reasonably configure: mode (light/dark) and optionally a primary color scale. The full 120-file theme system stays in the app, not the package.

Root Infrastructure Summary

File	Path A (Native)	Path B (Syncfusion)	Why
<code>tsup.config.ts</code>	Required	Required	No library build exists today; app builds produce HTML, not importable modules
<code>tsconfig.lib.json</code>	Required	Required	Library type declarations must exclude tests, pages, and app-specific code
<code>package.json</code> changes	Required	Required	No <code>exports</code> , <code>files</code> , <code>peerDependencies</code> , or <code>sideEffects</code> exist today
<code>tailwind-preset.js</code>	Required	Required	Consumers' Tailwind won't generate CSS for custom classes without the theme mappings
<code>src/styles/native.css</code>	Required	Not needed	Isolates native CSS from Syncfusion CSS
<code>src/styles/syncfusion.css</code>	Not needed	Required	Isolates Syncfusion CSS from native CSS
<code>src/providers/ThemeProvider.tsx</code>	Not needed	Required	Replaces zustand singleton with dependency-injectable context
Slim theme injector	Not needed	Required	Current injector requires the full 120-file ThemeConfig; package needs a minimal version

Effort Estimates

Phase	Complexity	Key Risk
Foundation	Medium	Build tooling + path alias resolution
Native package	Low-Medium	i18n decoupling is the main work; everything else is configuration
Syncfusion package	High	Zustand-to-Context refactor touches every wrapper component + theme system
Form field adapters	Low	Straightforward extraction

Recommendation: Start with **Phase 1 + Phase 2** (native-only). This gives you a usable package with minimal refactoring. Phase 3 (Syncfusion) requires significant architectural changes to the theme system and should be planned as a separate effort.

How the Project Was Built and Why It Needs Rework

Original Design Intent

SyncfusionThemeStudio was built as a **standalone single-page application** - a visual theme editor where designers tweak colors, typography, spacing, and component styles in real time and see the result immediately. It was never designed to export components for external consumption.

Every architectural decision followed from that goal:

1. Monolithic app build. Vite builds the project into `dist/index.html` + JS/CSS chunks. There is no library entry point because there was never a consumer - the app IS the product. The build pipeline produces an SPA with code-splitting for performance (Syncfusion CSS is lazy-loaded per component module via `loadSyncfusionCss()`), PWA support (`vite-plugin-pwa`), and manual chunk splitting to keep the initial bundle small despite Syncfusion's massive package sizes.

2. Global singleton theme store. The theme editor needs a single source of truth for the current theme - one place where all 200+ configurable properties live, persist to localStorage, and trigger re-renders across the entire UI when anything changes. Zustand with `persist` + `devtools` middleware was the natural choice. The store was designed for exactly one consumer: this app. It persists schema-versioned state to localStorage, deep-merges on hydration, and injects CSS variables into `document.documentElement.style` on every change. Components read from it directly because there is no boundary between "library code" and "app code" - it's all one app.

3. CSS variable injection at runtime. The theme editor lets users change any color, radius, font size, or component-specific style through the UI. Those changes must instantly reflect in every component. The solution: the theme store writes CSS variables to `:root` via `themeInjector.ts` (12 specialized injector functions across 16 files), and components consume those variables through CSS classes. This creates a live preview loop: user tweaks a slider -> store updates -> injector writes new CSS variable -> component re-renders with new style. This is the right architecture for a theme editor, but it bakes runtime JavaScript into what should be a pure CSS concern for a shared component library.

4. Tight path aliases. All imports use `@/` aliases (`@/utils/cn`, `@/stores/useThemeStore`, `@/components/ui/shared/buttonTypes`). These are resolved by both `tsconfig.json` (for TypeScript) and `vite.config.ts` (for the build). This works perfectly when everything lives in one project. It breaks the moment you try to compile a subset of the source into an independent package, because the alias resolver needs to know where `@/` points, and in a library build context, it points nowhere unless explicitly configured.

5. Monolithic CSS entry point. The main `src/styles/index.css` imports everything: Syncfusion base CSS, custom base variables, native component classes, Syncfusion overrides, and Syncfusion themed classes. It then declares a `@layer` ordering that controls specificity across all of them. This is correct for the app - it needs everything in one place with deterministic specificity. But it means you cannot import "just native CSS" or "just Syncfusion CSS" without pulling in the entire stack.

6. CSS layer specificity system. The project uses CSS `@layer` to solve a real problem: Syncfusion ships its own base styles, native components have their own styles, and the theme system overrides both. Without layers, specificity conflicts are constant. The layer declaration `@layer base, syncfusion-base, components, syncfusion-overrides, utilities` ensures that overrides always win over base styles regardless of selector specificity. This is essential for correctness but creates a strict ordering contract that consumers must replicate.

7. Lazy-loaded Syncfusion CSS. Beyond the three statically imported Syncfusion stylesheets (base, inputs, buttons), the remaining Syncfusion CSS (grids, calendars, navigations, popups, dropdowns, layouts, notifications) is loaded on demand via `loadSyncfusionCss()`. This reduces the initial CSS payload by deferring large stylesheets until the component that needs them is actually rendered. The lazy loader uses a module-level `Set` to track which modules are already loaded and prevents duplicate imports. This code-splitting strategy is app-specific and would need to be rethought for a package where the consumer controls when and how CSS is loaded.

What Needs to Change for Native Components

The native path is the simpler extraction because native components were built with a clean separation: they are pure React + CSS, never touch JavaScript state, and style themselves entirely through CSS classes that reference CSS variables. The changes are mostly about **packaging and configuration**, not refactoring component internals.

Change	What Exists Today	What Must Change	Why
Build system	Vite SPA build (<code>vite build -> dist/index.html</code>)	Add <code>tsup</code> library build (<code>tsup -> dist/native/index.js + .d.ts</code>)	A consumer needs importable ES modules, not an HTML page
Path aliases	<code>@/utils/cn</code> resolved by Vite + tsconfig at dev/build time	<code>tsup</code> alias config or pre-build rewrite to relative paths	Aliases don't resolve when the code lives in <code>node_modules/@scope/ui/</code>
CSS entry point	<code>src/styles/index.css</code> imports everything including Syncfusion	New <code>src/styles/native.css</code> importing only <code>base.css + components.css + native-overrides.css</code>	Consumer should not need <code>@syncfusion/ej2-base</code> installed just to import CSS
Tailwind config	<code>tailwind.config.ts</code> with app-specific values (sidebar/header spacing, safelist for theme editor palette display)	New <code>tailwind-preset.js</code> with only the color/radius/font/weight mappings	Consumer's Tailwind must generate CSS for classes like <code>bg-primary-500</code> , but shouldn't inherit the app's sidebar widths
Localization	16 components import <code>FM()</code> from <code>src/localization/helpers.ts</code> , which calls <code>i18n.t()</code> on a pre-configured i18next instance	Either: (A) add optional <code>t</code> function prop with English fallback, or (B) export a <code>configureLocalization()</code> that consumers call	<code>FM()</code> will crash at runtime if i18next isn't initialized. A shared package cannot assume i18next exists in the consumer's app
Package metadata	<code>package.json</code> has no <code>exports</code> , no <code>files</code> , no <code>peerDependencies</code> , no <code>sideEffects</code>	Add all four fields	Without these, the package can't be imported by subpath, publishes 50MB of test fixtures, bundles its own React copy, and can't be tree-shaken
Type declarations	<code>tsconfig.json</code> compiles everything (pages, stores, dev tools)	New <code>tsconfig.lib.json</code> that only includes component/util/type files	Library <code>.d.ts</code> output must not expose internal app types
SearchInput in shared	<code>src/components/ui/shared/SearchInput.tsx</code> is a React component inside the types barrel	Move to <code>src/components/ui/native/</code> or a separate barrel	Types barrels should only contain types and enums, not React components

What does NOT need to change: The components themselves. `ButtonNative`, `InputNative`, `TableNative`, etc. do not need code modifications (except the 16 that use `FM()`). Their internal logic, props interfaces, CSS class references, and rendering behavior are already correct for external consumption. The work is entirely infrastructure.

What Needs to Change for Syncfusion Components

The Syncfusion path requires everything from the native path **plus significant refactoring of component internals and the theme system**. The changes fall into three categories: the same infrastructure work as native, the store-to-context refactor, and the CSS isolation work.

Same Infrastructure as Native

All the build system, path alias, package metadata, type declaration, and Tailwind preset changes from the native path apply identically. These are table stakes for any package extraction.

Store-to-Context Refactor (the Hard Part)

This is the single largest piece of work and the reason Syncfusion extraction is rated "High" complexity.

Change	What Exists Today	What Must Change	Why
Theme consumption	Every Syncfusion wrapper calls <code>useThemeStore()</code> directly to read <code>mode</code>	Create a <code>ThemeContext</code> + <code>ThemeProvider</code> that accepts <code>mode</code> as a prop; change <code>useSyncfusionTheme()</code> to read from context instead of Zustand	The Zustand store is a module-scoped singleton. It cannot coexist with a consumer's own state management. Two apps on the same page would collide. Consumers must be able to control <code>mode</code> however they want (Redux, Context, URL param, cookie, etc.)
Theme injection	<code>themeInjector.ts</code> accepts a full <code>ThemeConfig</code> (13 top-level fields, hundreds of nested properties) and writes 100+ CSS variables to <code>:root</code> via 12 specialized injector functions	Create a slim injector that only handles mode (light/dark) and optionally a primary color palette	The full <code>ThemeConfig</code> interface exists to power the theme editor UI. A consumer who just wants light/dark buttons shouldn't need to ship 120 files of theme configuration
Store boot side effects	<code>useThemeStore</code> is created at module scope via <code>create()</code> . On import, it reads localStorage, runs migration, calls <code>injectThemeVariables()</code>	The package entry point must have zero side effects. <code>ThemeProvider</code> handles initialization explicitly when mounted	Module-scope side effects break server-side rendering (no <code>document</code>), break tree-shaking (bundler can't eliminate the import), and break test isolation (store state leaks between tests)
Default theme	<code>DEFAULT_THEME</code> is imported from <code>stores/theme/defaultTheme.ts</code> which imports the Fremen preset	The package needs a minimal default (just light/dark mode colors) or no default at all (consumer provides everything)	The Fremen preset imports from <code>stores/theme/defaults/</code> (27 files), which imports from <code>stores/theme/types/</code> (36 files). Shipping a "default theme" transitively pulls in the entire theme system
Theme presets	20 preset files (Arctic, Copper, Emerald, Fremen, Gold, etc.) in <code>stores/theme/presets/</code>	Do not ship presets. They are app-specific. If consumers want presets, export them as an optional subpath (<code>/presets</code>)	Presets are 200+ lines each with hand-tuned colors for the theme editor. They add ~4,000 lines that consumers using the package for UI components will never use
Store actions	<code>ThemeState</code> exposes 70+ mutation actions (<code>updateButtonConfig</code> , <code>updateDataGridConfig</code> , <code>updatePrimaryPalette</code> , etc.) assembled from 10 action creator modules	None of these ship. The <code>ThemeProvider</code> is read-only from the component's perspective - it receives <code>mode</code> as a prop, components read it via context	The 70+ actions exist for the theme editor's control panel. Shared components don't edit their own theme - they render with whatever mode the consumer provides

The refactor touches **all 17 Syncfusion wrapper components** because every one of them imports from the theme system:

```
// Current (every Syncfusion wrapper)
import { useThemeStore } from '@/stores/useThemeStore';
import { Mode } from '@/stores/mode';
// ...
const { mode } = useThemeStore();
const modeClass = mode === Mode.Dark ? 'sf-dark' : 'sf-light';
```

Each must change to:

```
// After refactor
import { useSyncfusionTheme } from '../hooks/useSyncfusionTheme';
// ...
const { modeClass } = useSyncfusionTheme(); // reads from React Context, not zustand
```

The `useSyncfusionTheme()` hook itself changes from reading `useThemeStore().mode` to reading `useContext(ThemeContext).mode`. This is a small code change per component, but it must be done carefully across all 17 wrappers and verified against the theme editor (which continues to use the zustand store directly).

Syncfusion CSS Isolation

Change	What Exists Today	What Must Change	Why
Base Syncfusion CSS	3 Syncfusion stylesheets imported statically in <code>index.css</code> + 6 more lazy-loaded via <code>loadSyncfusionCss()</code>	New <code>src/styles/syncfusion.css</code> that imports all needed Syncfusion base CSS statically	Lazy loading is an app optimization that doesn't make sense for a package. The consumer's bundler handles code splitting. The package just needs to declare which CSS files are needed
Override CSS	<code>syncfusion-overrides.css</code> (1,946 lines) overrides Syncfusion's default styles to match the theme system	Ship as-is, but must be importable independently without native component CSS	Syncfusion components render incorrectly without these overrides (wrong fonts, wrong colors, wrong spacing). They are not optional
Themed CSS	<code>syncfusion-themed.css</code> (281 lines) defines <code>.sf-button</code> , <code>.sf-btn-primary</code> , <code>.sf-dark</code> , <code>.sf-light</code> etc.	Ship as-is, same isolation requirement	These classes are what the wrapper components apply. Without them, <code>cssClass="sf-button sf-dark sf-btn-primary"</code> resolves to nothing
Layer ordering	<code>@layer base, syncfusion-base, components, syncfusion-overrides, utilities</code> declared in <code>index.css</code>	Must be declared in the package's <code>syncfusion.css</code> entry point AND documented for consumers	If the consumer's app uses CSS layers and declares a different order, Syncfusion overrides may not win over base styles, causing visual breakage

Critical: CSS Layer Ordering and Style Loading

Why Order Matters

The component library uses CSS `@layer` declarations to control specificity without relying on selector weight or source order hacks. This solves a real problem: Syncfusion ships its own base styles (`.e-btn { ... }`, `.e-grid { ... }`), the native components have their own styles (`.native-btn { ... }`), and the theme system needs to override both.

Without layers, you would need increasingly specific selectors to win:

```
/* Syncfusion ships this (specificity: 0,1,0) */
.e-btn { background-color: #317ab9; }

/* You need to override it (must be MORE specific) */
.sf-themed .e-btn.e-primary { background-color: var(--component-button-primary-bg); }
```

With layers, the declaration order determines the winner regardless of selector specificity:

```
@layer syncfusion-base {
  .e-btn { background-color: #317ab9; }           /* Layer 2: always loses */
}

@layer syncfusion-overrides {
  .e-btn { background-color: var(--component-button-primary-bg); } /* Layer 4: always wins */
}
```

The Layer Order Contract

The project declares this layer order in [src/styles/index.css](#):

```
@layer base, syncfusion-base, components, syncfusion-overrides, utilities;
```

What each layer contains and why it's in that position:

Position	Layer	Lines	Contains	Why This Position
1	base	395	CSS custom property declarations (:root { --color-primary-500: ... }), base HTML resets, focus styles, scrollbar styles	Must load first because everything else references these variables
2	syncfusion- base	~3,000+	Syncfusion's own shipped stylesheets (@syncfusion/ej2- base, ej2-react-inputs, ej2- react-buttons, etc.)	Must load after base (so it inherits reset styles) but before our overrides
3	components	1,395	Native component classes .native-btn, .native-select, .native-input, etc.)	Must load after Syncfusion base (so they don't get stomped by Syncfusion's global selectors) but before Syncfusion overrides (which are specific to Syncfusion wrappers)
4	syncfusion- overrides	2,227	Our overrides of Syncfusion defaults + themed class definitions .sf-button, .sf-dark, etc.)	Must load after Syncfusion base so that our var(--component-*) values override Syncfusion's hardcoded colors. This is the key specificity win
5	utilities	(Tailwind)	Tailwind utility classes (bg- primary-500, rounded-lg, text- sm, etc.)	Must be last so that utility classes can override any component style (e.g., className="native-btn bg-red-500" makes the override work)

What the Consumer Must Do

Native-Only Consumer

The native CSS entry point ships with the layer declaration built-in. The consumer's [app.css](#) must load it **before** Tailwind directives:

```
/* app.css - CORRECT ORDER */
@import '@your-scope/ui/styles/native'; /* 1. Package CSS (includes @layer declaration) */
@tailwind base; /* 2. Tailwind base */
@tailwind components; /* 3. Tailwind components */
@tailwind utilities; /* 4. Tailwind utilities (must be last) */
```

If the consumer loads Tailwind first, then the package CSS, Tailwind's `@layer base` will be declared before the package's `@layer base`, and the two will merge with the package's base rules appearing first - which is correct. But if the consumer declares their OWN `@layer` order, they must ensure `base` comes before `components` comes before `utilities`:

```
/* app.css - WRONG: consumer redeclares layers in wrong order */
@layer utilities, base, components; /* Utilities first = broken! */
@import '@your-scope/ui/styles/native'; /* Package layer declaration ignored */
```

The first `@layer` declaration in the document wins. If the consumer's app declares layers before the package CSS is loaded, the consumer's order takes precedence. This can silently break component styling if the order is wrong.

Recommendation: Document clearly that the package CSS import must be the **first CSS import** in the consumer's entry stylesheet, before any `@layer` declarations or Tailwind directives.

Syncfusion Consumer

The Syncfusion CSS entry point is more sensitive because it must control the relationship between Syncfusion's base styles and our overrides:

```
/* app.css - CORRECT ORDER */
@import '@your-scope/ui/styles/syncfusion'; /* 1. Package CSS (Syncfusion base + overrides + layer
declaration) */
@tailwind base; /* 2. Tailwind base */
@tailwind components; /* 3. Tailwind components */
@tailwind utilities; /* 4. Tailwind utilities */
```

The `syncfusion.css` entry point internally imports Syncfusion's base stylesheets and then the override layers. This order is critical:

```
/* What syncfusion.css does internally */
@import '@syncfusion/ej2-base/styles/tailwind.css'; /* -> goes into syncfusion-base layer */
*/
@import '@syncfusion/ej2-react-inputs/styles/tailwind.css'; /* -> goes into syncfusion-base layer */
*/
@import '@syncfusion/ej2-react-buttons/styles/tailwind.css'; /* -> goes into syncfusion-base layer */
*/
@import './layers/base.css'; /* -> goes into base layer */
@import './layers/syncfusion-overrides.css';
layer */ /* -> goes into syncfusion-overrides
layer */
@import './layers/syncfusion-themed.css';
layer */ /* -> goes into syncfusion-overrides
layer */

@layer base, syncfusion-base, components, syncfusion-overrides, utilities;
```

If a consumer also lazy-loads additional Syncfusion CSS (e.g., for grids or calendars), that CSS must land in the `syncfusion-base` layer, not outside of it. Otherwise it will have higher specificity than the overrides and the theme won't apply to those components. The package should document this and optionally provide a helper:

```
// Consumer lazy-loads grid CSS - must specify layer
import '@syncfusion/ej2-react-grids/styles/tailwind.css'; // needs to be in syncfusion-base layer
```

Both Paths: Common Pitfalls

Pitfall	What Goes Wrong	Fix
Consumer imports package CSS after Tailwind directives	Tailwind's <code>@layer</code> declaration runs first, package layers merge into Tailwind's order instead of the package's intended order	Always import package CSS first
Consumer declares their own <code>@layer</code> before package CSS	Their layer order takes precedence, potentially putting <code>utilities</code> before <code>components</code> (breaks utility overrides) or <code>base</code> after <code>overrides</code> (breaks variable references)	Don't redeclare <code>@layer</code> order. Use the package's declaration or ensure it's compatible
Consumer uses CSS Modules or Styled Components alongside package CSS	CSS Modules scope class names, so <code>.native-btn</code> becomes <code>.ButtonNative_native-btn_a3x2</code> . The package's CSS won't match	Package components must be used with global CSS, not CSS Modules. This is already the case since components use <code>className</code> strings
Consumer loads Syncfusion CSS from a different version	Syncfusion v32 base styles might differ from v33. Overrides were written for a specific version	Pin Syncfusion peer dependency range tightly. Document tested version
Consumer uses Tailwind v4 (CSS-first config) instead of v3	Tailwind v4 uses <code>@theme</code> instead of <code>@layer</code> . The preset format and layer interop are different	Provide separate preset files for v3 and v4, or document minimum version

Style Loading Summary

LOAD ORDER (top = loads first, must be first in consumer's app.css)

1. `@import '@your-scope/ui/styles/native'` ← OR syncfusion (not both)

```
@layer base          (CSS variables + resets)
@layer syncfusion-base (Syncfusion defaults) *
@layer components    (native component classes)
@layer syncfusion-overrides (theme overrides) *
```

* syncfusion.css only

* syncfusion.css only

2. `@tailwind base;` ← Tailwind's base resets

3. `@tailwind components;` ← Tailwind's component classes

4. `@tailwind utilities;` ← Tailwind utilities (MUST be last)

```
@layer utilities      (bg-*, text-*, rounded-*, etc)
These can override ANY component style via className
```

5. Consumer's own CSS ← App-specific styles (optional)

If the consumer imports both native and Syncfusion styles (because they use components from both), they should import the combined entry point or import both in order:

```
@import '@your-scope/ui/styles/native';
@import '@your-scope/ui/styles/syncfusion';
@tailwind base;
@tailwind components;
@tailwind utilities;
```

This works because both entry points share the same `base.css` (it will be included twice but CSS deduplication handles it), and the `@layer` declarations are additive - the first one to appear wins, and subsequent declarations just add rules to existing layers.