To run your React app in multiple spots, you have three options:

1. **iframe** — Embed the todo app in the invoice app via an <iframe>.
2. **Reusable App Component** — Share the entire todo app via npm.
3. **Reusable UI Component** — Share the todo app’s markup via npm.

**Approach 1: iFrame**

But leads to multiple issues:

1. If the two apps display the same data, you risk them getting out of sync.
2. If the two apps use the same data, you end up making redundant API calls to fetch the same data.
3. You can’t customize the iframed app’s behavior.
4. If a different team owns the framed in app, when they push to production, you’re instantly affected.

# Approach 2: Reusable App Component

# Sharing your app via npm instead of an iframe avoids issue #4 above, but the other issues remain. API, auth, and behavior are all baked in.

**Approach 3: Reusable UI Components**

I recommend a more granular approach using two flexible building blocks:

1. “Dumb” React components (Just UI. No API calls inside.)
2. API wrappers

# Dumb” React components are configurable, unopinionated, and composable. They’re reusable UI. When consuming a “dumb” component like this, you are free to provide the relevant data or specify the API calls the app should make.

However, if you’re going to compose “dumb” components, you need to wire up the same API calls for multiple apps. This is where API wrappers come in handy. What’s an API wrapper? It’s a JavaScript file full of functions that make HTTP calls to your API. My team creates API wrappers. We use [Axios](https://github.com/axios/axios" \t "_blank) behind the scenes to make the HTTP calls.

Imagine you have a user API. Here’s how to create a user API wrapper:

1. Create a JS file with public functions like getUserById, saveUser, etc. Each function accepts the relevant parameters and uses Axios/Fetch to make HTTP calls to your API.
2. Share the wrapper as an npm package called userApi.

Here’s an example.

|  |  |
| --- | --- |
|  | /\* This API wrapper is useful because it: |
|  | 1. Centralizes our Axios default configuration. |
|  | 2. Abstracts away the logic for determining the baseURL. |
|  | 3. Provides a clear, easily consumable list of JavaScript functions |
|  | for interacting with the API. This keeps API calls short and consistent. |
|  | \*/ |
|  | import axios from 'axios'; |
|  |  |
|  | let api = null; |
|  |  |
|  | function getInitializedApi() { |
|  | if (api) return api; // return initialized api if already initialized. |
|  | return (api = axios.create({ |
|  | baseURL: getBaseUrl(), |
|  | responseType: 'json', |
|  | withCredentials: true |
|  | })); |
|  | } |
|  |  |
|  | // Helper functions |
|  | function getBaseUrl() { |
|  | // Insert logic here to get the baseURL by either: |
|  | // 1. Sniffing the URL to determine the environment we're running in. |
|  | // 2. Looking for an environment variable as part of the build process. |
|  | } |
|  |  |
|  | function get(url) { |
|  | return getInitializedApi().get(url); |
|  | } |
|  |  |
|  | function post(url, data) { |
|  | return getInitializedApi().post(url, data); |
|  | } |
|  |  |
|  | // Public functions |
|  | // Note how short these are due to the centralized config and helpers above. 😎 |
|  | export function getUserById(id) { |
|  | return get(`user/${id}`); |
|  | } |
|  |  |
|  | export function saveUser(user) { |
|  | return post(`user/${user.id}`, {user: user}); |
|  | } |

[**view raw**](https://gist.github.com/coryhouse/09fb49dd0c13ca20cca6cc0fe2438f3e/raw/ceb68883b2412e7037457ebd4dc49ec1e6b2857f/userApi.js)[**userApi.js**](https://gist.github.com/coryhouse/09fb49dd0c13ca20cca6cc0fe2438f3e#file-userapi-js) hosted with ❤ by **[GitHub](https://github.com/)**

On my team, we share our React components and API wrappers as [private packages on npm](https://www.npmjs.com/pricing), but [Artifactory](https://jfrog.com/artifactory/" \t "_blank) is a nice alternative.

These Lego blocks give us the foundation for quickly building new solutions out of reusable pieces.

A highly composable system provides components that can be selected and assembled in various combinations to satisfy specific user requirements. — [Wikipedia](https://en.wikipedia.org/wiki/Composability)

So ideally, your “dumb” reusable UI component is composed of [other reusable components, also shared on npm](https://app.pluralsight.com/library/courses/react-creating-reusable-components/table-of-contents)!

With reusable React components and API wrappers published to npm, it’s trivial to build something awesome.

It’s like snapping Lego pieces together. 👍