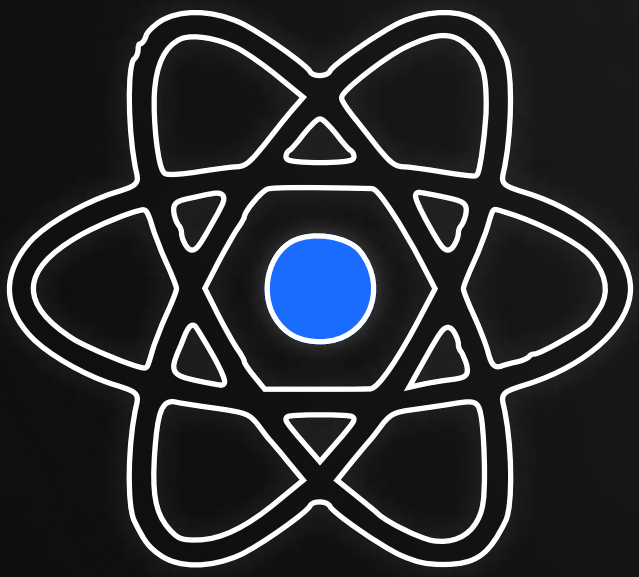


Best Practices for React.JS in 2024

sindo.dev

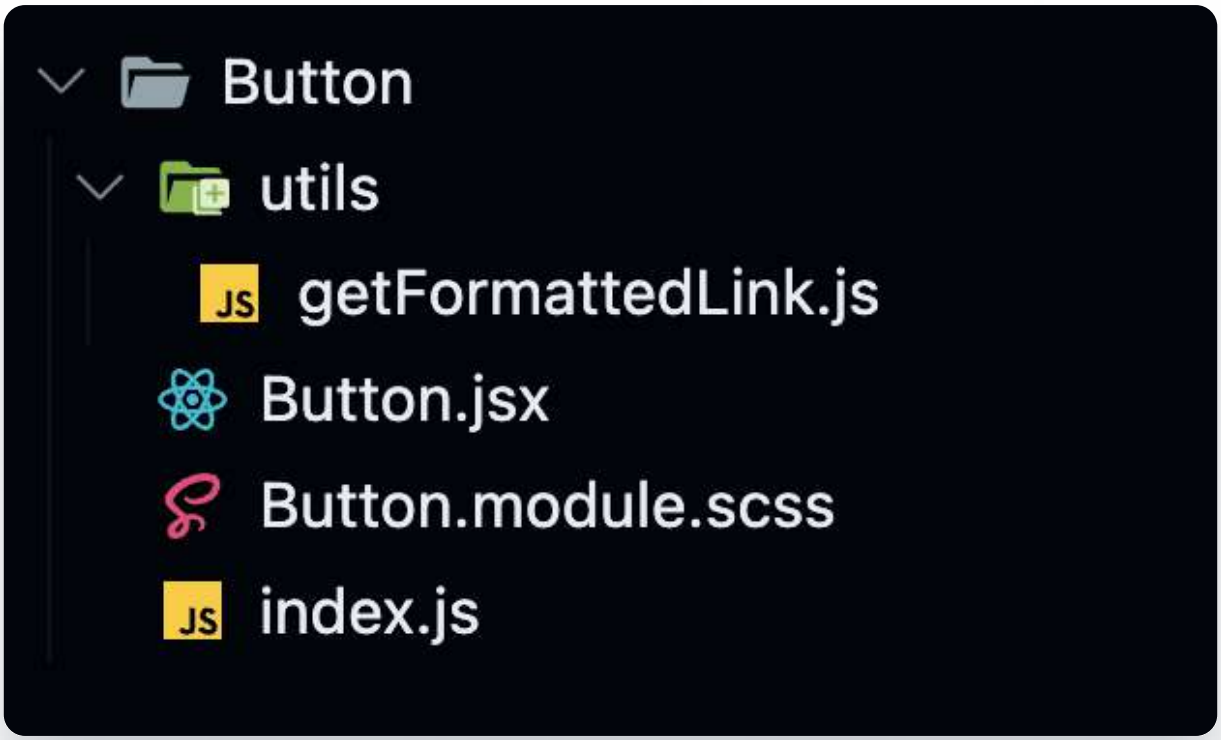
Introduction

This is a free e-book written by Sindo. To access even more e-books just like this one visit sindo.dev.

Table of contents

Colocation

Colocation refers to the practice of keeping related code files close to each other on the file system. In the case of React components, this means creating a folder for each component that contains all the relevant files, including the component code, styling, and utility functions. For example, for a **Button** component, you would create a Button folder that contains *Button.tsx* for the component code, *Button.css* for the styling, and *index.ts* to export the component as a default export for use in other files. If the Button component also has utility functions that are only used in that component, you would create a utils folder inside the Button folder and add the function files there.



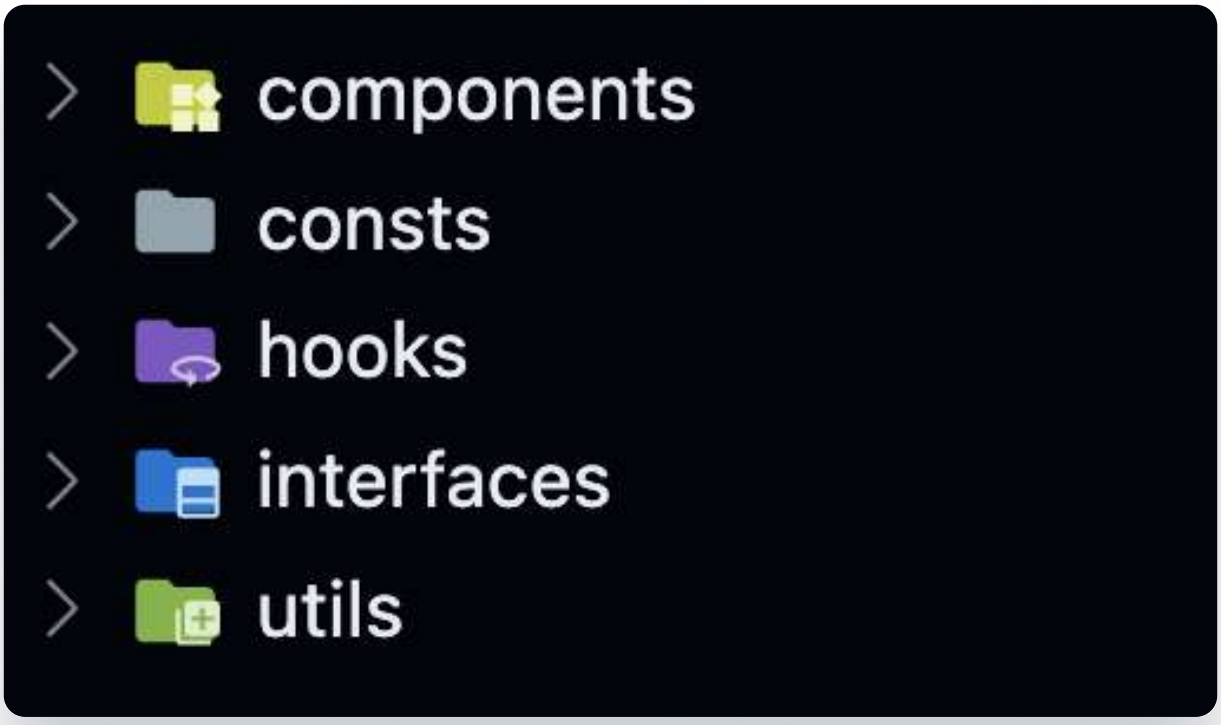
Colocation

By keeping all the files related to a component in the same folder, you can easily locate and modify them. This also helps to minimize the number of files you need to navigate and manage, making it easier to maintain and refactor your code. This folder file and folder structure allows you to instantly see and organise all items related to a specific feature or component.

Once you understand the concept of colocation, you can take it a step further and combine it with separation of concerns (discussed in the next chapter).

Separation of concerns

Separation of concerns is the principle of dividing code into logical and independent units, such as components, services, and utilities, to improve code reusability and testability. By separating concerns, you can create components that are more modular and easier to reason about. For example, you can create a service to handle API calls and a utility to handle string manipulation. This approach can make your code more scalable, since you can easily reuse code in different parts of your application, and can make it easier to test, since you can isolate and test each concern separately.

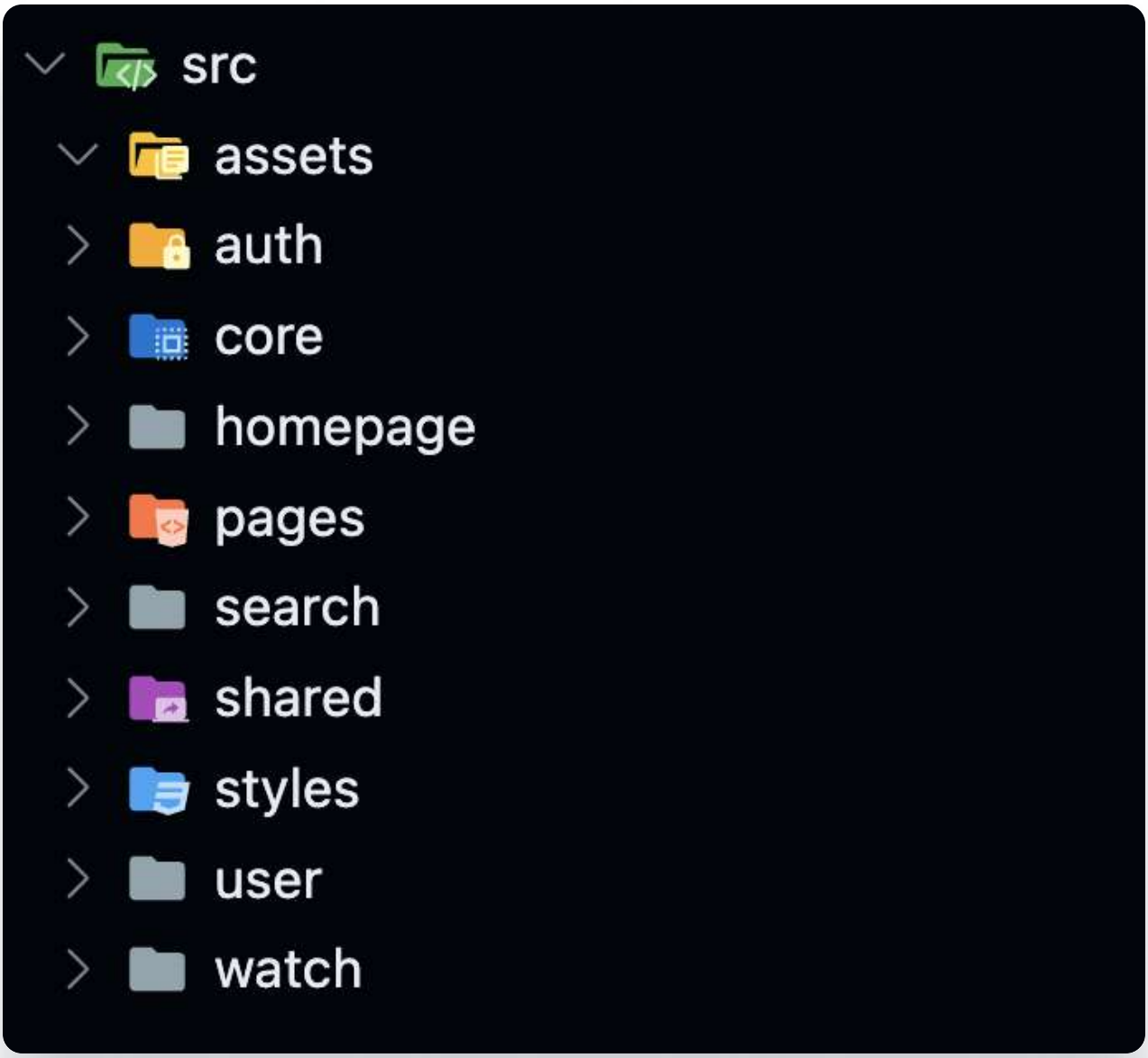


Separation of concerns +

Once you understand and make use of the separation of concerns principle, it’s time to make your entire codebase follow the convention - based on features.
Instead of having the general outline discussed earlier, we can subdivide this further by taking into account the different features our app contains.

Separation of concerns ++

For example - if we are writing a Netflix clone, we would have the auth feature, watch feature, search feature, profile feature, and so on. We would also have some components, utils, constants etc. that would be used across multiple features. Here's how we would handle that:

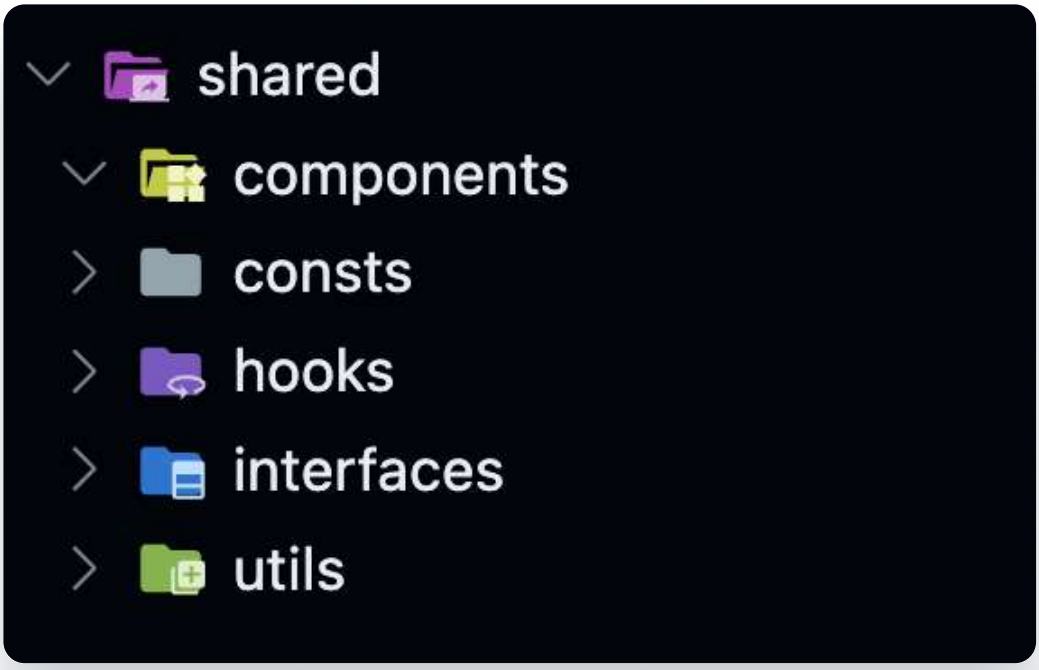


Separation of concerns ++

In this example every feature folder contains the same folder structure as outlined earlier. You'll also notice some folders are not features (*core* and *shared*).

Core folder - contains components, contexts, utils etc. used in the root of the application (for example in the *App.jsx file*).

Shared folder - contains components, consts, hooks, etc. that are shared across multiple features.



ES6

ES6 is a version of JavaScript that introduced many new language features, such as arrow functions, destructuring, and template literals, which can make React code more concise and expressive. For example, arrow functions can simplify the syntax of event handlers and map functions, while destructuring can make it easier to extract data from objects and arrays. Template literals can improve code readability by allowing for string interpolation and multi-line strings. By using ES6, you can write more modern and efficient code that is easier to read and maintain.

ES6

To ensure our project follows the latest standards make sure to:

- Use => arrow => functions instead of classes for components
- Destructuring props inside the function definition
- Set defaults for function arguments instead of performing *if* checks
- Use modules (import, export) instead of common JS files (*require*)

Here's a small component that follows these rules:

```jsx
import React from 'react';
import './ProfileCard.css';

const ProfileCard = ({ firstName, lastName, imageSrc, bio = '', ...rest }) => {
    console.log(rest); // Any additional props passed to the component
    return (
        <div className="profile-card">
            <h1>
                {firstName} {lastName}
            </h1>
            <span>{rest.initials}</span>
            <img src={imageSrc} alt="User profile image" />
            <p>{bio}</p>
        </div>
    );
};

export default ProfileCard;
```

Proper array mapping

Proper array mapping is a best practice for rendering lists in React, where you should use a unique key for each item in the list, rather than relying on the index of the item. This helps React efficiently update the DOM and avoids errors that can occur when the index of an item changes. By providing a unique key, React can more easily determine which items have changed and which have not. Additionally, using keys that are meaningful and stable can improve code readability and make it easier to debug problems.

For example, if we're mapping over a todo list, use the todo's ID instead of the index.

```
return (
    <div className="todos">
        {todos.map((todo) => (
            <div key={todo.id}>
                <h3>{todo.title}</h3>
            </div>
        ))}
    </div>
);
```

Fragments

Fragments are a React component that allows you to group a list of child elements without adding extra nodes to the DOM, which can improve performance and accessibility. By using a Fragment, you can avoid adding unnecessary divs to your markup, which can make your HTML cleaner and easier to style. In addition, using Fragments can improve accessibility by reducing the number of elements that screen readers need to read.

```
return (
    <>
        <div className="item-1">
            <span>Stuff inside item #1</span>
        </div>
        <div className="item-2">
            <span>Stuff inside item #2</span>
        </div>
    </>
);
```

Promises *vs* Async/Await

Promises and async/await are both mechanisms for working with asynchronous code in JavaScript. Promises are a way to handle asynchronous operations by chaining *then()* methods to handle resolved values and *catch()* methods to handle errors. Async/await is a more recent syntax that was introduced in ES2017, and it provides a more natural way to write asynchronous code that looks similar to synchronous code.

While both approaches are valid and have their own benefits, it is generally recommended to use async/await over promises when possible. Async/await can make asynchronous code more readable and easier to follow, especially for developers who are new to working with promises or asynchronous code. Additionally, async/await can simplify error handling by using traditional try/catch blocks.

Promises *vs* Async/Await

It is important to note that async/await is not a new technology, but rather a syntax that makes working with Promises easier. It is sometimes referred to as *"syntactic sugar"* because it does not provide any new functionality, but rather simplifies the way existing functionality is used. It is still important to understand how Promises work, as they are the underlying mechanism used by async/await.

Destructuring and ...spread

Destructuring and spread are two powerful features in ES6 that can simplify your code and make it more readable. Destructuring is a way to extract data from objects and arrays into variables using a concise syntax, while spread is a way to spread the contents of an array or object into a new array or object. Destructuring can be used to extract only the properties you need from an object, which can simplify code and reduce the amount of typing needed. For example, instead of accessing a property using *obj.prop*, you can destructure the property into a variable like *const { prop } = obj*. Similarly, you can use destructuring to extract values from an array and assign them to variables.

```
1  // Destructuring
2
3  const user = {
4      name: 'Sindo',
5      seniority: 'Senior',
6      hotel: 'Trivago',
7  };
8
9  const { name } = user; // We now have a "name" variable
```

Destructuring and ...spread

Spread is useful for combining arrays and objects or creating copies of them. For example, you can use the spread operator (...) to create a new array that contains the elements of two arrays like *const newArray = [...array1, ...array2]*. Similarly, you can create a new object that contains the properties of two objects by using spread like *const newObj = {...obj1, ...obj2}*.
Using destructuring and spread can make your code more concise and easier to read, as well as reduce the amount of repetitive code you need to write. However, it is important to use them appropriately and avoid overusing them, as this can make your code harder to follow and maintain.

```
1  // Spread operator
2
3  const names = ["Sindo", "Doky", "Bob"]; // ["Sindo", "Doky", "Bob"]
4  const namesWithBear = ["Bear", ...names]; // ["Bear", "Sindo", "Doky", "Bob"]
```

Pro Tip

Destructuring and ...spread

It's extremely important to keep memory in mind when using a spread operator.
If your goal is to modify an array, don't make the mistake of duplicating the array just with the intention of modifying it. Instead, if possible, try and modify the array in place. Otherwise, an entirely new array will be created in memory, effectively doubling your application's memory usage, and in more complex projects or inside loops, the performance difference will be staggering.

Note: Don't forget that React state may not be mutated, so don't try modifying your state array in place! You will undoubtably need to create a copy of it if you want to modify it.

```
// Adding items to an array

const names = ['Sindo', 'Doky', 'Bob'];
// names: ["Sindo", "Doky", "Bob"]
const additionalNames = [...names, 'The Rock'];
// additionalNames: ["Sindo", "Doky", "Bob", "The Rock"]
// Two arrays are now stored in memory

const companies = ['Google', 'Amazon', 'Facebook'];
companies.push('Meta', 'Tesla', 'Microsoft');
// companies: ["Google", "Amazon", "Facebook", "Meta", "Tesla", "Microsoft"]
// Only one array is now stored in memory
```

Template literals

Template literals are a feature in ES6 that allow you to include dynamic content in strings using placeholders, which can make your code more readable and maintainable. By using template literals, you can avoid concatenating strings with variables or hardcoding HTML markup, which can improve code clarity and reduce the risk of errors. Template literals also support multiline strings, which can be helpful when formatting longer text. Additionally, template literals can be used to define HTML templates that can be rendered dynamically with React (*albeit used rarely*).

```
1  const seasonNumber = 1;
2  const episodeNumber = 3;
3  const seasonAndEpisode = `Season ${seasonNumber} Episode ${episodeNumber}`;
4  console.log(seasonAndEpisode); // Season 1 Episode 3
```

Useful SVG Trick

SVGs are a type of image format that can be easily customized and styled with CSS. When using SVGs in React, it is a good practice to set the fill color using the *"currentColor"* keyword, which allows the SVG to inherit the color of its parent element. This can simplify the CSS styling and make it easier to change the fill color of the SVG dynamically, for example, in response to user input or state changes. Using *"currentColor"* can also improve accessibility by allowing users to customize the fill color of the SVG through their browser or operating system settings.
Keep in mind that you want to render most SVG's as images in your app, and should only use this approach when you need to modify the color of the SVG on the fly. Rendering the SVG as an image will decrease the count of rendered nodes in your HTML, leading to higher performance.

```jsx
import * as React from 'react';
const CloudIcon = (props) => (
  <svg xmlns="http://www.w3.org/2000/svg" width={800} height={800} fill="none" viewBox="0 0 24 24" {...props}>
    <path
      stroke="currentColor" // This inherits the CSS text color
      strokeLinecap="round"
      strokeLinejoin="round"
      strokeWidth={2}
      d="M3 13.65C3 16.603 5.418 19 8.4 19h8.1c2.485
      0 4.5-2.016 4.5-4.503 0-1.847-1.11-3.552-2.7-4.247C18.132
      7.323 15.684 5 12.69 5 10.35 5 8.346 6.486 7.5 8.5 4.8
      8.938 3 11.2 3 13.65Z"
    />
  </svg>
);
export default SvgComponent;
```

Linters

A linter is a tool that analyzes your code and highlights potential errors or issues, such as syntax errors, undefined variables, or unused code. Linters can help you catch bugs and improve code quality, consistency, and readability. In React, it is common to use a linter such as ESLint, which can be configured to enforce best practices and coding standards specific to React. Linters can also help you identify areas of your code that can be improved or simplified, such as by suggesting alternative syntax or highlighting opportunities for refactoring.

Having a linter can also ensure all code that is pushed to a remote repository is following a certain ruleset, which is especially useful when working with a team of people.

Example of linter errors and warnings:

```
./src/watch/category/components/CategoryVideoGrid/CategoryVideoGrid.tsx
25:47  Warning: '_id' is defined but never used.  @typescript-eslint/no-unused-vars
26:11  Error: Missing "key" prop for element in iterator  react/jsx-key
```

TypeScript

TypeScript is a superset of JavaScript that adds static typing, which can help catch errors earlier and provide better code documentation and understanding. TypeScript can be used with React to provide improved type safety, reduce runtime errors, and provide more accurate code completion and refactoring tools. Additionally, TypeScript can help make code more maintainable by providing a more explicit and concise syntax, which can help improve code readability and reduce the likelihood of bugs. While there is a learning curve to adopting TypeScript, it is a valuable tool for improving the quality and maintainability of your code.

```tsx
interface Props {
  title: string;
  categories?: string;
  href?: string;
  // Some more advanced types
  sanityImage?: SanityImageSource;
  options?: UseNextSanityImageOptions;
}

export const PartnerCard: React.FC<Props> = ({ title, categories, href, ...rest }) => {
  // ... rest of the component
};
```

Thank you!

Thank you for downloading this resource! I hope you find it valuable. For more resources like this one, be sure to keep an eye on https://sindo.dev as new content is regularly added.

Interested in working together?

Website: ***sindo.dev***
Email: ***sindoonyt@gmail.com***