## **Table of Contents**

[**Table of Contents 1**](#_6b7phpejk6p5)

[**Initial Steps 2**](#_othxlry4xcip)

[Use Typescript 2](#_jmewwzu9nb0o)

[Use Eslint 3](#_kirzc0w81ois)

[**Components 4**](#_xvdd6jrfyz7z)

[Favor Functional Components 4](#_3kqhgs8meero)

[Write Consistent Components 6](#_fsqu8ili5il2)

[Name Components 6](#_j7ich0nlzu11)

[Organize Helper Functions 6](#_qv0w1124vf6)

[Don't Hardcode Repetitive Markup 8](#_kujp8z1uke6m)

[Manage Component Size 10](#_yjh39vzfqusn)

[Write Comments in JSX 10](#_pr4v3uscrbfu)

[Use Error Boundaries 11](#_xpfakjyrbx09)

[Destructure Props 12](#_10z4jv8x2u2b)

[Manage the Number of Props 12](#_84nbndtd684t)

[Pass Objects Instead of Primitives 13](#_47tiybwj6v52)

[Conditional Rendering 13](#_pb7zpay8z8c)

[Avoid Nested Ternary Operators 14](#_pbqa0p90bb61)

[Move Lists in a Separate Component 14](#_niz0kbmj6o4v)

[Assign Default Props When Destructuring 15](#_927xji1uj60a)

[Avoid Nested Render Functions 16](#_q8liyd921ob2)

[Avoid prop drilling 16](#_6zx5iai0np0e)

[**State Management 19**](#_bcjcdlog46c0)

[Use Reducers 19](#_10iedaff42e)

[Prefer Hooks to HOCs and Render Props 20](#_o2mxb6eehnv1)

[Use Data Fetching Libraries 22](#_uq5fpvc5fb1t)

[State Management Libraries 22](#_5jmnop4rufih)

[**Component Mental Models 23**](#_d3mg0vobzv64)

[Container & Presentational 23](#_yp59sfegssbv)

[Stateless & Stateful 23](#_mh083b3ztnyf)

[**Application Structure 24**](#_vpmavxqo2mpk)

[Group by Route/Module 24](#_1y6pkedqkk4j)

[Create a Common Module 25](#_wpzzgis8oubi)

[Use Absolute Paths 25](#_mkpjd2i1z36g)

[Wrap External Components 26](#_ts12k2wvql50)

[Move Components in Folders 27](#_l4ryavz30yil)

[**Performance 27**](#_64wwp8z5um0z)

[Don't Optimize Prematurely 27](#_yib20d67ujgn)

[Watch The Bundle Size 28](#_w9m2goua3bxv)

[Use Code splitting 28](#_v63vyh2d7bh1)

[Code Splitting in React - Using React.lazy() and Suspense 29](#_6fe55r8oo620)

[Code Splitting by Routes 29](#_grld6bqapgza)

[Rerenders - Callbacks, Arrays and Objects 30](#_x3u0pkh751et)

[**Testing 31**](#_i7f84223befz)

[Don't Rely on Snapshot Tests 31](#_kumkm6bvtx45)

[Test Correct Rendering 31](#_xy1ypfvalmxy)

[Validate State & Events 31](#_i691yjtnbzhq)

[Test Edge Cases 32](#_qfkaaacbd0je)

[Write Integration Tests 32](#_gzoz14smumt4)

[**Styling 32**](#_iq0irak9pmm9)

[Use CSS-in-JS 32](#_enpdx5d1bkxs)

[Keep Styled Components Together 33](#_qh3brii3ty7r)

[Tailwind CSS 33](#_9vdlfox5gr7k)

[SASS/SCSS 33](#_rsdyy02rmzt3)

[**Data Fetching 34**](#_96bhkq6a3dh8)

[Use a Data Fetching Library 34](#_rrrg4jc7xz8p)

## 

## **Initial Steps**

### **Use Typescript**

TypeScript is a superset of JavaScript which makes a lot of sense when used with React because TypeScript allows for strongly typed JavaScript code with various code IntelliSense.

There are various advantages of using React with TypeScript over JavaScript, some of the bigger advantages are:

1. Variable and bug checks

TypeScript can identify bugs before even compiling the code using the IntelliSense which saves us a lot of time.

2. Readable, easily understandable code

Since TypeScript is a strongly typed language, there are types assigned for variables, functions, classes and properties which makes the code much more readable and easy to understand.

3. Interfaces

Complex type definitions can be defined using interfaces in TypeScript which allows strict type checks while writing complex object definitions.

4. Better support for JSX

TypeScript compiles JSX directly to JavaScript without the use of Babel.

5. IDE support

IDEs like VS Code make languages like TypeScript all the more powerful with code refactoring, code snippets extensions and real time type checking.

6. Support for existing React projects

Adopting TypeScript is easy, as files can be incrementally upgraded without causing issues throughout the rest of your project.

### **Use Eslint**

ESLint is a user interface tool with the ability to identify and report patterns found in ECMAScript / JavaScript code to make code more consistent and avoid errors. This is beneficial because ESLint identifies ways to make code better, and you don’t need or don’t agree with specific rules, they can be changed or ignored (either for the line, for the whole file, or the entire project). ESLint is designed so that all rules can be fully enabled, making it easier for developers to identify suspicious code.

1. You will find bugs and errors before they happen.
2. Your code (and your team’s code) will be more consistent.
3. ESLint is completely pluggable, every single rule is a plugin, and you can add more at runtime.

**Custom config for eslint:**

It can be configured in package.json or in a separate file .eslintrc.json

| {  "eslintConfig": {  "extends": ["react-app", "shared-config"],  "rules": {  "additional-rule": "warn"  },  "overrides": [  {  "files": ["\*\*/\*.ts?(x)"],  "rules": {  "additional-typescript-only-rule": "warn"  }  }  ]  } } |
| --- |

## 

## 

## **Components**

### **Favor Functional Components**

Favor functional components - they have a simpler syntax. No lifecycle methods, constructors or boilerplate. You can express the same logic with fewer characters without losing readability.

Unless you need an error boundary they should be your go-to approach. The mental model you need to keep in your head is a lot smaller.

| // 👎 Class components are verbose class Counter extends React.Component {  state = {  counter: 0,  }   constructor(props) {  super(props)  this.handleClick = this.handleClick.bind(this)  }   handleClick() {  this.setState({ counter: this.state.counter + 1 })  }   render() {  return (  <**div**>  <**p**>counter: {this.state.counter}</**p**>  <**button** onClick={this.handleClick}>Increment</**button**>  </**div**>  )  } }  // 👍 Functional components are easier to read and maintain function Counter() {  const [counter, setCounter] = useState(0)   handleClick = () => setCounter(counter + 1)   return (  <**div**>  <**p**>counter: {counter}</**p**>  <**button** onClick={handleClick}>Increment</**button**>  </**div**>  ) } |
| --- |

### 

### 

### 

### 

### **Write Consistent Components**

Stick to the same style for your components. Put helper functions in the same place, export the same way and follow the same naming patterns.

There isn’t a real benefit of one approach over the other

No matter if you’re exporting at the bottom of the file or directly in the definition of the component, pick one and stick to it.

### **Name Components**

Always name your components. It helps when you’re reading an error stack trace and using the React Dev Tools.

It’s also easier to find where you are when developing if the component’s name is inside the file.

| // 👎 Avoid this export default () => <**form**>...</**form**>  // 👍 Name your functions export default function Form() {  return <**form**>...</**form**> } |
| --- |

### 

### **Organize Helper Functions**

Helper functions that don’t need to hold a closure over the components should be moved outside. The ideal place is before the component definition so the file can be readable from top to bottom.

That reduces the noise in the component and leaves inside only those that need to be there.

| // 👎 Avoid nesting functions which don't need to hold a closure. function Component({ date }) {  function parseDate(rawDate) {  ...  }   return <**div**>Date is {parseDate(date)}</**div**> }  // 👍 Place the helper functions after the component function Component({ date }) {  return <**div**>Date is {parseDate(date)}</**div**> }  function parseDate(date) {  ... } |
| --- |

You want to keep the least amount of helper functions inside the definition. Move as many as possible outside and pass the values from state as arguments.

Composing your logic out of pure functions that rely only on input makes it easier to track bugs and extend.

| // 👎 Helper functions shouldn't read from the component's state export default function Component() {  const [value, setValue] = useState('')   function isValid() {  // ...  }   return (  <>  <input  value={value}  onChange={e => setValue(e.target.value)}  onBlur={validateInput}  />  <button  onClick={() => {  if (isValid) {  // ...  }  }}  >  Submit  </button>  </>  ) }  // 👍 Extract them and pass only the values they need export default function Component() {  const [value, setValue] = useState('')   return (  <>  <input  value={value}  onChange={e => setValue(e.target.value)}  onBlur={validateInput}  />  <button  onClick={() => {  if (isValid(value)) {  // ...  }  }}  >  Submit  </button>  </>  ) }  function isValid(value) {  // ... } |
| --- |

### **Don't Hardcode Repetitive Markup**

Don’t hardcode markup for navigation, filters or lists. Use a configuration object and loop through the items instead.

This means you only have to change the markup and items in a single place.

| // 👎 Hardcoded markup is harder to manage. function Filters({ onFilterClick }) {  return (  <>  <**p**>Book Genres</**p**>  <**ul**>  <**li**>  <**div** onClick={() => onFilterClick('fiction')}>Fiction</**div**>  </**li**>  <**li**>  <**div** onClick={() => onFilterClick('classics')}>  Classics  </**div**>  </**li**>  <**li**>  <**div** onClick={() => onFilterClick('fantasy')}>Fantasy</**div**>  </**li**>  <**li**>  <**div** onClick={() => onFilterClick('romance')}>Romance</**div**>  </**li**>  </**ul**>  </>  ) }  // 👍 Use loops and configuration objects const GENRES = [  {  identifier: 'fiction',  name: Fiction,  },  {  identifier: 'classics',  name: Classics,  },  {  identifier: 'fantasy',  name: Fantasy,  },  {  identifier: 'romance',  name: Romance,  }, ]  function Filters({ onFilterClick }) {  return (  <>  <**p**>Book Genres</**p**>  <**ul**>  {GENRES.map((genre) => (  <**li**>  <**div** onClick={() => onFilterClick(genre.identifier)}>  {genre.name}  </**div**>  </**li**>  ))}  </**ul**>  </>  ) } |
| --- |

### **Manage Component Size**

A React component is just a function that gets props and returns markup. They adhere to the same software design principles.

If a function is doing too many things, extract some of the logic and call another function. It’s the same with components - if you have too much functionality, split it in smaller components and call them instead.

If a part of the markup is complex, requires loops and conditionals - extract it.

Rely on props and callbacks for communication and data. Lines of code are not an objective measure. Think about responsibilities and abstractions instead.

### **Write Comments in JSX**

When something needs more clarity open a code block and provide the additional information just like you would in a regular function. The markup is a part of the logic so when you feel that something needs more clarity - provide it.

Business logic is always coupled to the markup, at least a little. So we should provide any context about the domain that may not be obvious.

| function Component(props) {  return (  <>  {/\* Subscribers should not see any ads. \*/}  {user.subscribed ? null : <**Advert** />}  </>  ) } |
| --- |

### **Use Error Boundaries**

An error in one component shouldn’t bring down the entire UI. There are rare cases in which we want to take down the whole page or redirect if a critical error happens. Most of the time we’d be fine if we just hide a specific element from the screen.

In a function that deals with data you may have multiple try/catch statements. Put error boundaries to use not just on the top level. Wrap elements on the screen that can exist separately to avoid cascading failures.

| function Component() {  return (  <**Layout**>  <**ErrorBoundary**>  <**CardWidget** />  </**ErrorBoundary**>   <**ErrorBoundary**>  <**FiltersWidget** />  </**ErrorBoundary**>   <**div**>  <**ErrorBoundary**>  <**ProductList** />  </**ErrorBoundary**>  </**div**>  </**Layout**>  ) } |
| --- |

### **Destructure Props**

Most React components are just functions. They get props and return markup. In a normal function you use the arguments it is passed directly, so it makes sense to apply the same principle here. No need to repeat props everywhere.

A reason not to destructure the props would be to distinguish what is external and what is internal state. But in a regular function there is no distinction between arguments and variables. Don’t create unnecessary patterns.

| // 👎 Don't repeat props everywhere in your component function Input(props) {  return <**input** value={props.value} onChange={props.onChange} /> }  // 👍 Destructure and use the values directly function Component({ value, onChange }) {  const [state, setState] = useState('')   return <**div**>...</**div**> } |
| --- |

### **Manage the Number of Props**

The question of how many props a component should receive is a subjective one. The number of props that a component has is correlated to how much it’s doing. The more props you pass to it the more responsibilities it has.

A high number of props is a signal that a component is doing too much.

If I go above 5 props I consider whether this component should be split. In some cases, it may just need a lot of data. An input field, for example, may have a lot of props. In others it’s a sign that something needs to be extracted.

Note: The more props a component takes, the more reasons to rerender.

### 

### 

### **Pass Objects Instead of Primitives**

One way to limit the amount of props is to pass an object instead of primitive values. Rather than passing down the user name, email and settings one by one you can group them together. This also reduces the changes that need to be done if the user gets an extra field for example.

Using TypeScript makes this even easier.

| *// 👎 Don't pass values on by one if they're related* <UserProfile  bio={user.bio}  name={user.name}  email={user.email}  subscription={user.subscription} />  *// 👍 Use an object that holds all of them instead* <UserProfile user={user} /> |
| --- |

### **Conditional Rendering**

In some situations using short circuit operators for conditional rendering may backfire and you may end up with an unwanted 0 in your UI. To avoid this default to using ternary operators. The only caveat is that they’re more verbose.

The short-circuit operator reduces the amount of code which is always nice. Ternaries are more verbose but there is no chance to get it wrong. Plus, adding the alternative condition is less of a change.

| **// 👎 Try to avoid short-circuit operators function Component() {  const count = 0  return <div>{count && <h1>Messages: {count}</h1>}</div> }  // 👍 Use a ternary instead function Component() {  const count = 0 return <div>{count ? <h1>Messages: {count}</h1> : null}</div>}** |
| --- |

### **Avoid Nested Ternary Operators**

Ternary operators become hard to read after the first level. Even if they seem to save space at the time, it’s better to be explicit and obvious in your intentions.

| *// 👎 Nested ternaries are hard to read in JSX* isSubscribed ? (  <ArticleRecommendations /> ) : isRegistered ? (  <SubscribeCallToAction /> ) : (  <RegisterCallToAction /> )  // 👍 Place them inside a component on their own function CallToActionWidget({ subscribed, registered }) {  if (subscribed) {  return <ArticleRecommendations />  }   if (registered) {  return <SubscribeCallToAction />  }   return <RegisterCallToAction /> }  function Component() {  return (  <CallToActionWidget subscribed={subscribed} registered={registered}/>   ) } |
| --- |

### **Move Lists in a Separate Component**

Looping through a list of items is a common occurrence, usually done with the map function. However, in a component that has a lot of markup, the extra indentation and the syntax of map don’t help with readability.

When you need to map over elements, extract them in their own listing component, even if the markup isn’t much. The parent component doesn’t need to know about the details, only that it’s displaying a list.

Only keep a loop in the markup if the component’s main responsibility is to display it. Try to keep a single mapping per component but if the markup is long or complicated, extract the list either way.

| function Component({ topic, page, articles, onNextPage }) {  return (  <**div**>  <**h1**>{topic}</**h1**>  <**ArticlesList** articles={articles} />  <**div**>You are on page {page}</**div**>  <**button** onClick={onNextPage}>Next</**button**>  </**div**>  ) } |
| --- |

### **Assign Default Props When Destructuring**

One way to specify default prop values is to attach a defaultProps property to the component. This means that the component function and the values for its arguments are not going to sit together.

Prefer assigning default values directly when you’re destructuring the props. It makes it easier to read the code from top to bottom without jumping and keeps the definitions and values together.

| // 👎 Don't define the default props outside of the function function Component({ title, tags, subscribed }) {  return <**div**>...</**div**> }  Component.defaultProps = {  title: '',  tags: [],  subscribed: false, }  // 👍 Place them in the arguments list function Component({ title = '', tags = [], subscribed = false }) {  return <**div**>...</**div**> } |
| --- |

### 

### **Avoid Nested Render Functions**

When you need to extract markup from a component or logic, don’t put it in a function living in the same component. A component is just a function. Defining it this way is nesting inside its parent.

This means that it will have access to all the state and data of its parent. It makes the code more unreadable - what is this function doing in between all the components?

Move it in its own component, name it and rely on props instead of a closure.

| // 👎 Don't write nested render functions function Component() {  function renderHeader() {  return <**header**>...</**header**>  }  return <**div**>{renderHeader()}</**div**> }  // 👍 Extract it in its own component import Header from '@modules/common/components/Header'  function Component() {  return (  <**div**>  <**Header** />  </**div**>  )} |
| --- |

## 

### **Avoid prop drilling**

Anyone who has worked in React would have faced this and if not then will face it definitely. Prop drilling is basically a situation when the same data is being sent at almost every level due to requirements in the final level.

|  |
| --- |

| // 👎 With Prop drilling  import React, { useState } from "react";   export default function Parent() {  const [fName, setfName] = useState("firstName");  const [lName, setlName] = useState("LastName");  return (  <>  <div>This is a Parent component</div>  <br />  <ChildA fName={fName} lName={lName} />  </>  ); }   function ChildA({ fName, lName }) {  return (  <>  This is ChildA Component.  <br />  <ChildB fName={fName} lName={lName} />  </>  ); }   function ChildB({ fName, lName }) {  return (  <>  This is ChildB Component.  <br />  <ChildC fName={fName} lName={lName} />  </>  ); }   function ChildC({ fName, lName }) {  return (  <>  This is ChildC component.  <br />  <h3> Data from Parent component is as follows:</h3>  <h4>{fName}</h4>  <h4>{lName}</h4>  </>  ); }; |
| --- |

The problem with Prop Drilling is that whenever data from the Parent component will be needed, it would have to come from each level, Regardless of the fact that it is not needed there and simply needed in the end.

A better alternative to this is using useContext hook. The useContext hook is based on Context API and works on the mechanism of Provider and Consumer. Provider needs to wrap components inside Provider Components in which data has to be consumed. Then in those components, using the useContext hook that data needs to be consumed.

// 👍 With Context API

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

let context = React.createContext(null);

function Parent() {

const [fName, setfName] = useState("firstName");

const [lName, setlName] = useState("LastName");

return (

<context.Provider value={{ fName, lName }}>

<div>This is a Parent component</div>

<br />

<ChildA />

</context.Provider>

);

}

function ChildA() {

return (

<>

This is ChildA Component.

<br />

<ChildB />

</>

);

}

function ChildB() {

return (

<>

This is ChildB Component.

<br />

<ChildC />

</>

);

}

function ChildC() {

const { fName, lName } = useContext(context);

return (

<>

This is ChildC component.

<br />

<h3> Data from Parent component is as follows:</h3>

<h4>{fName}</h4>

<h4>{lName}</h4>

</>

);

}

export default Parent;

## **State Management**

### **Use Reducers**

Sometimes you need a more powerful way to express and manage state changes. Start with useReducer before you reach for an external library. This is a great mechanism to do complex state management and it doesn’t require 3rd party dependencies.

In combination with React’s Context and TypeScript, useReducer can be really powerful. If you need multiple pieces of state, move them to a reducer instead.

| // 👎 Don't use too many separate pieces of state const TYPES = {  SMALL: 'small',  MEDIUM: 'medium',  LARGE: 'large' }  function Component() {  const [isOpen, setIsOpen] = useState(false)  const [type, setType] = useState(TYPES.LARGE)  const [phone, setPhone] = useState('')  const [email, setEmail] = useState('')  const [error, setError] = useSatte(null)   return (  ...  ) }  // 👍 Unify them in a reducer instead const TYPES = {  SMALL: 'small',  MEDIUM: 'medium',  LARGE: 'large' }  const initialState = {  isOpen: false,  type: TYPES.LARGE,  phone: '',  email: '',  error: null }  const reducer = (state, action) => {  switch (action.type) {  ...  default:  return state  } }  function Component() {  const [state, dispatch] = useReducer(reducer, initialState)   return (  … ) } |
| --- |

### **Prefer Hooks to HOCs and Render Props**

In some cases we need to enhance a component or give it access to an external state. In general there are three ways to do this - higher order components (HOCs), render props and hooks.

Hooks have proven to be the most efficient way to achieve such composition. From a philosophical standpoint, a component is a function that **uses** other functions. Hooks allow you to tap into multiple sources of external functionality without them conflicting with each other. No matter the number of hooks, you know where each value comes from.

With HOCs you get values as props. This makes it unclear whether they come from the parent component or the wrapping one. Also, chaining multiple props together is known to cause errors.

Render props lead to high indentation and bad readability. Nesting multiple components with render props in the same tree makes the markup look even worse. Also it only exposes the values in the markup itself so you have to write the logic there or pass it down.

With hooks you work with simple values, which are easy to track and don’t interfere with the JSX.

| // 👎 Avoid using render props function Component() {  return (  <>  <Header />  <Form>  {({ values, setValue }) => (  <input  value={values.name}  onChange={e => setValue('name', e.target.value)}  />  <input  value={values.password}  onChange={e => setValue('password', e.target.value)}  />  )}  </Form>  <Footer />  </>  ) }  // 👍 Favor hooks for their simplicity and readability function Component() {  const [values, setValue] = useForm()   return (  <>  <Header />  <input  value={values.name}  onChange={e => setValue('name', e.target.value)}  />  <input  value={values.password}  onChange={e => setValue('password', e.target.value)}  />  )}  <Footer />  </>  ) } |
| --- |

### **Use Data Fetching Libraries**

Often the data that we want to manage in state is retrieved from an API. We need to keep that data in memory, update it and access it in multiple places.

Modern data fetching libraries like [React Query](https://react-query.tanstack.com/) provide enough mechanisms to manage the external data. We can cache it, invalidate it and fetch it anew. They can be used for sending data as well, triggering a refresh for another piece of data.

It’s even easier if you’re working with a GraphQL client like [Apollo](https://www.apollographql.com/docs/react/). It has the concept of client state built in.

### **State Management Libraries**

In most cases you don’t need a state management library. They should be used in large applications that require managing complex state. There are plenty of state managers we can use but I would prefer [Redux](https://redux.js.org/).

Also to make redux asynchronous, we can use thunk or saga as middleware.

## **Component Mental Models**

### **Container & Presentational**

The predominant line of thinking is to split components in two groups - presentational and container components. Also known as smart and dumb.

The idea is that some components don’t have any functionality and state. They are just called by the parent component with some props. The container components contain the business logic, do the data fetching and manage the state.

This mental model is what the MVC structure is for back-end applications. It’s generic enough to work everywhere and you can’t go wrong with it.

**But**, in modern UI applications that pattern falls short. Pulling all the logic in a few components leads to bloat. They end up with too many responsibilities and become hard to manage. As an app grows, putting the complexity in a few concentrated places is just not good for maintainability.

### **Stateless & Stateful**

Think of components as stateful and stateless. The mental model mentioned above implies that a few components should be managing a lot of the complexity. Instead, it should be spread throughout the app.

Data should live close to where it is used. When you’re using a GraphQL client you fetch the data in the component that displays it. Even if it’s not a top level one. Don’t think about **containers**, think about **responsibilities**. Consider what is the most logical component to hold a piece of state.

For example, a <Form /> component should own the data of the form. An <Input /> should be receiving values and calling callbacks when a change occurs. A <Button /> should notify the form that it was pressed and let the form handle what happens.

Who does the validation in a form? Is the input field responsible? That would mean that this component becomes aware of the business logic of your application. How will it notify the form that there is an error? How will that error state be refreshed? Will the form know about that? If there’s an error but you try to submit, what will happen?

When faced with questions like this you should become aware that responsibilities are getting mixed up. In this case it’s better for the input to be left stateless and receive an error message from the form.

## **Application Structure**

### **Group by Route/Module**

Grouping by containers and components makes applications hard to navigate. To understand what component belongs where you need a good level of familiarity.

Not all components are equal - some are used globally, others are made for a specific part of the app. This structure works well for the smallest of projects. But anything that goes beyond a handful of components becomes hard to manage.

| // 👎 Don't group by technical details ├── containers | ├── Dashboard.jsx | ├── Details.jsx ├── components | ├── Table.jsx | ├── Form.jsx | ├── Button.jsx | ├── Input.jsx | ├── Sidebar.jsx | ├── ItemCard.jsx  // 👍 Group by module/domain ├── modules | ├── common | | ├── components | | | ├── Button.jsx | | | ├── Input.jsx | ├── dashboard | | ├── components | | | ├── Table.jsx | | | ├── Sidebar.jsx | ├── details | | ├── components | | | ├── Form.jsx | | | ├── ItemCard.jsx |
| --- |

Group by route/module from the start. This is a structure that supports change and growth. The point is not to have your application outgrow the architecture quickly. If it’s based on components and containers that will happen too fast.

A module based architecture is easy to extend. You can just add modules on top of it without increasing the complexity.

The container/component structure is not wrong but it’s too generic. It doesn’t tell the reader anything about the project besides that it uses React.

### **Create a Common Module**

Components like buttons, inputs and cards are used all over the place. Even if you’re not going with a module based structure it’s good to extract those.You can see what common components you have even if you’re not using Storybook. It helps to avoid duplication. You don’t want everyone on the team to make their own version of a button. Unfortunately, this happens way too often because of badly structured projects.

### 

### **Use Absolute Paths**

Making things easier to change is fundamental for your project structure. Absolute paths mean that you will have to change less if you need to move a component. Also it makes it easier to find out where everything is getting pulled from.

| // 👎 Don't use relative paths import Input from '../../../modules/common/components/Input'  // 👍 Absolute ones don't change import Input from '@modules/common/components/Input' |
| --- |

I use the @ prefix to signal that it’s an internal module but I’ve seen it done with ~ as well.

### **Wrap External Components**

Try not to import too many 3rd party components directly. By creating an adapter around them we can modify the API if we have to. Also, we can change the library in a single place.

This goes for component libraries like Semantic UI and utility components as well. The simplest thing you can do is re-export them from the common module so they’re pulled from the same place.

A component doesn’t need to know what library we’re using for the date picker - only that it exists.

| // 👎 Don't import directly import { Button } from 'semantic-ui-react' import DatePicker from 'react-datepicker'  // 👍 Export the component and use it referencing your internal module import { Button, DatePicker } from '@modules/common/components' |
| --- |

### 

### 

### 

### 

### **Move Components in Folders**

I create a components folder for each module in my React applications. Whenever I need to create a component I create it there first. If it needs extra files like styles or tests, I create its own folder and put them there.

As a general practice it’s good to have an index.js file which exports the React component so you don’t have to change import paths or have repetitive ones like import Form from 'components/UserForm/UserForm'. Still, keep the component file with its name so you don’t get confused when you have multiple ones open.

| *// 👎 Don't keep all component files together* ├── components  ├── Header.jsx  ├── Header.scss  ├── Header.test.jsx  ├── Footer.jsx  ├── Footer.scss  ├── Footer.test.jsx  *// 👍 Move them in their own folder* ├── components  ├── Header  ├── index.js  ├── Header.jsx  ├── Header.scss  ├── Header.test.jsx  ├── Footer  ├── index.js  ├── Footer.jsx  ├── Footer.scss  ├── Footer.test.jsx |
| --- |

## **Performance**

### **Don't Optimize Prematurely**

Before you make any kinds of optimizations, make sure that there is a reason for them. Following e best practice blindly is a waste of effort unless it’s impacting your application in a way.

Yes, it’s important to be aware of certain things but prioritize building readable and maintainable components before performance. Well written code is easier to improve.

When you notice a performance problem in your application - measure and identify the cause of your problem. No point in trying to reduce rerender count if your bundle size is enormous.

Once you know where the performance problems are coming from, fix them in the order of their impact.

### **Watch The Bundle Size**

The amount of JavaScript that has to be sent to the browser is the most important factor of your application’s performance. Your app can be blazing fast but chances are no one will find out about this if they have to load 4MB of JS to load it.

Don’t ship a single JS bundle. Split your application on the route level and even further. Make sure you’re sending the least amount of JS possible.

Load in the background or when the user shows intent that they’ll need another bundle. If a button press is triggering a PDF download, you can delay the download of the PDF library until the button is hovered.

### **Use Code splitting**

Many JavaScript frameworks combine all of their dependencies into a single file, making it simple to incorporate JavaScript into an HTML page. Because all of the JavaScript is in one place, the bundle only requires one link tag, and there are fewer calls required to set up the page.

The overhead of interpreting and executing the code causes the page load to slow down rather than speed up once a bundle reaches a certain size. Each page has its own tipping point, which you should test to find out where it lies. Everything depends on the loaded dependencies.

We can use dynamic import().

| // *👎 Before dynamic import*  import { add } from './math'; console.log(add(10, 15)); |
| --- |

| // *👍 After dynamic import*  import("./math").then(math => {  console.log(math.add(10, 15));}); |
| --- |

### **Code Splitting in React - Using React.lazy() and Suspense**

A component made with React.Lazy() is required for rendering when it is loaded. As a result, you should show some kind of placeholder content, such as a loading indicator, while the lazy component is loading. This is precisely the purpose of React.Suspense.

React.Suspense is a component for wrapping lazy components. A single Suspense component can wrap multiple lazy components at different hierarchy levels.

| import React, { Suspense } from 'react'; const About = React.lazy(()=>import('./About')) const App = () => {  return (  <**Suspense** fallback={<div>WAIT</**div**>}>  <**About**></**About**>  </**Suspense**>  ); }; |
| --- |

### **Code Splitting by Routes**

App routes are a good place to start when it comes to code splitting. Break an application down into chunks per route, then load that chunk when the user navigates that route. Under the hood, webpack creates chunks and serves them to the user on demand.

You can perform route-based code splitting without using an external package by using React.lazy() and React.Suspense. Simply convert all route components in your app to lazy components and wrap them in a Suspense component.

| import React, { Suspense, lazy } from 'react'; import { BrowserRouter as Router, Route, Routes } from 'react-router-dom'; const Home = lazy(() => import('./routes/Home')); const About = lazy(() => import('./routes/About')); const App = () => (  <Router>  <Suspense fallback={<div>Loading...</div>}>  <Routes>  <Route exact path="/" element={<Home/>}/>  <Route exact path="/about" element={<About/>}/>  </Routes>  </Suspense>  </Router> ); |
| --- |

### **Rerenders - Callbacks, Arrays and Objects**

It’s good to try and reduce the amount of unnecessary rerenders that your app makes. Keep this in mind but also note that unnecessary rerenders will rarely have the greatest impact on your app.

The most common advice is to avoid passing callback functions as props. Using one means that a new function will be created each time, triggering a rerender. I haven’t faced any performance problems with callbacks and in fact that’s my go to approach.

If you are experiencing performance problems and the closures are the cause then remove them. But don’t make your code less readable or more verbose unnecessarily.

Passing down arrays or objects directly falls into the same category of problems. They fail the reference check so they will trigger a rerender. If you need to pass a fixed array, extract it as a constant before the component definition to make sure the same instance is passed each time.

## **Testing**

### **Don't Rely on Snapshot Tests**

Ever since I started working with React I’ve had only one situation in which snapshot tests have caught a problem in my components. A call to new Date() without an argument had slipped and it always defaulted to the current date.

Besides this, snapshots have only been a cause for failed builds when a component is changed. The usual workflow is to make a change to the component, see that snapshots are failing, update them and proceed.

Don’t get me wrong, they are a good sanity check but they are not a replacement for good component level tests. I avoid even creating them anymore.

### **Test Correct Rendering**

The main thing that your tests should validate is whether the component works as expected. Make sure that it renders correctly with its default props and with ones passed to it.

Validate that for a given input (props) the function returns the correct result (JSX). Validate that everything you need is on the screen.

### 

### **Validate State & Events**

A stateful component will most likely change as a response of an event. Simulate the events and make sure that the component responds properly to them.

Validate that the handler functions were called and correct arguments were passed. Check if the internal state was properly set.

### **Test Edge Cases**

When you have the basic tests covered, make sure you add some to handle edge cases.

That would mean passing an empty array to make sure you’re not accessing an index without checking. Throw an error in an API call to make sure the component handles it.

### **Write Integration Tests**

Integration tests are meant to validate an entire page or a larger component. It tests whether it works well as an abstraction. They give us the most confidence that the application works as expected.

The components on their own could be working well and their unit tests could be passing. The integration between them could have problems, though.

## **Styling**

For styling we can use many methods based on the requirement of the project. Mentioning here some methods to achieve better code usability, readability and performance in styling elements.

### **Use CSS-in-JS**

“CSS-in-JS” refers to a pattern where CSS is composed using JavaScript instead of being defined in external files.

### 

### 

### 

### **Keep Styled Components Together**

When it comes to CSS-in-JS components it’s normal to have multiple ones in the same file. Ideally we’d like to keep them in the same file as the regular component that uses them.

However, if they become too lengthy, as styles can get, extract them in their own file living next to the component that uses them. I’ve seen this pattern used in open source projects like Spectrum.

### **Tailwind CSS**

Tailwind CSS is a solution to all the troubles traditional CSS imposes on building the user interface. Being a utility-first CSS framework, it makes building the user interface faster and easier.

Tailwind CSS is highly customizable and has all the building blocks necessary to build any design you wish. It is the utility-first framework that allows you to use utility classes to build your applications.

What makes this Tailwind CSS so unique is that it does not impose design specifications. It allows you to combine components to create designs without any opinionated styles to override.

### **SASS/SCSS**

Being an extension of CSS serves as a preprocessor that is compiled into Cascading style sheets. Every front-end developer should learn this first before trying out the tailwind CSS.

SASS gives developers some advanced features for writing CSS such as variables, loops, and nesting. Which allows them to create better-styled components for their applications.

## **Data Fetching**

### **Use a Data Fetching Library**

React doesn’t come with an opinionated way of fetching or updating data from an API. Each team creates their own implementation usually involving a service that holds async functions which communicate with the API.

Going that route means that we need to manage loading states and handle http errors on our own. That leads to verbose code with a lot of boilerplate.

Instead of doing that we should use libraries like [React Query](https://react-query.tanstack.com/) or [SWR](https://swr.vercel.app/). They make communicating with a server a natural part of the component lifecycle in an idiomatic way - a hook.

They come with caching built in and manage loading and error states for us. We just need to handle them. Also, they remove the need to use a state management library to handle that data.