Introduction

You are drawn to UI development for a reason. Maybe you're a visual person and you want to see the impact of your work. Perhaps you want to work on things that are closer to the user - nothing more user-oriented in the product than the UI. Maybe it's a bit of both.

I always got satisfaction knowing that people are interacting with my work daily. They're pressing the buttons I designed, filtering data using my logic, and experiencing bugs that I've written.

Some developers switch to the front-end looking for a fresh challenge. Building the interface of a large application is a solid test for any developer. It could also be just a new project at work. You decided to pick this book to improve your skills and knowledge before you dive in.

Regardless of why we started, we are all crafting UIs with React and we are all facing the same problems. We go through the docs, we take a course and we're still left with the same questions about application structure and best practices.

I had a lot of those when I started working with React in 2016. Back then the technological landscape was different. Most developers were still wrapping their heads around the idea of component-based development. When most people saw JSX for the first time they closed the browser. At least I did.

React's core philosophy, to focus on single elements was groundbreaking. Before that, the unit of work was the whole page. Tools like AngularJS were inspired by the traditional MVC model and aimed to recreate that pattern in the UI. React expanded this and made the unit of work the component. It established that the UI shouldn't be described as a collection of pages but as small elements that pass data between each other. It was also closer to the language than the alternatives. While other frameworks added their own syntax for loops and conditionals, React stuck to plain JavaScript.

Nowadays React powers thousands of applications and has a great community behind it. It's proven to be a reliable tool to build feature-rich UIs with. As the demand for React developers grows, there are still conflicting ideas about how the technology should be put to use.

There are so many amazing courses and books that can teach you the fundamentals. The syntax, state management, passing props, hooks, and all the rest. You can start with zero knowledge and finish with a portfolio of projects to showcase. A simple Google search will reward you with many pages filled with quality content on those topics. So why have I written this book then?

Most available materials will teach you the little pieces. But the content that shows you how to put them together is still scarce. There aren't enough materials on project architecture and software design. React suffers from the lack of a golden path. There are multiple ways to achieve the same outcome and while this gives freedom it can also be confusing. So where do you go once you've covered the fundamentals?

Throughout the years I've had both successful and not-so-successful projects. Some were well-organized others were a nightmare to maintain. Trial, error, and advice from great people helped me form a set of principles. This book is a collection of them.

Every React developer must know a few things about structuring a project, designing components, and common anti-patterns. You will learn a lot of things the hard way when you start building applications. I know I did. I want to save you some time and let you learn them at my expense.

Conversations about architecture and design philosophies can be too abstract and daunting. As developers, we're used to working with specifics so those conversations can be too vague. Does it matter that much how we name a component? Will the number of props make that big of a difference? Don't custom hooks make my code more complex than it should be?

A project is never built and done. It needs to be extended and modified. We can all create but how many of us can maintain and update a creation for

years? We also rarely work alone. Even if we understand our code well, will the person who reads it in a couple of months understand it? Will we understand it after some time?

Tao of React holds all the rules and guidelines that I've followed to build extensible projects that one can work on productively. Each rule specified in the book has proven to be effective in my practice. Some of them may have better alternatives based on the domain, but in all cases, they will have a positive effect on your codebase. Put simply - you won't be wrong if you follow them.

When you're going through the rules, keep in mind that I have formulated those opinions based on my experience. I'm confident in teaching them but I do know that there are things I'm yet to learn. There's more than one way to build software so if you've found alternative approaches to mine, that is great. Either way, I'm sure you'll find inspiration in these pages or something to incorporate in your work.

I wrote this because this is the book I wish I had when I started working with React.

How to Use This Book

Tao of React is split into four chapters - architecture, components, testing, and performance. I think those are the four most important topics for any React application. You need to establish both high and low-level rules. You need to be able to validate that everything works. And you need to know that it works with acceptable speed.

The chapters are just groups of short and self-contained rules. They focus on a specific problem and describe a way to solve it. Most rules have an example of a non-ideal approach so you can compare it with the proposed solution. Each rule can be read in isolation, the book is not meant to be read from cover to cover. You can open it in the middle, read a random rule and it will make sense.

There are also some general ideas to keep in mind that I've added at the end. They provide more context about the importance of software design so if you want to get more insight into the topic, make sure to go through them as well. They too are short and straightforward.

Notes About the Remastered Edition

Tao of React's first edition came out in the spring of 2021 with the promise that it will be the default, up-to-date resource on React project architecture. That's not an easy promise to hold up. Even though React's APIs have mostly stayed the same ever since hooks were introduced, the best practices around it are constantly evolving.

In the last couple of years, developers have pushed back against the misuse of the `useEffect` hook, server-side rendering is becoming even more important, the React team is recommending starting new projects with meta frameworks like Next by default, and server components were finally introduced in a stable environment. All these changes will, without a doubt, affect how we build and structure applications. Moving more and more responsibilities to the server means we will have to rethink state management. Modern meta frameworks' layouting and routing functionality will push us to rethink application structure and data fetching. At the time of this writing, web development is much more mature than it used to be. The front-end industry has not only grown but is also pushing back on over-engineering practices in favor of modern tooling focused on solving problems and delivering products. But at the same time, we will have to approach modern React development carefully because if we do not apply the design principles correctly, modern tools will only replace technical problems with architectural ones.

All this may sound daunting at first, but the rules inside this book are intentionally vague enough to be applicable in any environment, and focused enough to be actionable immediately. In the Remastered edition, we will focus on some of the modern problems in React and analyze how we can take advantage of evergreen software engineering principles to solve them. Reading this book will give you the up-to-date best practices together with the context of how and when to use them.

Table of Contents

1. Architecture

1.1. Create a Common Module

Conversations about architecture can be vague. We can argue about the benefits of one structure over another. We can debate using a state management library. But there's one thing that is guaranteed - we will have logic that will be used all across the project.

The reusable code in React applications consists mostly of components but it's not limited to them. Custom hooks, business logic, constants, and utility functions that are reused need a place to live. So one of the first things that we should do is create a common module that keeps them together and communicates that they're used across the whole application.

Reusable components are usually buttons, input fields, and content containers like cards. But often they include business-specific components as well. An analytics application will most likely have reusable chart components.

Media websites may have a configurable component for text formatting. The same goes for utilities and hooks. Some of them will be generic. For example, you can create one to track element visibility on the screen. That's a behavior that may be needed in many components. Other hooks will be domain-specific - they will deal with data fetching or data manipulation.

Depending on the domain of your application you may store business logic in there as well (more on that in rule 1.5. and 1.11.). Common operations, like calculations and formulas, that are used often should be implemented once, tested, and reused.

You need to plan what you're going to do with those reusable pieces from the start. If we don't create a place for them, they will clutter the project and we will end up with multiple implementations of common functionality.

Create a module that will host all of those components and logic. Then reference them from there. It doesn't have to be anything complex - a folder named common would work fine (check rule 1.7. for an example module structure). Developers can contribute and update it when necessary.

I’m a big advocate of Storybook because it improves the visibility of your reusable components and doubles down as a documentation tool. It makes browsing the common module much easier and gives managers and designers quick access to them. However, keep in mind that such a tool comes with a non-negligible maintenance cost. Each time a component changes you will have to update Storybook as well. Each change to props, each new use case - they need to be reflected there. Otherwise, you won’t be getting advantage of the tool. My advice would be not to use Storybook unless you have the resources to maintain it.

Make sure that your common module doesn’t get too bloated. If it grows too much it will become hard to navigate and manage. To avoid that you can split the functionality into smaller, more specific modules that are meant to be reused.

1.2. Use Absolute Paths

Making things easier to change should be one of your main goals when we're creating our application's structure. Most of us use reference paths when we're importing something but they are not ideal in larger projects. If you need to move files around or reorganize a module you need to modify a lot of import paths.

```
import Input from "../../../modules/common/components/Input";
```

Another problem is that those import paths make it harder to understand where things are coming from. In the example above it's still clear that you're importing something from another module. But what if we want to reference a hook from the same module?

```
import { useAnalytics } from "../../../hooks";
```

This is not necessarily a bad practice but it leaves some ambiguity. Can you tell if this hook is specific to a component, or if it's in the root of the module? Even if the source can be derived out of context or you don't expect to change the folder structure, I would always advocate for clarity.

```
import { Input } from "@modules/common";
import { useAnalytics } from "@modules/instrumentation";
```

With absolute paths, we make sure that change would be easier and everyone would know where something comes from with a glance at the imports section.

For this to work you will need to modify your Webpack configuration so it knows where to look for your files. You can define a prefix like `@modules` and tell Webpack to look in a directory named "modules" for such paths. That's the whole change. The pattern I follow is to prefix them with `@` because that's closer to the regular package naming conventions. I've seen

people use `~` as well to make sure internal imports are easily noticeable. As long as you use absolute paths, use whatever convention makes more sense to you.

Implementing absolute paths is not a complex thing to do but it may differ greatly depending on your setup. The most important thing you need to do is tell your module bundler where it should look for imports starting with your prefix of choice. If you happen to be using Webpack you can look at the resolution aliases in its documentation. The exact syntax may differ depending on the version you're using.

But there are more tools that we need to make aware of our decision to use absolute paths. We need to make sure our IDE or editor understands that as well since or it will mark them as unresolved. Tools like eslint and Jest need to be configured as well.

The setup itself is not complex, it comes down to adding a few lines in JSON configuration files but the syntax depends on the tools you're using.

1.3. Put Components in Folders

Most components are not implemented in a single file. There's one for the actual implementation, one for the tests, the styles, utility functions, and hooks if needed. If we need to split some of the logic into smaller subcomponents that count grows even more. Putting all components with their additional files in the same folder quickly becomes messy. To distinguish what files are used by what component you will have to open the implementation. This takes unnecessary time.

```
├── components
|   ├── Header.jsx
|   ├── Header.scss
|   ├── Header.test.jsx
|   ├── HeaderItem.jsx
|   ├── HeaderLogo.jsx
|   ├── Footer.jsx
|   ├── Footer.scss
|   ├── Footer.test.jsx
|   ├── hooks.js
|   ├── utils.js
|   ├── constants.js
```

A better way to structure components is by putting them in their own nested folders. Each component that represents a larger logical unit should be moved in its own folder.

Sometimes you need to split some parts of a larger component Smaller subcomponents like the items in a list should be in a nested `components` folder. They're considered private and shouldn't be imported outside the main component that uses them.

Developers that use this structure sometimes name the main component in the folder as `index` to avoid repetitive code paths `/Component/Component`. I avoid that because it leads to confusion when you have multiple components open in your IDE.

```
├── components
|   ├── Header
|   |   ├── core
|   |   |   ├── constants.js
|   |   ├── components
|   |   |   ├── HeaderItem.jsx
|   |   |   ├── HeaderLogo.jsx
|   |   ├── Header.jsx
|   |   ├── Header.scss
|   |   ├── Header.test.jsx
|   |   ├── hooks.js
|   |   ├── utils.js
|   |   ├── index.js
|   ├── Footer
|   |   ├── Footer.jsx
|   |   ├── Footer.scss
|   |   ├── Footer.test.jsx
|   |   ├── hooks.js
|   |   ├── utils.js
|   |   ├── index.js
```

But the problem with repetitive paths remains and they don’t look nice. To alleviate that we can still create an "index" file that will just export the main component in the folder. Keep all other files that are only used by that component inside its folder. We should put everything related to them there - tests, styles, hooks, and even utility functions. That way a component becomes a small self-contained module on its own.

1.4. Group Components by Route/Module

The conventional way to structure your application is in containers and components (also known as smart/dumb). This architecture is a safe choice that you can apply for most applications and it will work well. I like to compare it to the MVC architecture for back-end applications. It's generic enough to work in most contexts. It splits components into two logical groups.

Containers are usually the pages or larger logical sections of the application. Components are the smaller elements that only visualize markup, handle minimal logic, and don't keep state.

```
├── containers
|   ├── Dashboard.jsx
|   ├── Details.jsx
├── components
|   ├── Table.jsx
|   ├── Form.jsx
|   ├── Button.jsx
|   ├── Input.jsx
|   ├── Sidebar.jsx
|   ├── ItemCard.jsx
```

I'm not a fan of this structure for multiple reasons. First, any non-trivial application will be hard to navigate with this composition. Most projects quickly outgrow it with the number of components they require. Your "containers" folder will keep a small number of files but the "components" folder will become a mile long in no time.

Second, it encourages bad component design. Developers tend to give too many responsibilities to containers. When things start doing too much they become large and hard to manage. Introducing bugs becomes easier and testing becomes harder (think about how many things you will have to mock to test a specific behavior).

It doesn't tell anything about the application. When you're looking at a project with this structure can you tell what it does at a glance? An application's

structure should tell you its story and this doesn't say anything besides that it's a React app. Imagine being a new developer that will be building a feature in a large project designed this way. How will she know where to start looking?

Instead, group the application into smaller self-contained modules.

```
├── modules
|   ├── common
|   |   ├── components
|   |   |   ├── Button.jsx
|   |   |   ├── Input.jsx
|   ├── dashboard
|   |   ├── components
|   |   |   ├── Table.jsx
|   |   |   ├── Sidebar.jsx
|   ├── details
|   |   ├── components
|   |   |   ├── Form.jsx
|   |   |   ├── ItemCard.jsx
```

Each module should represent a logical part of your application. Initially, you can start by splitting your application based on your top-level routes. This sets clear boundaries between the different parts of your application. It makes it clear where everything is used and improves navigation.

The thing I like the most is you can find out what the application does by opening its modules. Then if you need to work on the dashboard, for example, it's much easier to locate the files that are relevant to your task.

As an alternative, you can use a hybrid approach that puts the reusable components, hooks and functionality on the top level of the application, and groups the rest based on the application's page structure. This is a compromise for those teams who find the presence of the `common` module confusing or want to achieve additional separation between reusable functionality and logic which is specific to parts of the application. With this structure, again, each folder inside `modules` represents a separate page.

```
├── components
├── hooks
├── utils
├── modules
|   ├── dashboard
|   |   ├── components
|   |   ├── hooks
|   |   ├── utils
|   ├── details
|   |   ├── components
|   |   ├── hooks
|   |   ├── utils
```

With this structure, everything which is used in two or more places sits on the top level of the project, leaving the `modules` directory to represent specific parts of the application. To some engineers this is preferable, because the `modules` becomes a direct mirror of the app structure. Alternatively, I've noticed teams use `screens` or `pages` instead of `modules` with this approach because it's closer to what the contents represent. I'd argue against using `pages` because it's already an established naming convention in a popular React meta framework, so I'd advise you to use `screens` if you settle for this structure.

A last option for teams who feel strongly against this way of structuring applications despite its benefits, is to use a top-level split but nesting the names of the pages inside of it.

```
├── components
|   ├── dashboard
|   |   ├── ...
|   ├── details
|   |   ├── ...
├── hooks
|   ├── dashboard
|   |   ├── ...
|   ├── details
|   |   ├── ...
```

```
├── utils
|   ├── ...
```

This is a compromise that I would use only as a last resort, but it's still favorable to a split in which everything lives in the same place. It still conveys context about the application's structure, but it requires more jumping between folders to find related functionality. You can use it as a starting point for restructuring an application in a messy state by first grouping things on the top-level and then extracting them to modules.

1.5. Manage Dependencies Between Modules

The module structure raises a very important question - how to manage dependencies between different modules. In this context, we use the word dependency to describe one module importing something from another.

In an ideal scenario, we have some main app configuration at the top level that puts all the different modules together. They import components and logic only from the “common” module and don’t have a connection between them. No module knows that another one exists.

In practice, we often need the same thing that was developed for another module. The module that deals with a details page may need a graph that is created for the dashboard. Is it that bad to import it directly from there? The best thing to do in such a situation is to move that graph in the “common” module and import it from there. If it’s needed in many places then it’s global, right? The more practical advice is to first duplicate the logic in both modules by making a copy of the component you need. This means building two graphs - one for the dashboard and another for the details page.

This goes against the popular advice to avoid duplication at all costs. I strongly believe that a little duplication is good for the codebase because it lets it evolve. Often different parts of an application need similar components but they start diverging with time. A graph for the details page may need different settings than one for the dashboard.

If we make an abstraction early we may end up with a component that is too configurable. Its implementation will require too much conditional logic and its complexity will grow. Next time you need a similar component you will be tempted to add a little more configuration and checks. After some time you end up with a hard to configure and maintain component.

Managing some duplication is easier than managing the wrong abstraction. Don’t be afraid of copy-pasting a couple of times so the idea can crystallize. However, keep in mind that duplication is costly. When you find yourself copying the same functionality three or more times - it’s time to do something

about it. You may have to create a new component in the “common” module or a reusable function. This is called the “Rule of Three” - duplicate, then on the third time you need the same piece of code, make an abstraction for it.

But not all logic fits into the “common” module. In larger projects, it may become too big - keeping together generic UI elements and domain-specific logic. In those cases, you may want to extract some of the domain logic in another module. For example, if your application has a lot of forms you can build a “forms” module that groups the different components, hooks, styles, and validation logic related to them.

One last question that remains is how to manage this duplication across a large application that’s worked on by multiple people. It’s easy to know what is being copied if you’re the only developer but that’s rarely the case. In a large team you may end up with a lot of duplication and there isn’t a technical solution for that. It depends on good communication between the developers, sharing information, and having a solid code review process.

1.6. Wrap External Components

The Node ecosystem is notorious for its usage of dependencies even for trivial functionality. Yet, this is unavoidable - we all know that building everything from scratch is impossible. That's why we build on top of other people's work. We pull third-party functionality and components to solve common problems.

Referencing an external component means that all internal ones will be bound to it. This is not necessarily bad. But it means that we couple our codebase to the library's API and specifics.

```
import { Button } from "semantic-ui-react";
import DatePicker from "react-datepicker";
```

Our vision for implementing a date picker could be different from the author's. We may want to manage the different formats with a single prop while the author has made the API more verbose and configurable.

We can accept the library's opinion and just pass the component the data in the format that it accepts. But this means that our application's logic and data management will be influenced by a third-party component. What if that library changes its API in a future version, in how many places will we have to make modifications?

We don't want external factors to shape the way we think about our implementation. We want to have a buffer between the external components that we use - call it an adapter or a wrapper.

We can create an abstraction on top of the external component that provides the desired API. It can then format the data and pass it to the external component.

```
import {
  Button,
  DatePicker,
} from "@modules/common/components";
```

This is a good way to set default values and hide complexity. You can alter the markup, style the component and make it indistinguishable from the other ones. This is the only place in your codebase in which the external library will be referenced.

We mentioned that we want to make future changes easy. This approach reduces the impact that a potential change will have in the future. If we decide to use a different library we will only need to make the change in the wrapper/adapter. Even if we're happy with the library's API and we don't want to modify it, it's still better to wrap it. It gives us control over it from the start and we can reference it like an internal component which is good for consistency.

1.7. How to Design a Module

A module is a group of files that together represent a logical unit of your application. If you're building an admin panel one module could be the main dashboard that shows statistics and charts. An e-commerce application will probably have a module for the product page and one for the home page.

If you're not sure how to start splitting your application into modules a good starting point is looking at the top-level routes of the application. Once you decide the boundaries between them you are faced with the challenge of structuring them.

There shouldn't be a predefined structure that each module should follow to the letter. You don't need to fit all modules in the same pattern but apply some limitations in which they can grow around. If a module contains only components, there's no need to create extra empty folders for consistency's sake. Shape them around the problems that they're solving.

Most times I end up with something like this:

```
├── modules
|   ├── dashboard
|   |   ├── components
|   |   ├── utils
|   |   ├── hooks
|   |   ├── api
|   |   ├── index.js
```

The components directory will probably be the largest one - it hosts all our components and their sub-components (the recommended component folder structure is described in another rule).

The utils folder will contain small reusable, generic functions that are not specific to the business or React. Such a function validates an email, for example.

The hooks folder will keep the custom hooks we build. You may not be used to seeing custom hooks everywhere but I recommend extracting as much logic as you can in them. They are a good abstraction mechanism for any kind of functionality - both generic and business-specific.

The api folder holds business logic that is specific to this module - functions, and constants. This is not the same as the utility folder. The api one should only deal with logic that is specific to the business or the domain like calculations or data manipulation logic.

The index.js file is the main entry point of the module. It holds the main module component - the one that defines the page structure, routes, and subroutes. Depending on your application's structure you may need more or fewer folders but this is a good starting point.

1.8. Project Structure Should Tell a Story

An application's structure should tell a story at a glance. Developers derive a lot of knowledge about a project based on the way the files are organized. When they open one for the first time and they see a generic folder structure that they've seen ten times before, it won't give them much insight.

An application's contents should provide information about the domain that it's serving. A glance through the modules should be enough to tell you the high-level parts of this application. It should tell you if it's a dashboard, a website, or an e-commerce app.

Opening a project to see folders named after libraries doesn't help at all. The knowledge about them could be gained by looking at the dependencies or examining the code. They should be just an implementation detail.

When I say this I'm often asked where do I put Redux (or another state management library) specific logic. Again, it depends on the project and the domain. I may put it in the api folder since it's more related to the business logic. I may create custom hooks to access and modify the data. In all cases, I won't let it be the most important aspect of my project.

1.9. Keep Things Close to Where They're Used

A pattern I see far too often is to put too many things at the root level. Utility functions, configuration files, and business logic can often be found there even if it's only used in specific areas of the application. To me this has always been confusing because I can't know where each of those things belong or where they're used.

If something is not meant to be accessible globally it shouldn't live at the root level because it sends the wrong message. We shouldn't put things there or even at the module level. When we need to create a utility function we should keep it close to where it's used.

First, put the logic inside the component file. Then, if it gets too long and you want to move some of it in a separate file - put it in the component folder. Then follow the rules from the "Manage duplication between modules" rule and move it up.

1.10. Don’t Put Business Logic in Utility Functions

Utility functions should be small tools that you can grab and use in multiple places and occasions. You can use a hammer to put a nail on the wall but it’s also used to put wood joints together. You should be able to lift a utility function and use it in a different module for similar purposes.

We want to extract some of the logic from our components so they don’t become too big or to avoid duplication. But a mistake we make too often is to bind those utility functions too tightly to our business logic.

```
function sortArticles(articles) {
  // ...
}
```

That makes them too specific and impossible to use outside of their current application. In many projects, utility files are just a collection of “stuff” - someone wasn’t sure where to put a function or a file so they jam it in the folder containing the utilities.

Instead, create generic functions that take more parameters if necessary but can be used in multiple places.

```
function sortCollectionByAttribute(collection, attribute) {
  // ...
}
```

If you’d rather use a domain-specific API you can still achieve that through a generic function. Create a business-specific function in the “api” folder that calls the generic one underneath.

1.11. Separate Business Logic From UI

React components are naturally coupled to the domain logic they represent. It's natural for a component to own it since it controls the behavior. That's fine when it comes to UI responsibilities and any attempts to change it would bring too much unnecessary complexity.

But often we need to go through a collection of entities and do some operations on them. We need to make calls to external APIs and validate user input according to business rules. This kind of logic is better to be out of the component's body. Most of those scenarios would fit well as custom hooks. This way you can still stay close to the idiomatic way of writing React and extract that logic.

Whenever possible I prefer to express the logic through a custom hook. It can become a natural part of the component's lifecycle this way. When I'm making an API call or I want to track an element's visibility with the `IntersectionObserver`, a hook fits best.

Sometimes I use it just to make another hook's API more polished. I don't want my components to be familiar with the structure of an external API's response. So I pass to them only the fields that they need in the format that they need them.

```
function useSalesAnalytics(date) {
  const { loading, data, refetch } = useQuery(GET_SALES, {
    variables: {
      date,
    },
  });

  if (!data) {
    return { loading, sales: { total: 0 }, refetch };
  }

  return {
    loading,
```

```
    refetch,
    sales: aggregate(data),
  };
}
```

What if the logic can't be written as a custom hook for some reason? If it's up to me I would still extract it. I usually create a folder called api and put functions and constants there to signal that they are domain-specific.

Aim to have components clean from domain-specific logic but don't take it as an absolute. When you have some trivial operation that doesn't require deep domain knowledge it may be okay to inline it.

1.12. Modules Should Own Their Routes

Most single page applications have at least a few routes and often nested ones as well. A common pattern is to declare all of them in a single main file in the root. I don't like that approach because that file can become too large. Size alone is not a problem but it means that your application's modules are all leaking their details.

This transfers ownership of the application's routing structure to the global level. Every developer must make changes in the same place no matter what part of the application they're working on and that creates opportunities for errors to sneak in.

A better way to structure this is to keep only the main route and file for each module on the global level. The nested routes and detailed structure of the page should be owned by the corresponding module. A change in the layout of a page should only affect the module it lives in. This way we split complexity in multiple places.

The big outlier for the rules in this book are small applications and prototypes. In small SPAs I'd put all routes in the same place. But for any non-trivial or larger project, I'd favor splitting them across the different modules.

1.13. Avoid Single Global Contexts

A common anti-pattern was the prop drilling problem in which you had to pass a prop through multiple components so it can reach its destination. Thankfully, we can use the Context API whenever we want to share data in different horizontal or vertical parts of the React tree to avoid passing unnecessary props.

It's easy to put all the global data we need in the same Context but that should be avoided as well. It's better to separate the data logically and split it across different contexts. That allows components to pull only the data they need.

We can have a separate context for the theme, another for user settings, one for the modal window and so on.

Following the same principle, there's no need to wrap the whole application in a context provider if it's used in a single module. If the other sections of the application don't need to know about some data there's no need to share it. For example, if we have an analytics dashboard and want to make some data accessible for all components in that module we can only wrap it in a Provider, not the whole application.

1.14. Shared Libraries and Design Systems

A common practice in projects with multiple teams is to extract reusable components and functions in a shared library. It's then distributed as an NPM module and developers can pull it in their codebase to save some time.

I think this is a great practice and helps you not to reinvent the wheel but you need to keep some things in mind. Such a library needs to be stable. Therefore if you're still in the active development phase it's better to wait a bit, even at the cost of some duplication, and let things settle down.

Building an application and a component library at the same time is hard. Those are two different initiatives that require resources and thought. Facing both challenges at the same time is a recipe for disaster.

Extracting a library is good on paper but requires a lot of maintenance work. The goal of such a tool is to reduce the amount of work so it's better not to rush it.

The same goes for design systems. They are every UI developer's dream project. Tired of all the off-the-shelf tools that we've been using, we wish we get the chance to build one of our own, designed for our needs.

A design system is more than a component library. It holds the rules and constraints that your UIs will be based on - fonts and font sizes, colours, spacing, variants, themes. Components require a much deeper level of thinking about API design since more people will be using them.

Also, they are not purely an engineering effort. To build a good design system you need to work with the design and product teams. Building one is a great challenge. Integrating it into your work is an even greater one. The companies that have such initiatives have entire teams focused on them - building, maintaining and fixing. A couple of people spending half their time on a design system don't have good chances of producing a great tool.

Small teams would be better off putting together a design guide and using a collection of common components. Underestimating the effort to build one will only give you trouble. If your team doesn't have the resources to let someone dedicate their time to it, use a tool like Tailwind instead.

1.15. Run the Application with a Single Command

Your application will have certain pre-requisites in order to run and function as expected. Outside of dependencies, you will probably need to set up configuration and go through a series of steps to prepare the project for your local environment. To ensure that your colleagues have a good development experience, the amount of steps necessary to start should be as low as possible.

In an idea scenario your application should start with a single command and you should have any other pre-requisites documented in the project's README. Joining a new team only to struggle with running its application for a few hours is a very unpleasant experience. Try to make it so running `npm install` and `npm start` or `npm run dev` makes the application start.

Of course, you might need certain environment variables. Ensure that those who can be hardcoded, are already added and the rest produce readable error messages when they're missing. If your app requires an API key to function, throw an error if its not set, instead of starting only to face failed HTTP requests.

Some applications depend on a `.env` or `.env.local` file to read the values of the environment variables. To avoid writing a complex command each time you're running the app or forgetting to refresh them, you could include it in the start command itself. This way when you run `npm run dev` underneath it could source the new environment variables like so `source .env.local && npm start`.

Alternatively, if your project requires a more complex set up, you should consider using Docker to ensure stability across different people's development environments. After all, we're not building in isolation and the larger the team the greater the impact of the environment's specifics. It's easy for a single developer to keep track of everything that's going on on their machine, but it takes a lot of effort as the number of people grows.

Docker can be daunting at first but you don't need to become proficient with it to use it. You can find a good preconfigured Dockerfile that works with React applications with a simple Google search, and add your specifics on top.

Also, it makes it easier to run multiple applications at once. If you're writing the front-end for an application made up of multiple services you can run them all with a few commands. A few years ago I was working on an app for a system that was made of a few Spring Boot microservices. I had little knowledge of how to work with them but being able to spin them up locally and configure them made my life a lot easier.

Something you need to be careful when using Dockerfiles is server-side rendering. If you're making a server-side request to another container it may not resolve. Due to how Docker's networking works if you need to send a request to another container you need to reference it by its name. Specifying localhost and a port won't work, so you will need to configure it with an environment variable.

1.17. Pin Dependency Versions

A bug introduced in a third-party dependency can be a great roadblock for development. It can halt a release or break an application in production. It's not rare to see developers running the same project with differences in the dependency versions that they have. Not managing the dependency version can lead to indeterministic environments that can cause hard to reproduce problems.

Instead of debugging dependency problems by rerunning npm install, pin the specific versions of the libraries you want to use, even the minor ones. Thirdparty modules can have bugs slip past the maintainers. We're all people, we make mistakes.

But to make sure your app will always produce a working build, pin the specific versions of the dependencies that you're using. Keep track of outdated modules with npm update and make sure they are getting updated intentionally.

1.18. The Styling Dilemma

One of the first decisions you have to make is what styling approach to chose. You have to pick between CSS (modules), SCSS, CSS-in-JS libraries, and utility libraries like Tailwind. There isn't an inherently right or wrong choice. Your decision should be based on the team's experience, preferences, and philosophies. All of those technologies are proven and they can support you to build applications of any scale.

There are countless opinionated articles about the benefits and problems of each technology. But the bottom line is that none of them are perfect, you need to make a trade-off and decide what problems you want to deal with daily.

But no matter what styling tool you pick, make sure that it doesn't add unnecessary complexity to your application. Each technology brings some overhead, but we don't want to add another source of complexity to an already complex application. What follows is my opinionated view regarding React application styling and the approach that I use.

Nowadays I avoid working with CSS and SCSS because I don't want to think about classes and naming conventions. I never was a fan of BEM because I felt it was too complex at scale. It adds another convention that you need to follow when naming classes, writing styles, and reviewing other people's code.

I would advise you to consider using a CSS-in-JS library or Tailwind. They remove the need to follow a convention or create complex class hierarchies. They help you keep the focus on building components rather than managing styles.

CSS-in-JS libraries like Emotion or Styled Components remove classes entirely. They allow you to describe styles in terms of wrapper components that are a regular part of your application. To me, this is the natural way to work with React - everything is a part of the component hierarchy. Styles are managed via props instead of conditional statements and classes. It allows us

to be even more descriptive. But let’s focus on the problems since they are uncovered only in practice.

First, CSS-in-JS components can take a lot of space. Usually, they are kept in the same file as the component that uses them so you can imagine that they can get lengthy. To avoid having to navigate giant files, I like to put the CSS30 in-JS components in a separate folder called "styles" or "elements". They can then be imported as regular components.

Another topic of discussion is naming them. Most developers like to prefix them with a word like “Styled” so they can be easily distinguished from regular components. I find this unnecessary. Conventions are useful when they bring some benefit but I don’t see how making a difference between regular and CSS-in-JS components helps. A CSS-in-JS component is no different than a regular one that prints its children. Some of them even take props to alter their appearance.

Then there is Tailwind. A relatively new utility class library. I started paying more attention to it in 2021 and the developer experience is surprisingly pleasant. It’s a complete design system that allows you to describe everything from spacing and colors to shadows and fonts in terms of a predefined scale. Implementing all of it on your own would take you months. What I like about Tailwind is that it makes the UI look good and symmetrical by default. But it’s not without its problems too.

It’s not that hard to learn but people need to get used to the mental model. You still need to put together all the utility classes and they can get hard to read when they’re many. That’s the tool’s biggest problem - when you open a component and see long class names made up of abbreviations it’s hard to wrap your head around how each element fits in the picture. That’s the problem with utility classes - they explain how the component looks but not what it is.

To avoid confusion we can extract components with long class names. Even if they don’t do anything besides rendering children. This way we can provide more context about specific elements and their importance. It’s better to see a

named component than a long string. When we need to conditionally add or remove any of them we can hide that complexity in the component too. The idea is to spread the complexity in many small components, not keep it all together when we need to manage complex styling.

When I'm starting a new project I usually pick CSS-in-JS or Tailwind depending on the team. If you're not familiar with them I'd encourage you to give them a try with an open mind, you might be surprised. If their ideas don't resonate with you, then you can go the conventional way with CSS/SCSS - a proven approach.

1.19. Avoid Hardcoding Links

Whether it's the links of your application's pages, the path to redirect after authentication, or the URL of a REST API that we need to fetch data from, we need to work with addresses. Managing them is one of the most common things that we do as UI developers. But too often we leave them hardcoded in components and hooks which makes them harder to maintain and modify.

We should be treating them like any other constant of high importance, keep them in a central place, and reference them from there. We can keep the main URLs for REST APIs as environment variables that can be configured. We can then extract the paths and reference them as constants so we won't have to work with hard-coded strings.

But URLs have a nested structure that will require a lot of constants to represent. They can get lengthy, naming them would be a challenge and it doesn't solve the problem with string interpolation. We will probably need to construct dynamic links to access specific data. To solve this we can build a small utility object that represents the link structure.

```
const links = {
  about: "/about",
  contact: "/contact-us",
  articles: {
    list: "/articles",
    single: (id) => `/articles/${id}`,
  },
};
const endpoints = {
  articles: {
    list: `${SERVICE_URL}/articles`,
    find: (id) => `${SERVICE_URL}/articles/${id}`,
  },
};
```

Our goals when designing a codebase are to avoid ambiguity, improve readability and make things easy to change. With those helper objects, we have a more descriptive way to use links and we limit possible changes to a single place in the codebase. We can use them as follows:

```
// Usage in anchor tags
<a href={links.articles}></a>
<a href={links.articles.single(id)}></a>

// Usage when fetching data
fetch(endpoints.articles.find(id)).then(...)
```

Avoid putting checks for empty strings in those functions, though. This shouldn't be their responsibility. The check that an id, for example, adheres to the required format should be done beforehand. Don't put complexity in them, they should act as more descriptive placeholders for the URLs.

I wouldn't even test those functions since the only thing they do is string interpolation. Yes, not being able to navigate because of a bug is a big problem but it would be better caught in an integration test instead.

In terms of ergonomics, this helps reduce line length. Longer lines stretch the imagination of tools like Prettier and they're not that pleasant to look at.

1.20. Add Your Own Touch

The architectural rules in this chapter will give you a good foundation to build upon. I'm confident that if you stick to them you will have a productive codebase that can grow and be extended.

But each domain has its specificities. You may reach a point in which the structure, for example, is limiting to you. If there isn't an obvious place to implement the logic you've probably reached the limits of your current structure.

Don't stress it, this is completely normal. Architecture isn't immutable. Add your own touch and shape the application in a way that will support its future growth. Add more files, change the directory layout or add naming conventions if you're confident they will help. Keep in mind that the changes you make now will also need to change in the future.

I can't give you specific advice here because all domains are different. But if you find yourself lost, see if you can reach out to other developers working in the same field. In my experience, people love to help. Companies in the media industry, for example, have reached pretty similar conclusions regarding how they use React. If it works for another company, chances are it will work for yours as well.

1.21. Use TypeScript

I have kept this book focused on React without stretching out into technological choices and opinions about tooling. But one topic I can no longer ignore is the importance of using TypeScript in a professional environment. In 2016 many teams were reluctant to introduce it because its support needed to be improved, the ecosystem and typings were largely missing, and it added a time cost with an unclear return on investment.

Nowadays, TypeScript is stable, and it's proven more productive to develop with than pure JavaScript (at least in a team environment). Its biggest plus is that more context about your domain and application gets stored in the code itself. You won't have to document as much or rely on colleagues reminding you of the expected values of an object or function.

Put simply, you will have fewer things to remember.

If your team is inexperienced with TypeScript or hasn't had prior encounters with static types, some initial learning curve will be required. You will spend time scratching your head over TS errors that don't make sense at a glance. But the more time you spend using types, the more you recognize how the compiler is protecting you from common errors and edge cases. You will notice how easy it is to come back to an older codebase if it has types.

And most importantly, you will see a correlation between static types and reduced bug count. I won't go as far as claim that they will reduce the errors you experience, but if used properly, TypeScript will make you think deeper about how data flows through your application. React has excellent support for it, and its testing, styling, and utility tools do too.

To avoid turning this into a TypeScript manual, I will only give a single piece of advice to people new to it - avoid using the `any` and `as` keywords. When TS was first introduced, these keywords were added as a measure to allow teams to gradually adopt it in a JavaScript codebase. They are a way to turn off the compiler's safeguards and tell it that you know better. And even though they're useful in cases when types are not present, nowadays, this is rarely a

problem. You should follow TS's warnings and errors and favor adding checks when necessary instead of silencing the error with `any` or `as`.

Another point that's worth stressing is that the more your TypeScript code looks like JavaScript, the better. This can be very counter-intuitive at first. After all, we want to use static types and get the benefits of validating them, why would we want to keep our work looking like the inferior language?

The TypeScript compiler can infer a lot of the types for you, making it unnecessary to write down type annotations everywhere. For example, if the returned value of a function is of type string, you don't have to explicitly specify it on the function's definition. TS will infer that this function will return a string and automatically assign the same type to the variable in which you're storing the function's result. The more work the compiler can do for us, the better. On one hand we reduce the unnecessary type noise in our code - there are even eslint rules that remind you to delete unneeded annotations. But most importantly, we make sure that we're not cheating the compiler or ourselves with a type annotation somewhere.

1.22. Create Layout Components

Most UIs try to maintain a sense of consistency across their pages. Not only throughout their colors, spacing, and fonts, but through the position of elements on the screen. Perhaps the content will be centered in a container with a certain width, or there will be an ever-present sidebar with filters that ease the user's browsing.

To make sure this structural consistency is kept in the application, it's best to create layout components that enforce it. Each one should be responsible of creating the page's skeleton, add context providers, set up SEO tags, and render its `children` in the proper place on the screen.

A mistake that I keep seeing is teams trying to put too much logic into a single layout component and have it change its structure based on configuration. For example, they will pass a `type` prop that specifies whether it should render a centered or two-column layout. In reality, you'd be better off with multiple different layouts that handle the specifics of each different page structure.

There will undoubtedly be some duplication between the layout components, but don't rush to create additional abstractions. The repetitive code in these components will rarely cause problems, because they are not changed often. Duplicated code can be detrimental when you often have to make changes in multiple places, but layouts are usually set up once and left untouched for long periods of time.

In an ideal scenario, you should be able to wrap your page in a layout and get all the structural, style-related and configurational aspects that the page needs. Follow the rules we defined about modules when you're wondering where to put them. If a single page needs a specific layout, keep it in its folder. If more than one does, then move it to the `shared` module or at the root level of your application (depending on what structure you're using).

1.23. Don't Wrap Context Providers

The number of contexts each page has to be wrapped with can get out of hand, especially for large applications. This will quickly make your page or layout components, depending on where you do the context setup, get the shape of a Christmas tree. The markup will be heavily indented and if any of the providers require a more complex setup, the logic could clutter the component.

Because of that, some teams are favoring an `ApplicationContextProvider` that wraps all the context setup and leaves the page or layout with a single provider, removing all the indentation and complexity. While I'm perfectly fine with this approach, I'd rather have the verbosity of all the contexts inlined instead of abstracted.

I'm stingy with my abstractions, and I don't think that wrapping the contexts gives enough value to warrant creating one.

Managing complexity is hard when you have a frequently-changing environment, but your context providers are most often set up once and forgotten. You may add another one, every now and then, but you will almost never interact with your context setup, once it's in a working condition. Because of that I'm not too worried about them going out of hand, and I'd rather have the additional clarity, even at the price of verbosity.

Perhaps the best advice would be to put them high enough in your component tree so you don't have to interact with them frequently.

1.24. Consider Client-Side and Server-Side Rendering

In a book full of concrete rules, this one will be more subjective. There is no concrete answer about when you should use client-side or server-side rendering, but there are directions we can follow to make a decision that fits our needs. Before we dive into the specifics, I need to point out that there is no favorite between the two - both approaches are applicable depending on the circumstances.

To make the correct choice, you need to analyze the product you're building and understand the business behind it. Without knowledge of the domain and the problems you're solving with this app, you won't be able to make an informed choice about your rendering approach.

Here are some pointers on when client-side rendering would be a better option.

1. Highly dynamic content - if you're building an interactive application like a diagramming tool or a chat, you will do most of your work in the browser either way. Rendering the initial HTML on the server would bring little benefit at the cost of higher complexity.

2. No need for SEO - if your application is not open to the public and is not meant to be crawled by search engines, then client-side rendering would be a suitable approach. This is the case for most dashboard applications and anything behind a login screen. A back-office UI built for a team in your company doesn't need to be perfectly optimized.

3. Faster initial response - if putting something in front of the user quickly is higher on your priority list, then client-side rendering can help show that something is happening on the page. Delivering the initial HTML is a lightning-fast operation. If you keep your assets in a CDN, loading the JS bundle and starting to fetch data won't take too

much time. However, this means that your users will see an incomplete UI while the application and its data are loading. The time to interactive (TTI) will be higher.

4. Simpler development - rendering on the client means you only have one environment to consider. Your application will only ever run in the browser, so you will have fewer corner cases and compatibility issues. All your data will be fetched through API calls, and you won't have to consider hydration and managing a server.

However, server-side rendering can solve another set of problems.

1. SEO-friendly - by rendering your initial HTML on the server, it will contain all the content in the response to the browser. This will allow it to be easily crawled by search engines, and you will be able to provide the necessary SEO tags to help your site rank higher. Also, if your application's pages are frequently shared on social media, you'd need control over how they look there. Setting all the `og` tags on the server will help you do that.

2. Initial page load - the time it will take to display something in front of the user will be higher than CSR - that's a fact. But when the page loads, it will display a complete page with all the content the user has requested in a fully interactive manner. This can be a plus if your user base has a bad network connection or slower devices.

3. Presentational sites with widgets - SSR allows you to take the best of both worlds - you will be able to deliver content to your users with the initial load, and you will have the ability to have interactive widgets working on the page. Many media companies are not building SPAs (single-page applications), but they still use React and SSR, so they can add interactions to the pages when necessary.

4. Better UX - Overall, showing a complete page when it's initially loaded provides a much better user experience. No application wouldn't benefit from that. The price we pay is additional complexity

> to deal with, so we must consider if dealing with that is worth it based on the business's problems.

SSR has many benefits, and I'd rather have it for most applications. Even GitHub, an app you can only access after you've logged in, is utilizing it. Open a page and consider what the user experience would be if you had to see five loading spinners on the page while your data was loading in the browser. But the fact is that a lot of the software built with React will do just as fine with simple client-side rendering. A startup fighting for survival with its SaaS product will do better with a new feature rather than a better initial loading experience. Your users can live with a couple of spinners if you solve their problems well.

The bottom line is that I don't have a clear answer for you. You must look into your business needs and make an informed choice. Your domain should dictate your architecture. Much like the structure of your application should be based on your product's pages, your rendering strategy should be picked for its needs.

1.25. Should You Use a Framework

At the time of this writing, meta frameworks like Next are gaining more traction and becoming the de facto way of using React for many developers. By solving the problems of SSR, data fetching, and routing, they reduce the number of technical problems we have to deal with so we can focus on the domain logic instead.

Like any other technology, I would only use a framework if it's solving a problem that I'm expecting to have. Any application requiring SSR would benefit from a battle-tested solution like Next's. Rolling it out on your own is a complicated and time-intensive endeavor that will often leave you with a half-baked solution. Trust me, I've done it, and it's not pleasant. Using mature technology will give you access to a wider community and libraries that will speed up your development.

It's important to note that using a framework increases the level of abstraction that you will be working on. The technology, no matter how unopinionated it's trying to be, will make some decisions for you, and dealing with their consequences can be problematic because you don't have fine-grained control over the internals. Having your own setup might be easier if you have specific needs related to the build process or the routing. But most modern React applications would benefit from having an off-the-shelf solution to some of their problems.

I bootstrap most of the applications I'm building with Next. Even if I don't need their features at the time being, being able to take advantage of SSR, routing, or a library that's specific to Next would save me a lot of time and effort. I'm not affiliated with Next, but they work closely with the React core team. Knowing that you're using a tool whose vision is aligned with the underlying technology is important for my decision-making process. I have yet to work with any other meta frameworks, so I will abstain from commenting on their capabilities.

There are three cases, however, in which I wouldn't use a framework - if I'm building something that would never need SSR if I'm building something very simple or too complex. Not having to render on the server takes away a lot of the benefits of the meta frameworks. A proof of concept or a prototype would never need anything too complex to warrant running them. And an overly complicated application would require granular control over the internals, which you can't trust to a third party. For any other case, I'd be more than happy to use one.

1.26. Abstract Framework Details

Turning to a framework to resolve common problems is something we've been doing for quite some time. It's the productive thing to do. But I've found that it's even more important to continue following good design principles, or you will just trade one set of problems for others.

The question that you're probably asking yourself as you read the framework-related rules is whether using a tool like Next would affect the rules about application structure this chapter started with. Next's router, for example, is file based and its layouting feature is too based on file location.

This is bound to have significant impact on how we build applications.

But I believe that by following evergreen development principles, we can design our apps in such a way that they are (mostly) oblivious of the framework or environment they run in. We can apply the same practices described in this book to confine the frameworks in a way that allows us to get the most out of them, but still have a productive domain-focused application structure.

One of the most important principles in software design is the one of co-location. Ideally, you want things that work together to live together. But Next's router is file-based and this immediatelly throws a wedge in the previous idea.

```
├── app
|  |  ├── page.tsx
|  |  ├── layout.tsx
|  |  ├── dashboard
|  |  |  |  ├── page.tsx
|  |  ├── settings
|  |  |  |  ├── page.tsx
```

Even if we organize the rest of our application into modules based on the domain, some of the functionality will live in the pages - data fetching, analytics, components, and at least some state.

```
export default function Page({ user }: PageProps) {
  const { items, isLoading } = useItems()

  return (
    <Layout title="All items" user={user}>
      <ItemList
        items={items}
        loading={isLoading}
        pagination={false}
        renderItem={(item) => <ListItem item={item} />}
      />
    </Layout>
  )
}

export async function getServerSideProps() {
  // fetch data, check user permissions, etc.
  return {
    // ...
  }
```

This leads to fragmentation in the structure because even if you use modules to organize your logic, a significant part of each one of them will be outside. You will have to think of where exactly to put hooks, reusable components, and utility functionality. We can deal with this problem by adopting an idea from hexagonal architecture, and using Next's file-based router as an adapter over our logic.

We can leave all our logic living in a module, and only expose a page component which can be rendered in Next's pages.

```
├── app
|  |  ├── page.tsx
|  |  ├── layout.tsx
|  |  ├── dashboard
|  |  |  |  ├── page.tsx
|  |  ├── settings
|  |  |  |  ├── page.tsx
```

```
├── modules
|  |  ├── items
|  |  |  |  ├── ItemsPage.tsx
|  |  |  |  ├── components
|  |  |  |  |  |  ├── ...
|  |  ├── dashboard
|  |  |  |  ├── DashboardPage.tsx
|  |  |  |  ├── components
|  |  |  |  |  |  ├── ...
|  |  ├── settings
|  |  |  |  ├── SettingsPage.tsx
|  |  |  |  ├── components
|  |  |  |  |  |  ├── ...
```

```tsx
// ✅ Leave Next's page only as a light layer on top of your page component
export default function Page({ user }) {
  return <ItemsPage user={user} />;
}

// ✅ Keep the server-side props logic in the page as well
export async function getServerSideProps() {
  // fetch data, check user permissions, etc.
  return {
    // ...
  };
}

// 👉 modules/items/ItemsPage.tsx - Move the actual functionality here
export default function ItemsPage({ user }: PageProps) {
  const { items, isLoading } = useItems();

  return (
    <Layout title="All items" user={user}>
      <ItemList
        items={items}
        loading={isLoading}
        pagination={false}
        renderItem={(item) => <ListItem item={item} />}
```

```
        />
      </Layout>
    );
}
```

We still use the `getServerSideProps` function in the page to prefetch data, validate permissions and anything else that needs to happen prior to rendering. But the actual components and the logic related to them happens in the modules directory. This way we can use the structuring rules from this chapter without compromising. I've seen this pattern applied in many companies that use Next and I've been applying it with great success so far.

2. Components

2.1. Write Deep Components

In his book “A Philosophy of Software Design” - John Osterhout outlined the concept of a deep module. A logical unit that hides a lot of complexity and allows access to it through a simple interface. This idea impacted the way I write software. It translates well to React applications where the logical unit is the component. Each component should allow access to a lot of functionality without requiring a large number of props. Imagine working with a button like this

```
<Button
  icon={}
  iconLoading={}
  iconSuccess={}
  iconError={}
  disabled={}
  text={}
  textLoading={}
  textSuccess={}
  textError={}
  className={}
  onClick={}
/>
```

You can say that this component is highly configurable and flexible. It gives you a lot of options. But does it provide any value? This abstraction doesn’t hide any details. How is using this component different than putting a regular button in place?

This is an example of a shallow component - all its details are exposed through props. Those are the kind that we should avoid. Not because they’re an anti-pattern but because they bring little value. It’s an extra layer that doesn’t remove any complexity.

Components often become shallow because we try to make them do too many things. But the fact that we need a button on most pages doesn’t mean they

should all use the same one. It’s better to work with multiple smaller but deeper components instead of one large shallow one. A way to avoid shallowness is to provide sensible defaults and expose props that can control more functionality underneath.

```
<IconButton
  icon={<UserIcon />}
  onClick={() => {}}
  state={/** "idle" | "loading" | "error" */}
>
  Submit
</Icon>
```

Each button only takes the props it requires and provides good default behavior that abstracts the details of its inner workings. We leave it to the button to manage the icons for the loading and error states, specifying only the main one. It will also manage the text it displays together with the disabled state. The best components are those that provide powerful functionality yet have simpler APIs (props). Their internal complexity becomes invisible.

2.2. Naming Components

Naming is one of the two hardest things in computer science. Thinking of good names for variables is hard because they often represent things that can't be reasoned about. Thankfully, each React component represents some visual item. We can see it and understand its purpose on the screen. Still, I sometimes feel confused and unsure how to name my components.

Finding the balance between a generic name and one that is too specific is hard. Reusable components like buttons and input fields are easy since they don't depend on the domain. But when you're building something specific you'd like the name to provide more context. We want the component to be distinguishable from all the others.

One way to solve that is through the module structure described in the first chapter. Since components are grouped depending on the part of the application that they belong to it's easier to derive context. You at least know what part of the application they're used in.

But looking at a component named Widget in the analytics module doesn't tell you much - it can mean anything. LivePaymentsWidget gives you the idea that it's probably showing real-time payment data. A component named Form in the user module can again refer to multiple things. It can be about authenticating, updating data, or something specific to the business. If we name it UserDetailsForm points to the idea that it's probably used to update the profile.

The rule I follow is that if a description of the element on the screen doesn't provide enough information, I prefix the component with context. You may have components with similar purposes and you want to distinguish them. If you rely only on the module what happens if you need to open two forms in your IDE? It gets confusing looking at two files with identical names and checking where they belong.

The only downside is you can sometimes end up with names that are too verbose. Longer component names can feel like reading a sentence but I think

that’s a good trade-off. Having some extra characters on the screen to remove some ambiguity from the codebase is a bargain.

2.3. Structure Component Files

I had worked with React for years when I realized that my mental model for structuring a component wasn't ideal. Early in my career, I developed the habit to place all helper functions and additional functionality before the component and export it at the bottom of the file. I think I adopted this practice from my Node.js projects where the module.exports statement is at the bottom of the file.

The turning point was when I got into a Reddit debate trying to defend my position. I was sure that exporting components at the bottom was the ideal way to structure a file and I reasoned that all the additional functionality that needs to be used in the actual component needs to be defined before it.

But the person said something that completely changed my view. When we open a JSX we're looking for the component, not the helper functionality. We should make access to it easy by putting it directly in front of the reader's eyes. Then if they want to see the details they can read through the helper functions and components.

That immediately made a lot of sense and even though I'd been exporting at the bottom for years I decided to make amends. The thing is that many components have a lot of functionality living with them - functions, interfaces, constants, configuration, styled components. Putting them at the top means that the reader needs to scroll through everything to find what they're looking for. The file simply becomes too noisy.

Instead, put the most important bits first - imports, interface, and the exported component. Pull everything crucial for that piece of functionality up so it can be seen immediately. If you feel that a constant or a configuration object needs to be there you can make an exception for them too but try to avoid that.

```
// Import statements
import { useState, useEffect } from 'react'
import useRouter from 'router'
```

```
// Put the interface here if you're using TypeScript/Flow
// It carries important information for the component

// Exported component's definition
export default function Component() {
...
}

// Utility functions, CSS-in-JS components, everything else...
```

2.4. Structure the Component Function

The same way we structure our modules and component folders we should be consistent in the way we implement the component. Repetitive patterns help to understand the logic and remove questions like "Where exactly should I put this:. Separate each component function into 4 logical parts - setup, logic, guard clauses, and markup.

```
export default function Component() {
  // Setup - state and hooks
  const [loading, setLoading] = useState();
  const { path, query } = useRouter();

  // Body - do something with the component's data

  // Guard clauses - prevent rendering if needed
  if (loading) {
    return null;
  }

  // The main returned result of the component...
}
```

The setup is where you define state and hooks. The logic is the body of the component where you prepare data to be displayed in the markup. If this part becomes too long that is a sign that some of it can be extracted and done in the setup part (as a custom hook).

Guard clauses are used to short-circuit the rendering process. They're conditional statements that decide whether you should render the component or something else. In case you're still waiting for data or there's an error you will most likely want to display a message or a loading indicator. It's a good practice to put them before the return statement to reduce the amount of conditional logic in the markup. Otherwise, you will have to make a lot of checks. Then you have the happy path of the component - the JSX that you want to render in normal circumstances.

2.5. Use Context as an Implementation Detail

The React Context is a powerful tool to share data between different parts of the component tree. It's a solution to the prop drilling problem and allows us to build better component hierarchies that are not deeply nested.

It's important to note that Context on its own is not a state management tool. It's a form of dependency injection - it's used to inject data or functionality in different parts of the application. Combined with useReducer it can be used to manage complex state that is beyond regular state management capabilities.

React Context's setup and usage are considered verbose by many. That's the reason many developers avoid it - if they have to work with context they'd rather use an external library that has a better API. But we should keep in mind that many of the state management libraries actually use context under the hood.

I always prefer to stick to pure React whenever possible and I don't use state management libraries unless the nature of the project absolutely demands it. They will affect our application's structure and logical flow and I want to avoid that whenever possible. Also, we don't have control over their APIs.

Using Context directly can be confusing and verbose so it's best to use it as an implementation detail. To get the most of it we should build a light layer around it that wraps its complexity and exposes a simpler API. We have complete control over what we want to expose and what we want to hide. We can make that functionality blend into our existing structure rather than alter it.

Then when we need to pull something from Context, we can use a simple custom hook that gives us direct access to the values. The components don't need to be aware of where those values are coming from. They don't need to know that there's a Context involved at all.

```
const StateContext = React.createContext();
const reducer = function () {
```

```
  // ...
};
const initialState = {};

// A common pattern is to create a custom provider
export function StateContextProvider({ children }) {
  const [state, dispatch] = useReducer(
    reducer,
    initialState
  );

  // Cache the value we pass to avoid too many rerenders
  const contextValue = React.useMemo(
    () => ({ state, dispatch }),
    [state, dispatch]
  );

  return (
    <StateContext.Provider value={contextValue}>
      {children}
    </StateContext.Provider>
  );
}

// Create a custom hook
export function useGlobalState() {
  return React.useContext(StateContext);
}
// Use the custom hook without referencing the context
function Component() {
  const { state, dispatch } = useGlobalState();
}
```

Since we're implementing this ourselves we need to be careful with performance. In this example we use React.memo to memoize the context value because if we pass an object directly we will rerender the provider too often.

To make this even more powerful we can abstract away the dispatch function as we did in the "Abstract Reducer Details" rule. That completely removes any

details related to React and gives the user a simple set of functions to use without worrying about rules, updates, and reducers.

Another pattern that I've seen is creating two different context providers - one that carries the state and another one for the dispatch functions. This way you can have components access only the state or functions depending on what they need.

2.6. Create Custom Hooks

Hooks are the idiomatic way to compose functionality in a component. They are a great way to abstract complex logic, group other hooks together in a single logical unit, and reuse functionality between components.

If you need to fetch data and modify it before rendering it, it can be extracted in a hook. This way the components that use it will only work with the clean result and they don't have to implement the business logic. Their responsibility will be to render the prepared data. The modification logic should be an implementation detail.

If you need to check whether a user's authenticated and take actions based on that, extract it in a hook. A component doesn't need to be aware of the details of how a check is done. It should only call the hook.

Any generic behavior needed in UI applications like tracking the scroll percentage or an element's visibility is good candidates for custom hooks. If you need to use a browser API like the IntersectionObserver you can wrap it in a hook. You can handle things like polyfills, configuration, and set up there, providing a clean API for your components.

Above everything, custom hooks come in handy when we want to group other hooks together. As a general rule whenever a component is doing something non-trivial with hooks, it's best to encapsulate that behavior. Whenever you put five hooks in place, consider grouping the logic together in one hook or using a reducer.

You can find the following code in most React projects. It's not necessarily bad, but imagine how complex it would become if you need to add a couple more hooks and handlers in there.

```
function Component() {
  const [state, setState] = useState(false);

  const on = useCallback(() => setState(true), []);
```

```
  const off = useCallback(() => setState(false), []);
  const toggle = useCallback(
    () => setState(!state),
    [state]
  );

  // ...
}
```

Let's refactor this. We can extract that logic in a custom hook. The component doesn't have to know the details of how the toggle works. Extracting it in a custom hook allows us to provide a more declarative API.

```
function useToggle() {
  const [state, setState] = React.useState(false);
  const handlers = React.useMemo(
    () => ({
      on: () => setState(true),
      off: () => setState(false),
      toggle: () => setState(!state),
    }),
    [state]
  );
  return [state, handlers];
}
```

This hides all the complexity in a small easy-to-read function which can be used like this:

```
const [toggleState, { on, off, toggle }] = useToggle();
```

When you're only using two pieces of state it would be simpler to keep them directly in the component. But if you need to implement logic around them it's better to extract it and let the component work with the API. We have the habit of giving components too many responsibilities. Instead, they should be the place where all logic for this element comes together. It doesn't have to be fully implemented in the component.

This toggling logic can be reused in many places across your application. Modal windows, toast popups, loading spinners require such functionality. Instead of implementing it for each one, you now have a generic toggling hook that you can put to use.

Don’t shy away from creating abstractions when they could bring value.

2.7. Abstract Reducer Details

Reducers are great when you need to manage complex state. But as complexity grows, the reducers leaks details in the implementation. Your components get cluttered with verbose dispatch calls and you need to manage the actions' names with constants.

Suddenly, you find yourself implementing your own Redux. This becomes a bigger issue if you need to pass the dispatch function down to child components. The whole component tree becomes coupled to the implementation.

To avoid this, create an abstraction over useReducer as a custom hook. Contain the complexity there and export functions with the dispatch type predefined. Avoid calling the dispatch functions directly. It should be used as an implementation detail and wrapped in another function.

```
function useBooks() {
  const [state, dispatch] = useReducer(
    reducer,
    initialState
  );

  return {
    books: state,
    selectBook: (index) =>
      dispatch({ type: "select book", payload: index }),
    updateBook: (book) =>
      dispatch({ type: "update book", payload: book }),
  };
}
```

This way the component can access the state and update it using simple functions that are easier to use. It doesn't require knowledge of the underlying implementation. It reduces the possibility for errors when setting the dispatch type and makes adding new actions more safe.

```
function Component() {
  const { books, selectBook, updateBook } = useBooks();

  // ...
}
```

2.8. Avoid Short-Circuit Conditional Rendering

The most concise way to render conditionally is the short-circuit approach.

```
function Component() {
  const count = 0;
  return <div>{count && <h1>Messages: {count}</h1>}</div>;
}
```

But in some situations, short-circuiting may backfire and you may end up with an unwanted “0” in your UI. Why is that? Isn’t `0` a false value?

The way comparison operators work in JavaScript is they don’t return “true” or `false` - they return one of the values of the comparison. In this case, the returned value is `0` and since it’s a number it will get printed on the screen.

To avoid this, use ternary operators by default. This way you explicitly provide the desired result in both scenarios. The only caveat is that ternaries are more verbose.

```
function Component() {
  const count = 0;
  return (
    <div>{count ? <h1>Messages: {count}</h1> : null}</div>
  );
}
```

This is harder to read than the short-circuit approach, especially in cases with long markup. But in this scenario, we want to reduce the chance for possible errors and remove ambiguity. Ternaries provide more clarity because both results are visible to the developer.

Usually, it’s better to go with the more concise option. Less code means fewer opportunities for bugs to sneak in and less code to test. But the fact that something is more verbose doesn’t make it a bad approach. In some cases, short implementations make incorrect behavior less obvious because the

developer doesn’t have to think about the details. In such cases, I favor clarity and safety over simplicity.

2.9. Avoid Nested Ternary Operators

Sometimes the conditional logic is too complex to be handled with a single ternary operator. The chain of checks gets longer and if we chain ternary operations, it won't be easy to read. Yes, you could always format this to look better but I don't want to write code that is readable only in specific circumstances.

```
isSubscribed ? (
  <ArticleRecommendations />
) : isRegistered ? (
  <SubscribeCallToAction />
) : (
  <RegisterCallToAction />
);
```

Instead, separate the logic in its own component and use guard clauses. Avoid using if-else blocks as well, they don't differ that much from the ternaries. Instead, return early - it's only a single logical path this way. It also makes extending easier - just add another if statement.

```
function CallToActionWidget({ subscribed, registered }) {
  if (subscribed) {
    return <ArticleRecommendations />;
  }
  if (registered) {
    return <SubscribeCallToAction />;
  }
  return <RegisterCallToAction />;
}
```

It's always better to be obvious in our intentions. This is more verbose but it's easier to follow and is abstracted in its own component. It removes the complexity from the parent one.

2.10. Avoid Long Conditional Chains

When a component has complex conditional logic, the number of guard clauses can become too high to handle. The component becomes too imperative and may rely on the specific order of conditions that we want to avoid.

When this happens, we can use an object as a component map and drastically reduce the amount of logic we keep in the body of the component.

```
function Prompt({ state }) {
  const Component = PromptComponents[state];
  return <Component />;
}

const PromptComponents = {
  guest: GuestPrompt,
  registered: RegisteredPrompt,
  subscriber: SubscriberPrompt,
  promotion: PromotionPrompt,
  pro: LatestStories,
};
```

I find it even easier to follow than using guard clauses. Instead of writing logic, adding another component is just a matter of configuration. The component function is even simpler. ` Component maps also communicate that all those conditional cases are equal in importance. Guard clauses may be understood as exceptional cases when the golden path can't be rendered. When using a component map, it's obvious that the purpose of this component is only to decide what to render.

When using TypeScript you can use an enum to make sure that the passed prop always corresponds to an existing component in the map. If you can't add a static type, consider adding a check and throwing an error, or communicating that a wrong identifier was provided.

2.11. Use Component Maps for Complex Conditional Logic

Using a simple object for a component map is great when the only thing that varies is what we want to render. However, often we want to modify the behavior as well, we may want to call a different function based on the passed identifier. There may be additional places in the markup that we want to alter as well.

To avoid creating multiple maps or a complicated conditional chain we can unify the behavior in a single, more complex object.

```
function Prompt({ state }) {
  const { title, Component, handler } =
    PromptComponents[state];
  return (
    <div>
      <h3>{title}</h3>
      <Component />
      <button onClick={handler}>Submit</button>
    </div>
  );
}
const PromptComponents = {
  guest: {
    title: "Register to read 3 free articles",
    Component: RegisterPrompt,
    handler: handleRegistration,
  },
  registered: {
    title: "Subscribe for unlimited content",
    Component: SubscriberPrompt,
    handler: handleSubscription,
  },
  // ...
};
```

This approach allows you to manage more complex scenarios without increasing the complexity of the implementation too much. In all cases, you want to avoid having conditional logic for exceptional cases in the markup. The map should provide you with the pieces and the component should just fill in the blanks. It shouldn't be making any decisions because that beats the purpose of this approach.

In situations in which only one component in the map differs in implementation, it's tempting to make that using an inline conditional check. But even in those cases, it's better to be explicit and keep all the logic in the map.

2.12. Avoid Top Level Conditionals in JSX

When deciding whether to render a component or not, avoid using top-level conditional rendering. The return clause of the component should specify its happy path - the markup that it wants to render when everything with its data and configuration is fine. All the conditionals that validate its behavior should be used before the final return statement.

```
function Component({ loading }) {
  return loading ? null : <div>...</div>;
}
```

Use guard clauses and return early when a condition is met instead. Loading, errors, insufficient access - in all those cases you will want to cancel the execution. More often than not you will want to return a different component or a message and using guard clauses provides the most clarity.

```
function Component({ loading }) {
  if (loading) {
    return null;
  }
  return <div>...</div>;
}
```

But most importantly, they separate the main execution path from all the exceptional ones. When a developer opens the component file, looking to check something, they know where to focus. With this approach, we achieve extensibility (new guard clauses can be added easily), simplicity (no complex conditionals), and clarity (the main execution path is obvious).

2.13. Create a Common Error Boundary Component

We can take one thing as an absolute - things will fail. It's impossible to create something which is 100% reliable. Even if we handle all cases, there's still the chance that something goes wrong in the network and our API call fails. In some cases, it's beyond our capabilities to prevent errors.

Failures are common. We call them exceptions but there's nothing more common than them. We shouldn't be avoiding them or working around problems. We should embrace them and make them a part of our normal development process. We should plan for them, design for them, and implement the business logic always keeping them in mind.

In large projects, one of the first common components you should make is an error boundary. This is a component that can catch an unhandled error so your application doesn't fully crash. It limits the impact of the problem and allows you to handle the error however you see fit.

```
<ErrorBoundary
  message="Oops, something went wrong"
  fallback={<CartErrorMessage />}
>
  <Cart />
</ErrorBoundary>
```

The React docs have described how to implement error boundaries in detail so I won't be repeating them. It's best to implement such a component early to avoid ending up with multiple implementations and encourage the team to use it.

It should print out a default message, GIF, or image that will communicate there was an error even if it's not configured. However, you should accept a message or a fallback component to display in case anything goes wrong. For

example, you may want to display a form to contact an administrator if a dashboard chart throws an error for some reason.

2.14. Use Error Boundaries for Non-Critical UI Sections

The reason we use error boundaries is that we don't want to take down the entire UI because of a single error. There are rare cases in which we want to do that - critical data is missing or the most important component breaks. In those situations, we want to make the application unusable.

But most of the time, we can manage the error on the component level. The question is where exactly to put an error boundary. When you take a look at a page split it into logical parts - header, sidebar, details section, widgets, etc. Then consider which of those do complex data manipulation, calls to external APIs, or rely on user input. All those are potential opportunities for the application to crash.

Wrap the non-critical sections but make sure they don't break the UI when they're missing. We shouldn't break the layout if a widget is missing because of an error. Make sure that your layout supports not having elements wrapped in error boundaries. A better alternative is to always return some wrapper element as a fallback to make sure it fills the space on the page.

When you're managing the error you can also fall back to the same component but give it some basic props or render it in "preview only" mode. This way we can put the page in a working state. Usually, I add an error boundary on the top level as well. I want to make sure that the user doesn't see a blank page if something goes terribly wrong.

An error can be an opportunity for engagement as well. Make your product team flex their creativity and see how you can use error boundaries to your advantage.

2.15. Use Reducers for State Management

No matter how granularly you split your components some of them will need to manage higher complexity than others. The biggest factor is the internal state. Even in functional components, managing state is not only about displaying and calling the update function. Usually, it requires additional handler functions and logic. The more pieces of state a component needs to manage, the more difficult it is to follow how they change.

```
function Component() {
  const [isOpen, setIsOpen] = useState(false);
  const [type, setType] = useState(TYPES.LARGE);
  const [phone, setPhone] = useState("");
  const [email, setEmail] = useState("");
  const [error, setError] = useState(null);

  //...
}

const TYPES = {
  SMALL: "small",
  MEDIUM: "medium",
  LARGE: "large",
};
```

When you need to do complex state modifications or just manage multiple fields, the useState hook's capabilities are not enough. React provides a powerful tool for those purposes - the `useReducer` hook.

It's a great mechanism and I prefer it to external libraries. It allows you to model all changes in the UI as a result of some action. It's similar to Redux with the difference that it can be done on a single component level and it doesn't require any dependencies. It's similar to how the "Elm Architecture" and ReasonReact work.

```
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
  );

  // ...
}
```

Boilerplate is the biggest downside to this approach. But in exchange for that, you get the ability to manage more data descriptively. To extend the logic you only need to add another action in the reducer instead of modifying the body of the component. As a general rule, when you find yourself using multiple pieces of state - refactor to a reducer function instead.

2.16. You May Not Need a State Management Library

State management libraries will have a great impact on your application's data flow, design patterns, and mental models. You will need to manipulate and access data in line with the library's philosophy.

Most applications can be implemented with the built-in state management capabilities that React provides. Most small and medium projects don't need external libraries anymore, especially since the introduction of useReducer and Context. But there is a line in which they start hindering you and you need a more powerful tool.

State management libraries shine in large applications or complex ones with many interactions. There isn't a clear threshold that signals when using an external tool is a good idea but here are some pointers.

Your application's state is large and it updates frequently. We're not talking about a theme and some user settings but large quantities of data that are used in multiple places. If it needs to be updated often using a tool will make it easier.

The logic to update the state and create side effects is too complex. Making API calls in response to an event is easy when the interactions are not that many. But at scale you need a standardized way to do that and picking a library is a good bet.

You're building a large project with many people. State management libraries bring a standardized development flow. Everyone familiar with the tool will describe their logic in the same way.

You want to understand the way your application's state changes over time and some of the state management libraries provide powerful introspection tools. This book won't give advice on what specific library to use but I would strongly encourage you to consider Recoil, Jotai and Zustand. They are all implemented around the basic ideas of atoms and provide idiomatic-to-React

ways to interact with state via hooks. They are all quite minimalistic and won't have an effect on your application's structure or design.

2.17. Use Data Fetching Libraries

React doesn't come with an opinionated data fetching mechanism. Each team creates its implementation usually involving a service module that exposes functions to communicate with the API.

This gives us a lot of freedom but it also means that we need to manage loading states and handle errors on our own. Most data fetching logic consists of boilerplate around those two scenarios - managing spinners and showing error messages.

The implementation is trivial but we don't want to clutter our components with the same repetitive logic. This is an ideal case to create a common abstraction in the form of a custom hook. Thankfully, there are open-source libraries that have done exactly this.

Modules like React Query, SWR, or Apollo Client (if you're using GraphQL) make communicating with a server a natural part of the component lifecycle. We can use a simple hook that manages loading and error states for us. We just need to handle them in our UI by showing the appropriate component.

A big reason to use state management libraries in the past was data fetching. They made managing states and caching easier. Data fetching libraries, like the ones mentioned above, are lighter alternatives that come with everything we need and don't alter our whole application's structure. The only thing that we need to provide is a fetcher function that pulls the data. It is best kept as a part of the business logic inside the "api" or "core" folder in your module.

Also, remember that you should always follow the "Create Custom Hooks" rule. Whenever possible we want to use hooks because they can be composed together. If there's additional complexity involved in the data fetching (aggregating or formatting the data) we can make a custom hook that uses React Query underneath and returns the formatted data to the component.

2.18. Favour Functional Components

Class components are more verbose and the different lifecycle methods introduce extra complexity. In the past, they were our only option whenever we wanted to work with the state.

```
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 have simpler syntax. No lifecycle methods, constructors, or boilerplate. You can express the same logic with fewer characters without losing readability. Since the introduction of hooks, there is no reason not to use functional components by default.

The mental model you need to keep in your head is a lot smaller. The whole component executes each time together with all its hooks, no need to remember when exactly each method is called.

```
function Counter() {
  const [counter, setCounter] = useState(0);
  handleClick = () => setCounter(counter + 1);
  return (
    <div>
      <p>counter: {counter}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}
```

I prefer functional components for consistency - they can all follow the same structure making them easy to understand. It makes refactoring easier as well. If you need to add state to a stateless component you only need to use the hook. Otherwise, you'd have to refactor it to a class.

Hooks make the code easier to read since each render executes it from top to bottom. Lifecycle methods give you more precision but the logic jumps from method to method. Each one takes different arguments and has a different effect on the returned value. With functional components, it's just state, props, and returning markup.

The only exception to that is error boundaries, you still need a class component for them since they can't be implemented with a hook (I expect that to change in the future).

2.19. Write Consistent Components

Some of the rules in this book describe good practices about component structure and design. But even if you don't follow them to the letter it's important to be consistent in your codebase. You want to keep helper functions in the same place across all your components. You want to follow the same naming patterns, the same overall design for your component.

Consistency is important because it reduces the mental load that each developer will have to take to understand your codebase. You want all components to look as if they were written by the same person even if there's a team of ten working on the codebase.

Imagine reading a book where each chapter was written by a different author. It would take you some time to get used to their writing style and word choice. It's the same in programming. There aren't any real benefits of one coding standard over the other. It's all in the eyes of the beholder. But you won't get benefits out of any style guide if you apply the rules in it selectively.

Maybe you feel uncomfortable writing comments on someone's pull request because of a small stylistic inconsistency. But you should treat those as a code smell. If they deviate from the agreed standard that everyone follows you should ask for the code to be reviewed. Inconsistency hurts your codebase in ways that are not obvious. If we could find a way to calculate the amount of lost time per year because of inconsistent structure, we would start taking it seriously.

The exception to this is if you inherit a codebase. You are thrown in a product with established practices that don't adhere to your team's understanding. You may not be able to refactor everything but you can improve it. Create new components formatted with a consistent structure. Whenever you need to touch the old ones, try to get them closer to the new standard. Aim to leave things better than you found them (also known as "the boy scout principle").

2.20. Always Name Components

At the time of this writing, I still find codebases with components implemented using anonymous functions. I've never understood this practice - it doesn't save you a lot of writing but can leave you in perplexing situations.

```
export default () => <form>...</form>;
```

Anonymous functions should be used when we're passing a callback to a function, there isn't any benefit of using them for components. Naming a component means that we will get better error messages and we will pinpoint problems faster. It would be hard doing that with a component tree made of anonymous functions.

```
export default function Form() {
  return <form>...</form>;
}
```

Even if we can live with the bad error messages, we won't be able to see the components' names in React DevTools. We'd have to explicitly provide them but I wouldn't do that if the names can be inferred from the component function's name.

The last problem is related to developer experience. We rarely work on a single component. Most times I have multiple JSX files open at any time. If the component function is not named I would always need to look at the file name to make sure I'm editing the right things. In the rare cases in which we have multiple components named the same way (e.g. form components in different modules), we'd have to check the file paths and that is not ideal.

2.21. Don’t Hardcode Repetitive Markup

When you’re building a list of filters, a navigation, or some other repetitive component whose contents are not dynamic, it’s tempting to hardcode the items in the markup.

```
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>
    </>
  );
}
```

This implementation is easy to manage for small components but it means that changes need to be done in multiple places. If you need to redesign this

component you will have to do a lot of copying and pasting. This isn't a code smell and it won't cause you that much trouble but it leaves room for errors and leads to a more imperative code style. The fact that some contents are fixed doesn't mean that we should abandon our principles for software development. A rule I follow is not to work with hardcoded data in a way that I wouldn't work with a dynamic one.

The argument most developers have is the one for simplicity. Since the data isn't dynamic, it would be simpler to just hardcode it in the markup. Yes, it may be simple to write such code but it won't be simple to read and maintain it. Such a file may easily get a few hundred lines long. Would you rather maintain that or maintain a more descriptive implementation?

```
function Filters({ onFilterClick }) {
  return (
    <>
      <p>Book Genres</p>
      <ul>
        {GENRES.map((genre) => (
          <li>
            <div
              onClick={() =>
                onFilterClick(genre.identifier)
              }
            >
              {genre.name}
            </div>
          </li>
        ))}
      </ul>
    </>
  );
}

const GENRES = [
  {
    identifier: "fiction",
    name: Fiction,
  },
```

```
  {
    identifier: "classics",
    name: Classics,
  },
  {
    identifier: "fantasy",
    name: Fantasy,
  },
  {
    identifier: "romance",
    name: Romance,
  },
];
```

The alternative is to use a configuration object with a strict format and render the data from it. This way we only define the markup once and we change it in a single place. Also, when adding or removing filters we don't risk breaking the markup. We just add or remove a field in the object.

The configuration approach improves the extensibility of the implementation. The filters example is simple but if the hardcoded entries require a new field that would involve a lot of manual work. Copying and pasting is easy when it's done in a couple of places. If you have to do it more than that you increase the possibility for errors.

2.22. Organise Helper Functions

Many of your components will implement some kind of behavior. Whether it is event handlers, requests to external APIs, or transforming a piece of data. When that logic is short and easily grasped it can be placed inside the markup. It won't break the reading flow and it even provides more context to the person reading the code.

But when those functions are long or complex it's better to extract them from the markup and keep them separated. If people want to get familiar with their workings they can scroll and read through the function. Otherwise, they can look at them as a black box.

```
function Component({ date }) {
  function formatDate() {
    // ...
  }
  return <div>Date is {formatDate()}</div>;
}
```

Knowing when to move a function outside of the JSX is a matter of feel. Sometimes a longer function is better inlined - this is often the case with event handlers. In other situations, you will want to extract the logic to a function, even if it's short, just so you can name it and provide context about what it's doing.

Another consideration is where to keep the extracted function - inside the component or outside. My rule of thumb is to always put them outside unless they need to hold a closure on the whole component. They should work as pure functions that take input and produce output.

```
function Component({ date }) {
  return <div>Date is {formatDate(date)}</div>;
}
function formatDate(date) {
```

```
    /// ...
}
```

If a function is inside the component's body it will be able to access its whole state. We want to be explicit about where each piece of state is used and what a change in it will affect. Because of that, it's better to keep all the helper functions outside, bellow the component implementation and pass the values they need as arguments.

We should aim to reduce the size of the component implementation and putting the functions outside is a good first step. By working with pure functions we make it easier to spot bugs and extend the functionality in the future.

2.23. Favour Small Components

React applications are easier to manage when they're built from many small components. Components are just functions - they take input (props) and return a result (JSX). The good thing about React is that it's closer to the language. It is subject to the same software design rules that we use in other projects.

Functions need to be small and do one thing (have one responsibility). In the same way, each component should render an element on the screen and put together the logic for it. When designing our components we should actively be looking for opportunities to split larger elements into multiple smaller ones. The more granular our UI is, the easier it will be to work on.

We can judge a component's responsibilities by its length. It signals how much a component is doing and how many responsibilities it has. Smaller components are focused on one thing. That makes them easy to read - there are fewer branching conditionals and side effects. It's easier to test since there are fewer edge cases and less logic. You can be more confident when making changes - you're not affecting anything else because the component has only one responsibility.

There isn't a strict limit on how long a component should be. Line count is not an objective measure. Sometimes a component can be lengthy because it renders complex markup. Think about responsibilities instead. Ask yourself what the component is doing. If you need to use the word "and" in its description then it may be a sign that it needs to be split.

But what do you do with complex components that represent a whole page or a larger widget? They're bound to have more logic in them. Yes, they can be the main entry point but they don't have to implement all the functionality themselves. They should be the place where all the logic comes together but it should be split across different child components. That way the complexity is spread across many layers and even those main components that represent a page are easy to understand.

2.24. Write Comments in JSX

When building UI applications we describe a large part of the business logic through markup. Take conditional statements for example. You decide what to render depending on the user's data and preferences. Even stateless components hold a piece of the domain knowledge when they fire off events.

We're used to writing comments in regular functions. We describe their highlevel purpose before the implementation and provide more context inside. But when it comes to components we don't have that habit because the most complexity is usually held in the markup. I encourage people to write comments, even in JSX, when they can make the implementation clearer to someone who isn't familiar with it.

```
function Component({ user }) {
  return (
    <>
      {/** Only authenticated users can shop. */}
      {user.anonymous ? <AuthPrompt /> : <Cart />}
    </>
  );
}
```

I believe that comments are a great tool when used properly. Many people advocate for self-documenting code that doesn't need comments but for all my years as a software engineer I'm yet to find such code. We're not domain experts from the start and comments are helpful for anyone new to the business.

Programming languages are limited. Their vocabulary is many times smaller than that of the English language. Comments should tell you "why" something is done. The code should describe "how" it works. It's impossible to do both with the vocabulary of your programming language.

It's better to be explicit in your intentions than have someone scratching their head two years later when they inherit the codebase. When something needs

more clarity - provide it.

2.25. Always Destructure Props

The best thing about React is that it's close to the language. Its main building block, the component, can be expressed with a simple JavaScript function. It takes props, returns JSX, and occasionally produces a side effect or two.

In a regular function, the arguments are passed separately. React components are different in that regard. They receive only one argument - props. Everything passed to the component is inside that object and it can be accessed by name.

This design decision was made because components can accept a lot of props. Imagine the component function taking six or seven arguments. Also, it would make the props order specific and that approach is prone to errors when the number of arguments is high.

```
function Input(props) {
  return (
    <input value={props.value} onChange={props.onChange} />
  );
}
```

When we receive an object we can access the different values by name without thinking about order. It's way simpler than the alternative. Deno, the new JavaScript runtime, also uses a similar pattern for functions that take more than three arguments. They wrap them in an object to encourage better design.

This pattern helps us accept props reliably but it doesn't mean that we should access everything from the props object. In regular functions, we work with the values directly, so it makes sense to apply the same principle here.

```
function Input({ value, onChange }) {
  return <input value={value} onChange={onChange} />;
}
```

A reason not to destructure the props would be to distinguish what is external data and what is internal state. But in a regular function there is no distinction between arguments and local variables, so let's avoid creating new patterns. Working with values directly is simpler.

2.26. Be Careful With the Number of Passed Props

Since a React component is just a function we should follow the same design principles we follow in any language. If a function accepts too many parameters it may be a sign that it has too many responsibilities. I wouldn't split a function just based on the number of arguments it takes. But if I pass more than three arguments to one, I would review it.

React components usually take more props so my threshold there is higher. When I go over five passed props to a component, I would see if it should be split. Again, I don't make that decision based on length and number of props alone but based on responsibilities. In some cases, a component just needs more data and that's fine.

Too many unrelated props are a good sign that a component can be split. Passing the same configuration parameters over and over means that default values can be set inside the component. Then we will have to pass fewer values when using it and pass more props only in specific cases. The number of props is a subjective measure so only take it as a call to review. It's not a code smell right away.

A large number of props is not just a signal of mixed responsibilities. It's often a sign of a leaky abstraction. This happens when a component exposes too many of its details. The fewer details a component hides, the less valuable it is. This is often the case with highly configurable components that expose every single HTML attribute or event that they can receive. In such cases, it's better to write the implementation directly instead of using an abstraction.

It's better to split such components into smaller ones that are more specific and provide good defaults for their use cases. Another approach would be to use the highly configurable component as an implementation detail for other reusable components that come with default configuration again. Still, I would favor the first approach since it's simpler.

Lastly, the more props a component takes, the more reasons it will have to rerender. We shouldn't be designing our APIs with performance in mind but

it’s worth keeping that in consideration. It’s another reason why components should receive only what is necessary for their work.

2.27. Assign Default Props When Destructuring

I avoid using the defaultProps property that React provides to set default values. When there's an alternative built into the language I would rather use it than a technology-specific approach. If we use defaultProps we split the information we have about the values in two places. In longer components, the developer will need to scroll back and forth to see the default values.

```
function Component({ title, tags, subscribed }) {
  return <div>...</div>;
}
Component.defaultProps = {
  title: "",
  tags: [],
  subscribed: false,
};
```

Since JavaScript provides a way to specify default values during destructuring, I would always favor that approach. It makes the code easier to read from top to bottom and keeps all information about the props together.

It's more natural to me to have a glance at the props while looking at the implementation. If I'm using TypeScript and I have an interface it's most often above the component definition. This means that everything I need to know about the props and their values is always located in the same place in each component file.

```
function Component({
  title = "",
  tags = [],
  subscribed = false,
}) {
  return <div>...</div>;
}
```

2.28. Pass Objects Instead of Primitives

Passing a large number of props to a component may be a sign of mixed responsibilities or a leaky abstraction. But even when this is not the case, I still consider ways to improve the design of my components. One way to do that is to group related data instead of passing it as separate props.

```
<UserProfile
  bio={user.bio}
  name={user.name}
  email={user.email}
  subscription={user.subscription}
/>
```

This makes it clear exactly which fields of user object this component is using. If the object is large and we're only accessing selected pieces from it, it's better to be explicit and pass them like that. Even if the implementation is more verbose.

But when we are using most of the data from an object, we should pass it directly and let the component deal with it. It can destructure it and still use the values directly so it won't affect its implementation. This also reduces the changes that need to be made if this object gets an extra field in the future.

```
<UserProfile user={user} />
```

2.29. Avoid Spreading Props

One practice that I avoid is to spread an object when I want to pass its fields to a component. To me, this is a bad design decision because it doesn't make it clear what values are passed to the component. It's not obvious if the component uses the whole object or just some values from it. It saves you from writing a few extra lines of code but increases the level of obscurity.

```
<Article category={category} author={author} {...article} />
```

What is the difference between this and grouping fields together (the approach in the previous rule)? They seem the same at first but the intention is different. When we're passing a whole object to a prop, the component has communicated that it wants to be passed an object. We use the component's API and let it deal with pulling the properties it needs.

When we're using the spreading approach, it's not clear what props we are passing to the component. Instead, either modify the component to accept the whole object or explicitly pass the fields that are needed.

```
<Article
  author={author}
  category={category}
  title={article.title}
  url={article.url}
  content={article.content}
/>
```

2.30. Move Lists in Components

One of the most common things we do as UI developers is to display lists of data. Rows in a table, products in an online shop, even the tags in a blog - they are all dynamic lists that we need to map over and turn into markup.

```
function Component({ topic, page, articles, onNextPage }) {
  return (
    <div>
      <h1>{topic}</h1>
      {articles.map((article) => (
        <div>
          <h3>{article.title}</h3>
          <p>{article.teaser}</p>
          <img src={article.image} />
        </div>
      ))}
      <div>You are on page {page}</div>
      <button onClick={onNextPage}>Next</button>
    </div>
  );
}
```

It's not a complex operation, it's one of the most trivial ones. But when we have to map over a collection in a component that already has long or deeply indented markup, the listing will further hurt its readability. If you have more than a single loop and a few conditional statements here and there, the markup will become a puzzle or braces and curly brackets.

But the problem with lists is not just the length of their markup, they often carry a lot of complexity with them. We often need to filter or enrich the collection before rendering it. We may need to implement an event handler as well. To avoid putting too much complexity in one place, I extract lists in their own component, even if the markup isn't much.

```
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>
  );
}
```

The parent component doesn't have to know the details of the list. It just passes the data as a prop and it's the child component's responsibility to handle it from there. It should deal with parsing and rendering. If it needs to communicate an event to the parent it can accept a prop.

The exception to this rule is a simple component where displaying the list is one of the only things they are doing. If there isn't much markup or there is only a single level of indentation it should be okay to keep it there. It's a matter of feel but when you're in doubt - extract it.

2.31. Avoid Nested Render Functions

It's surprisingly common for React developers, even experienced ones, to create nested render functions inside the component. They are usually responsible for a section of the markup and the logic and state related to it.

Those functions are nested to communicate that the markup is specific to this component only.

```
function Component() {
  function renderHeader() {
    return <header>...</header>;
  }
  return <div>{renderHeader()}</div>;
}
```

If there is one pattern that you should avoid - it's this one. We extract functionality because it's long or complex. But since components are just functions, nesting such a function is the equivalent of defining another component inside the main one. The drawback is that it relies on a closure instead of communicating the data it needs via props. It doesn't hide any complexity since the logic is still in the parent component.

The nested component has access to all of the parent's state instead of the fraction it needs. This practice hurts readability because it scatters JSX all over the component. When I find JSX I subconsciously know that I've reached the return statement of the component. But this is not the case with nested functions. We should aim to write components that can be read from top to bottom. This is not possible because you have to jump up and down in the same file to follow the logical flow.

I haven't found any benefits of using nested functions. We don't keep multiple components in the same file, so we shouldn't define them in the same function.

```
import Header from "@modules/common/components/Header";

function Component() {
  return (
    <div>
      <Header />
    </div>
  );
}
```

Instead, extract the logic in its own component. React is close to the language so we will still be using a function. But instead of relying on closures, we will be using the new component as a pure function. It will define the props that it needs to render properly and we will have a more descriptive way to use them. It will hide the new child component's complexity and make the parent one easier to read. If we want to communicate that the child component is private to the parent one, we can place it inside a components folder in the directory of the parent.

2.32. Favour Hooks

In some cases, we need to enhance a component or give it access to an external state. Historically, there are three ways to achieve this - higher-order components, render props, and hooks. We won't cover the basics of each of those approaches since they are well described in the docs. Instead, we will focus on the design decisions around them.

The trouble with higher-order components is that they make it unclear where the data is coming from. Everything is passed as props. It's impossible to compose a few of them together, you need to create a hierarchy much like in object-oriented programming. That is known to cause problems if one of the components in the chain doesn't propagate props properly.

Render props put too much complexity inside the markup of the component. They force your logic to be handled there while this should be a decision left for the developer. I prefer to prepare my data in the body of the component and only render it in the returned JSX. Using render props leads to increased indentation and bad readability.

```
function Component() {
  return (
    <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>
    );
}
```

There is nothing inherently wrong with higher-order components or render props. They are just more complex. Hooks have so far proven to be the most efficient way to achieve composition. Looking at it philosophically, a component is a function that is composed of other functions. Each hook gives it more functionality.

Hooks are more descriptive than the alternatives and they can be composed to create more powerful abstractions. The only magical thing about them is the rule of hooks - they always need to execute in the same order. I think that this improves the component design because we know that each time we execute the whole component's logic in the same way. Some developers dislike the rules of hooks, but at least they are strict. Working with higherorder components and render props involved conforming to some rules too, they are just not well defined.

Hooks don't impact the markup which is a big factor. The logic a component uses shouldn't be reflected in its markup. That puts too much responsibility in the presentational layer. All things considered, working with hooks is nothing more than calling a function. No matter how many hooks you use it's clear where each piece of data is coming from.

```
function Component() {
  const [values, setValue] = useForm();
  return (
    <>
      <input
        value={values.name}
        onChange={(e) => setValue("name", e.target.value)}
      />
      <input
        value={values.password}
```

```
          onChange={(e) =>
            setValue("password", e.target.value)
          }
        />
      </>
    );
}
```

2.33. Model Components as Stateless and Stateful

When I started working with React the predominant mental model was to split components as containers and presentational ones, also known as smart and dumb. The container components are the ones that contain the logic. They hold the state and do the data fetching. The presentational components only accept props, render markup and deal with events.

This mental model was made when people were still figuring out React. It's the equivalent of what the MVC structure is for back-end applications. You have some container component which takes the role of a controller. It deals with all the complexity and passes the data to be rendered to the views. It's not a bad structure, it's generic enough to work in most projects.

But in modern UI applications, there are better ways to model your components. Pulling all your logic inside a few large container components leads to large all-knowing components with too many responsibilities. They become unmanageable as your application grows. Concentrating complexity in a few places is not good for maintainability.

React's philosophy is that each component is its own logical unit. It should deal with the complexity related to it. Because of that, it's better to model components as stateful and stateless. Functionality should be implemented where it belongs naturally. If you end up with a similar component structure - that's fine. What's important is for that to happen because the application fits that model, not because it was forced.

If a list of filters is displayed only in the sidebar, then fetching and displaying them should be that component's responsibility. No need to pull them on the page level together with everything else. On the other hand, forms shouldn't be responsible for submitting their data. They should validate it and show errors but they shouldn't be aware of what happens with the contents afterward, it's not the form's responsibility. When it's successfully submitted it should be its parent that deals with the data.

2.34. Favor Multiple Smaller useEffect Calls to a Single Big One

The `useEffect` hook has been subject to great criticism in the last couple of years. A member of the React core team even called it a mistake. But regardless of how you feel about it and how complex it seems, knowing how to use it properly is crucial to building well-designed applications.

One of the common mistakes people make with useEffect, is to treat it like the old lifecycle methods that React provided and put all functionality inside a single hook call.

```
const [data, setData] = useState(null);
const [windowWidth, setWindowWidth] = useState(
  window.innerWidth
);
const [countdown, setCountdown] = useState(10);
const [filter, setFilter] = useState("popular");

useEffect(() => {
  // Data fetching
  const fetchData = async (filter) => {
    try {
      const response = await axios.get(
        `https://my-api.com/data?filter=${filter}`
      );
      setData(response.data);
    } catch (error) {
      console.error("Error fetching data: ", error);
    }
  };

  // Setting a global event listener
  const handleResize = () => {
    setWindowWidth(window.innerWidth);
  };
```

```
  if (!data) {
    window.addEventListener("resize", handleResize);
  }

  // Call the fetchData function
  fetchData(filter);

  // Starting a countdown when the component mounts
  if (!data) {
    const countdownInterval = setInterval(() => {
      setCountdown((prevCountdown) =>
        prevCountdown > 0 ? prevCountdown - 1 : 0
      );
    }, 1000);
  }

  // Cleanup function
  return () => {
    window.removeEventListener("resize", handleResize);
    clearInterval(countdownInterval); // Stop the countdown when
the component unmounts
  };
}, [filter]); // Run the effect when the component mounts and
`filter` changes
```

This `useEffect` call has three distinct responsibilities - it fetches data and sets state, sets up a global event listener, and a countdown interval. There are multiple reasons why we want to avoid practices like this, but the biggest one is complexity. Handling all these use cases together forces us to think jointly about operations that should have nothing in common in the first place.

Since the whole effect function executes each time, we have to rely on conditional statements to control logic that we want to run only once. To accomplish this, we will need to check for the contents of the `data` object, which may not be the most reliable signal if the component has mounted. Alternatively, we could set up a flag to check if the component has successfully mounted, but this raises the complexity even more.

We should favor multiple smaller `useEffect` calls to resolve all these problems.

```
// Data fetching
useEffect(() => {
  const fetchData = async (filter) => {
    try {
      const response = await axios.get(
        `https://my-api.com/data?filetr=${filter}`
      );
      setData(response.data);
    } catch (error) {
      console.error("Error fetching data: ", error);
    }
  };

  fetchData(filter);
}, [filter]); // Run when the componen mounts and `filter`
changes.

// Setting a global event listener
useEffect(() => {
  const handleResize = () => {
    setWindowWidth(window.innerWidth);
  };

  window.addEventListener("resize", handleResize);

  // Cleanup function
  return () => {
    window.removeEventListener("resize", handleResize);
  };
}, []); // Run only once when the component is mounted

// Starting a countdown when the component mounts
useEffect(() => {
  const countdownInterval = setInterval(() => {
    setCountdown((prevCountdown) =>
      prevCountdown > 0 ? prevCountdown - 1 : 0
    );
```

```
  }, 1000);

  // Cleanup function
  return () => {
    clearInterval(countdownInterval); // Stop the countdown when
the component unmounts
  };
}, []); // Only run the effect when the component mounts
```

The only functionality that depends on the filter is the data fetching, and by splitting the effect calls into multiple smaller ones, we ensure that the other ones don't need to know about it. We remove all the conditional statements, make the hook calls more readable and easier to debug, and for those focused on performance - we reduce the amount of work that our application will do (even though this will rarely be a cause for concern).

2.35. Favor Focused useEffect Calls to Granular Ones

Even though the name of this rule may sound in conflict with the lessons from the previous one, its purpose is entirely different. While some engineers tend to put too much functionality in a `useEffect` call, those who go to the other extreme and put too little are still harming the design of their applications.

Granular effects often lead to chaining calls when one effect triggers a second, the second triggers a third, and so on. This makes the logic of the component hard to follow, leaving you scrolling up and down through the component body, trying to figure out where exactly a piece of data gets set.

```
const [users, setUsers] = useState([]);
const [processedUsers, setProcessedUsers] = useState([]);

// Fetch data
useEffect(() => {
  const fetchUsers = async () => {
    try {
      const response = await fetch(
        "https://some.rest.api/v1"
      );
      const data = await response.json();
      setUsers(data);
    } catch (error) {
      console.error("Error fetching users:", error);
    }
  };

  fetchUsers();
}, []);

// Process data
useEffect(() => {
  const processUsers = () => {
    const mappedUsers = users.map((user) => ({
```

```
      ...user,
      fullName: `${user.name.first} ${user.name.last}`,
    }));

    setProcessedUsers(mappedUsers);
  };

  if (users.length > 0) {
    processUsers();
  }
}, [users]);
```

This is a pattern I've found frequently in React applications - having one effect to fetch the data and another to format it. But I believe that if you're chaining effects that work on the same piece of data, you're splitting your responsibilities too granularly. In this example, we need an extra piece of state for the raw data, so the second `useEffect` can get triggered, even though we won't really use it in the component body.

Instead, have each `useEffect` be focused and encapsulate an entire logical operation. This way, you remove complexity, improve the readability of your hook calls, and make them easier to test.

```
const [processedUsers, setProcessedUsers] = useState([]);

useEffect(() => {
  const fetchAndProcessUsers = async () => {
    try {
      const response = await fetch(
        "https://some.rest.api/v1"
      );
      const users = await response.json();

      const mappedUsers = users.map((user) => ({
        ...user,
        fullName: `${user.name.first} ${user.name.last}`,
      }));
```

```
        setProcessedUsers(mappedUsers);
      } catch (error) {
        console.error(
          "Error fetching and processing users:",
          error
        );
      }
    };

    fetchAndProcessUsers();
  }, []);
```

As a side note, I have to point out that this is one of the main reasons why `useEffect` is claimed to be too complex of a primitive. But I'm convinced that the problems with the hook are not related to its API but how people use it. Having your `useEffect` calls maintain focus without making them too bloated eliminates 90% of the potential design problems you may face.

2.36. Whenever Possible, Don't Fetch Data in useEffect

Perhaps the most common use case for `useEffect` is to fetch data the component needs to render when it mounts on the screen. But, whenever possible, you should initiate the data fetching as early and as high in the application hierarchy as you can. Fetching in `useEffect` leads to the famous waterfall rendering problem in which components further down the chain can't initiate their data fetching until their parent renders fully. This leaves a trail of spinners that doesn't provide the best user experience.

A few years ago, showing loading spinners was all the craze. Marketing people loved showing that something was loading, and it must have shown higher conversion rates than a slow-to-load page. But from a technical perspective, we moved data fetching closer to the components that use that data. And no application wouldn't benefit from its data being fetched all at once.

Now, accomplishing this depends greatly on the environment in which you're using React. Here are some of the common scenarios.

If you are using React Router, you can utilize their `loader` API to fetch data on the route level and then access it deeper into the component sub-tree. Those who use Next.js have the opportunity to preload the data on the server in `getServerSideProps`, or if you're using the latest release with React Server Components, you can fetch on the page component level and pass the value down. Alternatively, there are data fetching libraries like `react-query` that allow you to pre-fetch the data early and only pull it from their in-memory cache in the component.

There are many options to fetch data early nowadays.

However, if your application doesn't focus on improving UX or you cannot do this, I wouldn't go out of my way to implement early data fetching. Many SPAs are dashboards and admin panels running in a corporate environment that doesn't optimize for user retention. I'd argue that, realistically, showing a few

spinners to deliver something faster would be a good trade-off. If your app is already running in production, and you have a full backlog of features, bugs, and improvements, I wouldn't prioritize this immediately. After all, it requires additional complexity that will take time and effort to implement properly.

But if you're just setting up a new project and have complete control over the tech you're running, I'd absolutely consider it.

2.37. Don't Handle Events in useEffect

The `useEffect` hook is meant to be used for handling side effects that arise as a result of the component being rendered. But because of the wording, we sometimes use it for all possible side effects, including ones unrelated to rendering.

I've seen (and implemented) event handlers that set up a flag tracked by an `useEffect` dependency array. Then that effect checks the state of the flag and fires off a request if it's set.

```
const Component = () => {
  const [data, setData] = useState([]);
  const [fetching, setFetching] = useState(false);

  useEffect(() => {
    if (fetching) {
      service.getNextPage().then((nextPageDaata) => {
        setData([...data, ...nextPageDaata]);
        setFetching(false);
      });
    }
  }, [fetching]);

  return (
    <div>
      <Dashboard data={data} />
      <button onClick={() => setFetching(true)}>
        Load More
      </button>
    </div>
  );
};
```

All variations of tthis pattern are complexity hotbeds that can be easily eliminated if we remove the `useEffect` call altogether.

```
const Component = () => {
  const [data, setData] = useState([]);

  const handleLoadMore = () => {
    service.getNextPage().then((nextPageDaata) => {
      setData([...data, ...nextPageDaata]);
    });
  };

  return (
    <div>
      <Dashboard data={data} />
      <button onClick={handleLoadMore}>Load More</button>
    </div>
  );
};
```

In this case, implementing the event handling in an imperative fashion is much easier to read and maintain, than trying to force a declarative implementation. Treat the event as a regular imperative action and execute whatever functionality you need as a direct result of it. Make a call and set the new state when the promise resolves. Not everything has to be handled as a result of rendering.

2.38. Avoid useEffect Calls for Computed Data

Often you will want to display derivative data based on data that you've fetched. For example, the number of items in an array, the average of some values that all items have or an aggregate of a value in a collection.

In these cases, it's tempting to reach for an effect and implement the logic to compute the derived data, using the raw input in the dependency array.

```
const UsersList = ({ users }) => {
  const [userCount, setUserCount] = useState(0);

  useEffect(() => {
    setUserCount(users.length);
  }, [users]);

  return (
    <div>
      <h1>Users List</h1>
      <h2>Total Users: {userCount}</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
};
```

In this particular example, we're adding an extra piece of state, and an effect for something that could be inlined inside the returned JSX instead. We're adding complexity for no real gain. We would be better of just using `users.length` inside the component body, instead of managing it in an extra piece of state.

```
const UsersList = ({ users }) => {
  return (
```

```
    <div>
      <h1>Users List</h1>
      <h2>Total Users: {users.length}</h2>
      <ul>
        {users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
};
```

The same goes for more computationally heavy operations. You would be better off using `usesMemo` for them than handling them in `useEffect`.

2.39. Put The Component Body in a Custom Hook

One principle that I keep stressing in this book is how important it is to create light abstractions around our logic. Another area where we can apply this is for the helper methods we often have to define in the component's body. We need to handle events or run some other kind of business logic, that we can't always inline in the JSX. Together with that we have state definitions, and effects.

```
const ShoppingCart = () => {
  const [cart, setCart] = useState([]);

  useEffect(() => {
    const fetchCartContents = async () => {
      const data = await cartService.getContents();
      setCart(data);
    };

    fetchCartContents();
  }, []);

  const addItemToCart = (item) => {
    setCart((prevCart) => [...prevCart, item]);
    cartService.addItemToCart(item);
  };

  const removeItemFromCart = (itemToRemove) => {
    setCart((prevCart) =>
      prevCart.filter((item) => item !== itemToRemove)
    );
    cartService.removeItemFromCart(itemToRemove);
  };

  const clearCart = () => {
    setCart([]);
    cartService.clearCart();
  };
```

```
  return (
    <div>
      <h1>Shopping Cart</h1>
      {cart.map((item, index) => (
        <div key={index}>
          <p>{item}</p>
          <button onClick={() => removeItemFromCart(item)}>
            Remove
          </button>
        </div>
      ))}
      <button onClick={() => clearCart}>Clear Cart</button>
      <ProductList onClick={addItemToCart} />
    </div>
  );
};
```

It's easy for the component body to grow in size and complexity if we define all its functionality in the component body, but the more we mix business and visualization logic, the harder it will be to maintain. Generally, a React component should stay as close to the idea that it should accept data and return JSX. It should focus on displaying the correct elements depending on the data it gets and firing off the right events. But I don't subscribe to the idea that the component should be aware of the details around how these effects work.

If we extract this functionality to a custom hook, we will make the component oblivious about what set gets set and what APIs get called, making it leaner and easier to test.

```
const ShoppingCart = () => {
  const {
    cart,
    addItemToCart,
    removeItemFromCart,
    clearCart,
  } = useShoppingCart();
```

```
  return (
    <div>
      <h1>Shopping Cart</h1>
      {cart.map((item, index) => (
        <div key={index}>
          <p>{item}</p>
          <button onClick={() => removeItemFromCart(item)}>
            Remove
          </button>
        </div>
      ))}
      <button onClick={clearCart}>Clear Cart</button>
      <ProductList onClick={addItemToCart} />
    </div>
  );
};
```

The component remains unaware that an effect is called when it mounts. It doesn't know if the helper methods modify the state or how they do it. Its sole responsibility is reacting to the data it receives.

An argument against this approach is that the object returned by the custom hook may become large and harder to understand if the component requires a lot of helper methods or pieces of state. But consider the alternative. Is it better if all that functionality is inlined in the component body, making it a couple of hundred lines longer? If the values you need to abstract in the custom hook are that many, this may signal bad design. Instead, it would be best to consider extracting another component that would remove some of those responsibilities.

2.40. Consider Component Composition Before Contexts

The prop-drilling problem is one of the common issues we have to deal with when designing our application's data flow. Often, we will have to pass data from one component to another, which is deeper into the tree. To achieve this, we can explicitly pass it down through every component on the way to its destination, regardless of whether they need the data.

```
export default function App() {
  const [user, setUser] = useState({ name: "John" });
  return (
    <div className="App">
      <Dashboard user={user} />
    </div>
  );
}

function Dashboard({ user }) {
  return (
    <div>
      Dashboard
      <DashboardHeader />
      <DashboardContent user={user} />
    </div>
  );
}

function DashboardHeader() {
  return <header>Dashboard Header</header>;
}

function DashboardContent({ user }) {
  return (
    <main>
      Dashboard Content
      <WelcomeMessage user={user} />
```

```
    </main>
  );
}

function WelcomeMessage({ user }) {
  return <h1>Welcome, {user.name}!</h1>;
}
```

But this creates additional complexity and confusion in the components that pass the data through. They become aware of application details they don't need, and in a large component, it will take a lot of work to figure out what data it truly needs and what it's just passing down the chain. You will have to mock that prop when you're testing, even though the component itself doesn't actually use it.

To avoid this, we're often quick to jump to contexts. They are the most practical way to pass data implicitly between multiple levels in the component tree. But an often overlooked technique is to use component composition. You can leverage the fact that a component accepts children to pass down the prop multiple levels without explicitly defining it everywhere.

```
export default function App() {
  const [user, setUser] = useState({ name: "John" });
  return (
    <div className="App">
      <Dashboard>
        <DashboardHeader />
        <DashboardContent>
          <WelcomeMessage user={user} />
        </DashboardContent>
      </Dashboard>
    </div>
  );
}

function Dashboard({ children }) {
  return (
    <div>
```

```
      Dashboard
      {children}
    </div>
  );
}

function DashboardHeader() {
  return <header>Dashboard Header</header>;
}

function DashboardContent({ children }) {
  return (
    <main>
      Dashboard Content
      {children}
    </main>
  );
}

function WelcomeMessage({ user }) {
  return <h1>Welcome, {user.name}!</h1>;
}
```

There is no difference in how the components rendering the "drilled" prop behave - they accept it as usual. But by allowing the other components to render what is passed to them, we can avoid the design problem without compromising or introducing additional functionality and complexity through a context.

Of course, this approach is not always applicable. Not all components can utilize the `children` prop, and we can't always change a component's definition. But this technique is a useful one to have in your toolbox as a low-complexity solution.

2.41 Extract a Component For Repetitive Logic

The markup that each React component returns is often full of various conditional checks, loops and even domain logic. Our natural reflex when we notice that we're writing the same thing again and again is to follow the DRY principle and extract it into a reusable function.

```
const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          {user.name} -{" "}
          {user.status === "online" ? (
            <span style={{ color: "green" }}>Online</span>
          ) : (
            <span style={{ color: "red" }}>Offline</span>
          )}
        </li>
      ))}
    </ul>
  );
};
```

If you have logic checking the user's status in multiple places in the application, it would be good to extract a utility function and manage the status message in a single place for consistency.

```
const getStatusText = (status) => {
  return status === "online" ? Online : Offline;
};

const getStatusStyles = (status) => {
  return status === "online"
    ? { color: "green" }
    : { color: "red" };
};
```

```
const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          {user.name} -{" "}
          <span style={getStatusStyles(user.status)}>
            {getStatusText(user.status)}
          </span>
        </li>
      ))}
    </ul>
  );
};
```

But this leaves a problem unsolved - we still have to manage the markup. If every component that needs to display the user's status has to write their own `<span>` and call the utility functions, then we haven't hidden that much complexity. We can wrap all this functionality in a single function instead.

```
const displayStatus = (status) => {
  return status === "online" ? (
    <span style={{ color: "green" }}>Online</span>
  ) : (
    <span style={{ color: "red" }}>Offline</span>
  );
};
```

But here we've reached a point where we're using a regular function to return JSX, and every time we have to do something like this, we'd be better off defining a React component instead.

```
const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user) => (
        <li>
```

```
          <UserStatus key={user.id} user={user} />
        </li>
      ))}
    </ul>
  );
};

const UserStatus = ({ user }) => {
  return (
    <div>
      {user.name} -{" "}
      {user.status === "online" ? (
        <span style={{ color: "green" }}>Online</span>
      ) : (
        <span style={{ color: "red" }}>Offline</span>
      )}
    </div>
  );
};
```

With this approach the JSX in the `UserList` component becomes easier to read and we hide the decision-making details around displaying the status inside a custom component. Note, however, that the `UserStatus` component doesn't return a `<li>` HTML element, but a `<div>` instead. This is so we don't couple it to this specific call-site. We don't want to force each component that needs a `UserStatus` to display it in a list.

2.42. Don't Extract Custom Components Without Purpose

Creating components is a powerful way to define your own markup language that makes sense for your company or domain. Having reusable pieces that you can put together quickly is something I recommend every team invest time into. But going overboard with this and creating too many custom components is a form of over-engineering that has more drawbacks than benefits.

```
const Header = ({ user }) => {
  return (
    <div>
      <WelcomeMessage name={user.name} />
      <p>Enjoy your stay.</p>
    </div>
  );
};

const WelcomeMessage = ({ name }) => {
  return <h2>Welcome, {name}!</h2>;
};
```

Consider this example in which we've created an external component to give a domain meaning to a title. It's rare that we will every need to reuse this component in other places, and even if we do it will hardly save us any typing. It doesn't hide any complex functionality that we want to extract from a larger component, and it doesn't remove a long markup that could hurt readability.

When you're extracting a component consider four aspects - can it be reused, is it removing complexity from another component, is it defining its own logic, and is it abstracting a large block of markup. If the answer is no to all of them, then consider if this component would actually lead to worse readability and maintainability.

3. Testing

3.1. Arrange Act Assert

Even though their structure and clarity are often overlooked, tests are code as well. As such, they need to be maintained and modified. When you change the implementation of a component you'll most likely need to update its tests as well.

"Arrage-Act-Assert" is a pattern for formatting unit tests that makes them clearer. It doesn't come from the React world but I've found it to be a great model to follow. It states that each test should be split into three logical parts. Arrange - first, we prepare all the necessary conditions and data. We're setting up mocks and test data that we pass through props to the component. Act - then, we simulate events and interact with the component in some way. Assert - finally, we check whether the component reacts in an expected way The benefit of splitting tests like this is that we make a clear distinction between what is tested, its setup, and verification phases. It also helps us spot common problems like tests that cover too much.

```
test("Checkbox changes the text after click", () => {
  // Arrange
  const checkbox = shallow(
    <Checkbox labelOn="On" labelOff="Off" />
  );
  // Act
  checkbox.find("input").simulate("change");
  // Assert
  expect(checkbox.text()).toEqual("Off");
});
```

This is the simplest possible scenario, simulating an event and validating the component's reaction to it. It's a simplistic approach and you may be wondering how to cover more complex tests. You may want to make multiple actions and make assertions after each one of them.

Those are tests that you would want to break up into smaller ones. Tests still follow the software development best practices, including the single

responsibility principle. You will get more value out of your suite if you have more granular tests rather than few large ones. They will give you better insights into what is breaking. This is another reason why I don't like code coverage as a metric of testing quality but there's a separate rule for that.

3.2. Don’t Test Too Many Things at Once

You’ve done all the preparation for a complex component and now it’s time to write the assertions. You simulate an event and validate its state. It’s tempting to continue with more changes and assertions in the same method. But this can be problematic - tests that do too much should be avoided.

The single responsibility principle states that a function should be doing only one thing. This is in full effect when it comes to testing as well. Tests should validate a single encapsulated flow. Otherwise, when a test fails you wouldn’t know where exactly the problem is.

```
test("Checkbox changes the text after click", () => {
  const checkbox = shallow(
    <Checkbox labelOn="On" labelOff="Off" />
  );
  expect(checkbox.text()).toEqual("On");
  checkbox.find("input").simulate("change");
  expect(checkbox.text()).toEqual("Off");
});
```

This test seems simple and well-written. But you will notice that it validates the default behaviour of the component together with its changed state. Those should be two separate tests.

```
test("Checkbox renders with default props", () => {
  const checkbox = shallow(
    <Checkbox labelOn="On" labelOff="Off" />
  );
  expect(checkbox.text()).toEqual("On");
});
test("Checkbox changes the text after click", () => {
  const checkbox = shallow(
    <Checkbox labelOn="On" labelOff="Off" />
  );
  checkbox.find("input").simulate("change");
```

```
  expect(checkbox.text()).toEqual("On");
});
```

If you follow the “Arrange-Act-Assert” pattern this problem can be caught easily. If the test is a long chain of simulations and assertions you should consider splitting it into smaller independent ones.

When you need to write a test that requires a complex setup, make sure you cover the validity of the simulations and preconditions in previous tests. That allows you to focus each one on a particular interaction. It’s normal and expected for tests to be more verbose than the code they’re testing.

It’s important to prepare a new instance of the component in each test. We want to make sure that no internal state gets carried over from previous executions. Doing that ten times in a file gets lengthy and often the developers don’t pay attention to the quality of testing props. We can be less strict with the software quality rules when it comes to testing but that much duplication can lead to errors.

What I would do in such situations is extract a common function that deals with the component rendering.

```
function createComponent({ labelOn = "On" }) {
  return shallow(<Checkbox labelOn={labelOn} />);
}
```

This gives us a default value for each prop, allowing us to only specify the props that are significant for each test. This gives more clarity and makes the arrangement phase of the test much easier.

3.3. Don’t Rely on Snapshot Tests

The first test written for each component is often a snapshot test. It renders the whole component and saves its markup in a file to validate against. It helps us catch unwanted changes to the component by checking the rendered result.

Snapshot tests sound like a great tool in theory but I avoid using them anymore. Even since I started working with React, I’ve only had one situation in which a snapshot test caught a real problem in a component. A colleague was creating a date incorrectly and it kept defaulting to the current one each day. There wasn’t a regular test to cover it and the snapshot caught the problem.

This example aside, most developers just run the command to update the snapshot when their build fails and proceed with their work. I judge a test by its usefulness and so far snapshots have mostly been wasted time for the teams I’ve worked with.

They shouldn’t be used as a replacement for regular assertions. I’ve seen tests in which the developer has decided to avoid writing explicit checks and used a snapshot instead. Unfortunately, this doesn’t give us reliable information on what was tested and where the problem is. Each change to the component would make the whole suite turn red, making it hard to find which snapshots to update and which have caught an error.

Snapshot tests are fine to use as a sanity check. They can signal to the developer what parts of the application are affected by their change and there may be a situation in which you have overlooked something. Still, they are no replacement for proper testing.

3.4. Test The Happy Path First

The most important thing a test suite should validate is whether a component renders properly and works as expected. You want to validate that it produces the expected result when passed all the props with correct values and types.

The happy path is going to be used 80% of the time and you want to make sure it works well. If we introduce a regression in it, it will have a greater impact than a problematic edge case. In those first few tests, we should validate that the component renders correctly with default props (if it has such) and with passed props. We should validate that all critical elements are on the screen and conditional statements work as expected.

The functionality related to user interactions is also on the happy path. Components change their state or call callback functions passed as props. We should simulate events like clicking and typing to make sure they are reflected in the component's state. We should use mocks to make sure callback functions are called the exact number of times we expect, with the values we expect.

Those tests are the backbone of our suite. They aim to give us confidence that the main logical flow of the component is covered. In conditions with limited time, it's better to have a few tests that validate the happy path for most components than extensive tests for a few of them.

3.5. Test Edge Cases and Errors

Once the happy path is covered we want to turn our focus to unpleasant scenarios in which our components may behave unexpectedly or throw errors. There are two main causes for that - incorrect props or uncaught exceptions.

We want to make sure that our components work well with empty collections or objects with missing properties - when those cases are viable, of course. Pass empty arrays to make sure that the component is not throwing an exception by trying to access an element by index without checking. Pass an object with missing nested properties to test whether we're checking for their existence before using them.

The error handling flow is often overlooked since failures are not that common. There are numerous reasons why an API call could fail, for example. We don't want to be left surprised by our application's behavior in such cases. In a long enough timeframe, exceptions are guaranteed, so they shouldn't be something "exceptional" in our codebase. They should be a firstclass citizen, a normal logical flow that shouldn't be frightening.

Validate that the component can recover when a network request fails. Make sure that this is reflected in its state or it calls the appropriate callback to signal for this. Another common opportunity for errors is parsing a deformed response object. Imagine that a bug sneaks into the production environment of the API we're calling. If we get an invalid response object we want to be able to recover. Depending on your application's domain there may be other causes for errors that you need to cover.

3.6. Focus on Integration Tests

When we test a single component in isolation we write unit tests. They validate the smallest logical unit - its behavior and rendered response. But unit tests are not a sufficient indicator of our application's quality. In some cases, even if they are all green, we could still experience bugs.

This is because the integration between the components may be incorrect. Each one of them may be implemented perfectly but their interactions may not be implemented properly. Those are the kind of problems unit tests can't catch. To make sure that a whole feature works as expected we need to write integration tests that test it as a whole.

In most teams, I would advise starting with integration tests, especially in a startup environment. Integration tests give the most value for the effort that you put into them. They can validate the behavior of a whole widget, a form, or an entire page - a complete logical flow. They won't give you precise feedback on where the problem is but you know that when they are green, your application works as expected.

A failure in a unit test will tell you where exactly the problem lies (if the test is written well) but not what it affects. On the contrary, a "red" integration test will show you what parts of your product are failing but you will have to diagnose where the problem lies. They produce the most value, though. The implementation cost is not high but the benefit of immediately knowing if there is a problem with our application is high.

When pressed with deadlines and limited time, focus on integration tests. Use them to test your application as a black box and go to production with confidence. A tool that you can use for your integration suite is Cypress. You can run a set of tests locally against your local environment to check for unexpected regressions.

Another good idea is to schedule tests that run against your environment each day. You could use your staging environment (if you have one) to validate the complete logical flows. You could schedule tests on production as well but you

need to be more careful with the data you generate there. You need to make sure you won't make a mess, so tread carefully.

3.7. Don’t Test Third Party Libraries

Be careful not to get carried away and test functionality that is beyond your application’s responsibilities. A common mistake is to write tests that validate an external library’s behavior. A question I usually ask at interviews is how to test a component connected to a Redux store.

The library already has an extensive test suite. We can take for granted that it will work as expected if we follow its rules. Still, it’s not uncommon to find tests that go beyond the component’s functionality and test the library’s responsibilities. This is not a good use for our time and not a valuable test to have. Instead, we should be focusing on our component’s integration with Redux.

We should test our components’ integration with external modules without touching the internals. But those boundaries often get blurry when we deal with state management libraries. A better way to approach such testing is to mock the external functionality and validate the component’s behavior up until the point of interfacing with the library.

You can simulate an event and validate that the state management library’s function was called with the proper values in response to it. To continue the Redux example, you can mock a store and validate that the component’s actions are reflected in it but that is not a true representation of your application, just a small fraction of it. Those are your component’s limits, anything beyond them goes in the area of integration tests. In an integration test, you can validate a whole feature, implicitly testing the integration with the library.

Most UI applications use many third-party dependencies. Some of them we rely on explicitly, like the state management ones. Others are small modules that other libraries have dragged in. We can’t and we shouldn’t test all of them, that’s not a good use of our time. The stability of those libraries is handled by their competent maintainers. Our responsibilities are to implement

the business logic of our application. So we should be writing tests that focus on it, it’s the only thing we can control.

3.8. Don’t Focus on Coverage Percentage

Code coverage is a metric that most modern testing tools can generate. It shows what percentage of your code is covered by tests. The more complex tools can even produce a visualization, showing you which branches of your logic have been tested and which aren’t.

Coverage percentage is often held as a badge of honor and used as a requirement when making pull requests. Imposing such arbitrary requirements is not a productive way to think about testing because it sets a minimum threshold that everyone is aiming for.] Code coverage measures quantity, not quality. It sounds like an excellent idea and a sign of high engineering standards. It implies that most of your logic has been tested and your application works as expected. But it’s not a sign that we have tested the right things. We can have good coverage, yet still, experience errors because important functionality isn’t covered.

I’ve worked on projects with more than eighty percent code coverage that still throw errors in production. The test suite consisted mostly of unit tests and that satisfied the requirement. Yet the components’ integrations remained unchecked and that lead to errors. Coverage percentage is just a number that the developer keeps in mind, it’s a target. Too often once we reach it we commit our code without considering the quality of our tests.

The extent to which something should be tested depends on its importance and complexity. A stateless component with a few conditionals in it shouldn’t have the same weight as a stateful one with complex logic. Our focus would be better spent testing the latter.

Too often we need to work with limited time and faced with that scarcity we need to make trade-offs on testing. Some developer teams completely disregard testing if they don’t have the resources to impose a high coverage.

Testing is not all or nothing, each test can be valuable. It’s especially important when dealing with deadlines - we should focus on the most critical parts.

Tests are code that needs to be maintained. Make sure you get value out of the time you put in them.

3.9. Consider Removing Unnecessary Tests

Many software projects, especially larger ones, carry what is known as dead code. Some functionality that is no longer useful (or used at all) but it's still in the product because people are afraid to remove it. No one is sure why it's there in the first place or what damage removing it could cause. This is often the case with tests as well.

There may be a couple of tests on the project you're working on right now that fail randomly. The application is still working fine and restarting the test suite usually fixes the tests. Yet, we still keep those tests in our codebase and even update them when we implement new functionality. With time we learn to ignore them and we pass that attitude to new members of the team.

No one wants to remove them because they may catch a potential problem down the line. But the irony is that everyone dismisses them, so even if they find an actual problem, chances are no one will care. The person unlucky enough to face the "red" test may finally add the keyword to skip them and restart the build. This is how bugs sneak into production.

A failing test should be a sign of a problem. Something in our application doesn't work the way we expect it to. With such inconsistent tests, we don't get any valuable feedback. Instead, they are confusing us and dragging maintenance costs with them. The reasons why those tests fail differ but often it's because of a race condition when testing async logic incorrectly.

I've always followed the principle that fewer well-written tests are better than many unreliable ones. When you find such tests you have two options - removing or reworking them. Before doing anything it's best to gather as much context as possible. Why were they written? What are they validating?

Why are they failing? Then if you don't see value in them, just delete the tests. You've gone so far without them anyway. The alternative is to rework them. That implies that you've found a way to fix the problem and you've found value in keeping the tests.

Both options are good. Leaving the tests in the codebase is the only wrong thing to do. If you can't rework the problematic tests at this moment, just remove them. If you happen to need them in the future you can always find them via source control, fix the problems, and add them again.

3.10. Keep Tests Close to The Implementation

When proper UI engineering was still in its infancy, writing tests was an afterthought. The developer community was still figuring out how this kind of engineering would look at scale so it inherited a lot of practices from other languages.

One such practice was to separate tests from implementation functionality so they don't clutter the heart of the project. This comes from the idea that testing and development are somehow separate and encourages that way of thinking.

A React application may have tens of components. Keeping all of them together will quickly create a mess. An unstructured list of files is so hard to browse that more often we'd use the search capabilities of our IDE to find the file instead. The alternative would be to replicate the structure of the application in the testing folder but that leads to double work each time something in the project is changed.

Keeping tests separate means we must rely on a naming convention instead of colocation. As software projects become larger, the community is moving in a direction in which writing tests becomes a normal part of the development lifecycle. Because of that, it's natural to keep tests closer to the implementation This pattern can be seen in other programming languages like Go.

We should keep the testing file together with the component of functionality they are written for. It makes it easy to see what is tested and what isn't. Each change is going to be contained in a single folder and it will encourage the developers to write tests as they create new components.

3.11. Refactor With Tests

Writing software is not that different from writing a book - it becomes better with each refactor. The version of Tao of React you're reading has undergone three rewrites. In the same way, implementations will be "edited" multiple times before they reach their final state. When you see a well-designed component you can be sure its first implementation was a mess.

Refactoring is a normal part of development. No software architecture is ever final. No component design can take all things into consideration. Often an unexpected change in the business requirements will mean that we need to alter our implementation.

But refactoring is tricky because we aim to make a change to the design of the code without altering its functionality. In the end, we want to have the same result but with a different structure. There are two reasons we might want to refactor something. Too often we do it because the code has become unpleasant to work with. Maybe it was put together in a hurry or it wasn't designed well. The second reason is when the current structure is not made to be modified to such an extent. An example of that is refactoring a component to use a reducer instead of a simple state.

It's tempting to dive right in and make amends. But before we do that, we need to have a way to validate that the behavior is still the same after the refactoring. Even a small change in the structure is enough to introduce a bug.

To avoid this we must make sure that the component has been tested. The most important tests are those that validate its rendered result and side effects like callback invocations. We want to make sure it works well as a black box. Tests that validate its internal logic won't do us much good when we're changing it. If we don't have such a suite our first job should be to write it. Then we can run them in watch mode and start refactoring with confidence.

3.12. Tests Are Not Quality Assurance

It's important to know that tests are not a measure of quality. We write tests so we're confident that our components work as expected. But there's always a chance that our tests were written around our bugs. Most regressions that reach productions have gone past the testing suite. If we don't understand the logic of the feature well, we won't write tests that validate it.

A test suite will only validate that the code works in the way we expect it. They're guard rails but it's our responsibility to put them where they can help. Tests can uncover problems if used correctly. That's the whole philosophy of Test Driven Development. We start by describing the end state in the test. Then we write the implementation that adheres to it. I've caught problems with my implementation writing them this way.

At least for now, I haven't managed to integrate TDD into my workflow. There are many benefits of applying it but it's not how my mind operates. Since I'm not using it personally I won't advocate for it. I just want to highlight that the existence of a test suite is not a sign of great quality.

Regressions are a sign that a problem wasn't understood by the developer. If someone implements incorrect behavior, the tests will validate the incorrect behavior. The bottom line is that bugs can't be caught without a clear understanding of the problem we're trying to solve.

Sadly, this is not a purely technical problem to solve. It's not about automated checks, formatting tools, and code coverage requirements. It's about cultural change. The best way to catch problems is to create a culture in which people care and understand their products. A culture in which code reviews are not just scrolled through but deeply examined. Quality is an organizational effort, not purely a technical one.

4. Performance

4.1. Avoid Premature Optimisation

The larger the application we're working on, the more we need to consider its performance. Such problems are unavoidable and products of a bigger scale have more opportunities for them to surface. But before we invest time and effort in performance optimizations we must make sure there's a reason.

Premature optimization is the root of all evil and following best practices blindly is a waste of time and effort. Improvements put to use without need can backfire and create problems where they didn't exist. Caching and memoizing have benefits for large data sets. Using that practice for light computations will use more memory and increase the load time.

We don't need to micro-optimize every single component. It's important to write code with performance in mind but it shouldn't be the leading factor in our design decisions. Writing components that are easy to maintain and extend is most important. Write readable and well-structured code - such implementations are easy to improve if needed.

When that dreaded day comes and you get feedback that your application is slow, don't rush. Don't memoize everything. Pinpoint the problem first. For example, there's no reason to optimize the rerender count if the bundle size is enormous.

If the application's performance suffers a big hit after a release, there's a big chance that a bug was introduced in it. We can get a better result if we review the latest released code. I've experienced such situations first hand and the problem was fixed with a two-line change.

Most times there isn't a single performance issue to be dealt with. Measure which problems have the greatest impact. Start with the low-hanging fruit so you can release improvements quickly and give a notable improvement to your application. Don't dive into the most complex problems immediately. This improves the user experience and buys you time to solve the harder ones.

4.2. Use the Rules of Laziness

There are two general rules that I keep in mind when it comes to performance. They are aptly named "the rules of laziness" since they're about doing less work. They are general principles to keep in mind in any situation where performance is an important factor. I no longer remember where I learned about them so I can't credit their inventor.

First, if we don't do something we won't have to waste resources on it. If an operation isn't critical for the product or the user at this point in time - don't spend time and memory on it. The best way to improve performance isn't to speed things up - it's to do less. A great example is sending a big bundle of JavaScript code to the browser. Even unused code takes its toll - it still has to be parsed and executed.

Second, if we can do something later, never do it now. At any point in time, we want to be doing the least amount of work necessary. It's common to split our application into small bundles and send the user only the files required for the application to operate. Load files, assets, and data only when you're sure that they're needed to reduce the initial loading time.

4.3. Use Code Splitting

The single biggest factor that will affect your application's performance is the amount of JavaScript you're sending to the browser. We can talk about all kinds of optimizations and improvements but if your application is delivered as a single file, a few megabytes large, it won't matter.

The applications we are working on tend to grow. A successful business will add more features and this will always have an impact on the size of the application. We don't want to build products that become slower and less pleasant to use as they are growing.

There are two costs that we need to keep in mind when we talk about JavaScript size. First, it's the cost of delivery - sending the files to the browser. Second, the cost of parsing and executing that code.

The application can be blazing fast, but chances are no one will find out about this if they can't download it. A common technique that addresses both of the costs we mentioned above is code splitting. In the past, we used to ship a single JS file but the size of modern applications and the advancements like http2 make splitting the application into multiple small files a better choice. The philosophy behind this technique is to allow the user to download the least amount of code required to run the application. If they are opening a single page, we should only send them the code for that page. There is no need for them to download the code related to other functionalities of our app since they may not get to use them at all.

The question that people usually have is which parts of the application to split in separate chunks. A good starting point is to make a separate bundle for each route. This change alone will have a solid impact on your application's performance. Then, look for a non-critical part in each page that you can split. Those can be components that are not visible initially - opened in a separate tab or a modal window or visually heavy components like charts.

4.4. Rerenders - Callbacks, Arrays and Objects

Before we go into the details of render optimizations, I want to point out that the majority of React applications won't suffer a performance impact because of that. Implementing a good code-splitting strategy will have a far greater effect than micro-optimizing components.

With that said, it's good to keep some things in mind when developing components. If many of them need to render too often this may create laggy and unpleasant behavior. When building a dynamic application with many interactions, you want to have good control over what renders and when. The reason why a component may rerender unnecessarily is when it accepts props that are compared by reference - arrays, objects, and functions.

Passing a callback function directly is often cited as a potential performance problem. Each render creates a new function that gets passed down the component chain and triggers rendering even if there's no reason for it.

I am honest in telling you that I've never had performance problems in any application because of callback functions. Perhaps I have never worked on anything on such a large scale, but in reality, most people don't. Don't worry about them unless you've covered everything else and you still need to squeeze out any possible improvement. Focus on making your code clear and readable. In most cases, passing an anonymous function is the right thing to do.

Passing arrays and objects as props fall into the same category of potential performance problems. They are compared by reference so they will trigger a render if passed inline. Again, those will rarely exceed the performance budget of your application but I have a design problem with arrays and objects that are passed directly. If it's a static variable it's much better to define it as a constant somewhere outside the component and use it from there. That allows you to name it so it can communicate its purpose and as a side effect avoid performance problems.

4.5. Memoize Computed Data & Anonymous Functions

As a general rule, the less computation that we do in the UI the better. Whenever possible, get the data you need in the format that you need to display it in and avoid formatting it in the browser. Unfortunately, most times this is not an option. We may not have control over the services that provide it or we may need to structure it in a format acceptable by a library.

When we have to do complex grouping, filtering, or sorting operations on large quantities of data we should consider memoizing it. One option is to use React.memo which will cache the whole component for a given set of props. An alternative is to use the hook useMemo and store the value of a function invocation.

In the first case, the component will rerender only if its props fail a shallow comparison. In the latter, we need to provide a list of depending variables that will trigger the function when changed.

Those are powerful tools that we can put to use when dealing with large datasets but we shouldn’t use them too carelessly. Caching comes with a cost. It introduces some overhead because React needs to do a lookup before rendering or returning the data. That’s why we don't wrap every single component in React.memo. In the same way, we shouldn't wrap every function that maps over an array with useMemo. For smaller data sets that would actually cripple our application’s performance. Performance-related changes applied incorrectly can backfire.

Sorting a collection of a hundred items is better off not being memoized. Caching that will lead to a slower execution than running the functionality directly. This form of caching gives benefits only if used for expensive operations or for components that need to render frequently. The same goes for useCallback. You don't want the optimization to cost more than the code it's optimizing. Put this hook to use only if you’re passing the cached result to an expensive component whose rendering you want to control completely.

Don’t use it to pass an event handler to a button.

But each time you have a problem with rendering and you’re thinking of using caching, consider whether your component structure is not the actual problem. Too often we are trying to fix logical and structural problems with technical solutions. If we don’t want to rerender a component too often it may be better to move it to a different place of the component tree.

In some cases, this is not an option but it’s always worth exploring. Often there is some unrelated piece of state that gets updated and that triggers the whole branch of the React tree. We should look for opportunities to lift expensive components above or at the same level of the tree in which that unrelated state resides. This way we don’t have to implement caching and fret about its impact but alter the hierarchy a bit.

4.6. Improve the Business Logic Implementation

When we're talking about performance improvements we usually focus on the technology. We've talked about the speed of delivery of our application. We've addressed React's specifics that we need to keep in mind. But to build effective products we need to pay attention to an often-overlooked factor - the business logic.

React's interactions with the DOM are fast. Unless we're actively avoiding the best practices we will get sufficient performance. Few software products work on a large enough scale that will push it to its limits. When Concurrent Mode becomes stable it will unlock the door to even more complex UI applications.

But React's speed is only related to rendering. If we write inefficient algorithms our application will be slow regardless of the technology. We can follow the rules to the letter but the users' screens will still block and be dominated by loading spinners. If the logical flow of our application is not designed well enough, it will still have an odd feel even though it's technically sound.

Managing data fetching is the biggest factor because it needs to be communicated to the user - usually done with a loading spinner. But showing loading indicators too often can lead to an unpleasant user experience.

Whenever possible, consider using optimistic updates. If the user's interaction is not critical, immediately show them a successful result even if we haven't got a response from the server. That makes the UI feel responsive and snappy. If the action is not successful we can revert to the previous state and show a message.

Gaining knowledge of the business domain can help us make good trade-offs and implement operations in a more performant way. We can keep data in memory to avoid server round trips and provide instant feedback. We can communicate with the back-end developers and consider caching on complex queries.

Making those decisions requires an understanding of the domain and knowledge of the infrastructure. Of course, we have limited control over the logical flow of a business. Sometimes the best we can do is communicate to the user that we're loading data - show a spinner and hope for the best.

But even if we can't make larger improvements to the user experience we still need to make good decisions when implementing the business logic. We should aim to write efficient algorithms and fetch data in a non-blocking manner.

5. Things to Keep in Mind

There's more to building software than writing code. Our industry is still young and we're figuring things out as we create. They say that software is eating the world but we're still arguing about fundamental principles and we don't have established ways to build products.

We work in an imperfect world where best practices don't always hold up. Things seem straightforward in the books but no plan survives first contact with reality. Features and changes often come with hidden complexity that can ruin a well-designed architecture.

The problems with building software are not always down to logic and creativity. Sometimes they're more art than science. Especially when it comes to software design and architecture.

When you read the principles in this book you will be itching to apply some of them. This chapter will give you more details about doing that and approaching the conversations about software design with your teams. The next points are not specific to React - I've found them to be true regardless of the technology or company.

5.1. Why is Software Design Important?

It's hard to make analogies between the software world and the analog one. As developers, we're not bound by the same rules. Our work is flexible and ever-changing. We rewrite and refactor so often that we overlook the application's structure and design - after all, it will change. We rush to create and build without putting constraints on those processes.

To put things in perspective, imagine that you're building a house. Would you start laying bricks straight away? No, you take measurements, you make preparations, you consider how the rooms will be laid out.

It's the same when we're building software. Thinking about the design of the system or the structure of the codebase makes you go through the whole problem before you start the implementation. We want to think about the problems in advance while we're still on the conceptual level. If we don't, we'll still face them but without any prior preparation. You don't want to find out that your foundations are not sufficient while you're building the second floor.

If a builder gets to work without making sketches and plans of the process you'd think they're insane. But way too often we dive into a technical problem without considering even the basic architectural challenges. Okay, structure and modularity are important, but should we be paying that much attention to naming and API design? Does it matter that much how many props my components take or how I name them?

Let's go back to the house metaphor again. It's not just a matter of physics and calculations - it's about beauty as well. You want to have a nice view from the bedroom, a well-equipped kitchen, and a spacious living room. Maybe you'd like to have a porch where you can spend the evening with a loved one.

You can live in one big rectangular space with everything together but you want a more pleasant living experience. That's why API design and naming are important.

You can put all your components in a single file and flatten the folder structure so everything can be on the same level. It will work, you will be able to build something this way - but it will be neither pleasurable nor productive to work in such an environment There are many successful ways to build software but you want to have a good development experience. You want the people who join your team to stick around and have fun, not to complain about the state of the codebase (at least not too much).

Everyone can solve a problem and leave it behind. But creating an application that is going to be extended and modified requires more thought and care. You want to build a codebase that is flexible, reusable, understandable, testable, and extensible. This doesn't happen by accident, it requires deliberate thought and action.

Is this only important for the developers? No, it's important for the project, it's important for the business. Successful products live longer and throughout their life, they change and evolve. If the application's architecture doesn't support that, product development will suffer. Developers will have to find workarounds or take massive amounts of tech debt for each next feature.

That's why how you write is as important as what you write.

5.2. Avoid Analysis Paralysis

I've stressed the importance of planning and designing but that too should be done in moderation. The things we write today will be refactored or rewritten as the business changes. No architecture is going to be suitable forever, so make decisions based on what you know at the moment and don't overthink it.

When you plan the structure and design you can find yourself engaging in endless discussions that don't seem to have the right answer. Every choice has a downside that will be quickly pointed out to you. If you're trying to find an approach that won't require a compromise you will be left disappointed.

There are no right answers. Even those that seem irrefutably correct now may change in the future. No architecture stands the test of time and our job as builders is to make changes easier to happen. Businesses pivot, market conditions change and ideas get left behind - we can't plan for everything.

When you find yourself paralyzed, weighing all the different options, and not being able to make a call, remember that no decision is going to be right in the long term. A debate on architecture is healthy but when it goes into deep philosophical topics on the nature of software you've crossed the line that gives good returns.

This is not a problem specific to React or UI development. Software projects can become a field of heated discussion. The rules in this book will help you start with a good foundation for your products and unblock you so you can start building.

5.3. Don’t Postpone Architectural Changes

Change is inevitable. The business decides that they want to add another module to the application and you have not foreseen that. You realize that some aspects of the structure are not ideal. Some of your decisions may start hindering you instead of helping you.

When this happens, don’t wait to make the architectural change. Don’t postpone and build on top of shaky foundations. If the complexity of your application is about to rise and you decide you need a complex state management tool - integrate one. If you build the new functionality with your current structure you will probably be left with it.

The more you build on those foundations the harder it would be to alter them in the future. If you can’t make time to change something now you won’t be able to do it in a few months when there’s more functionality to refactor. There are ways to adopt those changes gradually. If you use the module structure described in chapter one, you will be able to integrate a tool like Redux in a single module of your application. Then you can start refactoring the other ones.

Most projects don’t end up in a bad state because of a single critical decision. No, they’re a result of numerous small compromises and "insignificant" mistakes that compound through time. Whenever you need to make a change, consider how much you’re deviating from the initial design. If you go too far from it, chances are it’s not ideal for what you’re building at the moment.

The technical changes are not the hardest part of this effort, it’s communicating the need to do them with your management. It’s important to approach the discussion by showing that this change is in everyone’s best interest. Be prepared to talk about estimates and how that could impact work on features. The longer you wait before making the change the more resources it will take so have the uncomfortable conversation early and start planning.

5.4. Consistency is Most Important

I want to emphasize that no matter what practices you follow, the most important thing is to be consistent and follow them in the whole application.

More problems and confusion arise from inconsistency than anything else. In an ideal world, you want your whole codebase to look as if it was written by a single person. That reduces the size of the mental model that every developer needs to keep in their head.

You learn the rules of the project once and you apply them everywhere. You can focus on writing code and implementing business logic instead of deciphering every colleague's style. The code is a reflection of a person's mind. The more of it we can standardize, the better. Let people show their creativity in the implementation rather than code structure and component design.

After all, we spend more time reading code than writing it. When you start a book by a new author it takes some time to get used to their style. Imagine having to do that three times a day as you're looking for something in the codebase. Make it easier for the team and follow the same principles.

Developers derive most of the information about a project from the codebase. Inconsistencies make that harder.

There isn't a correct naming convention, there isn't a correct code formatting standard. No matter what everyone tells you, there aren't great benefits of one over the other. Most approaches are "good enough", what matters is consistency.

5.5. Make the Back-End Do the Work

If you're responsible for both the front-end and back-end of your application or working in a cross-functional team that spans both layers of the stack, try to make the back-end do as much work for you as possible. Doing heavy computational work in the UI can make the rendering process slow. If your front-end is very dynamic and the browser has to aggregate large quantities of data it will lead to a choppy user experience.

One way to avoid this is to offload some work to the back-end. Consider whether the data you visualize can be pre-aggregated and formatted from the API. This will leave the React application to only decide what and when to render. This is a good option if you're fetching and restructuring data afterward. In this scenario, the back-end can effectively save you a lot of work. But abstain from adding additional API requests just to handle data manipulation. The latency and potential network failures will just change one set of problems for another.

Another symptom of the same problem is if your front-end is overfetching data i.e., loading a large collection from the server, wasting bandwidth, and slowing down the response, only to render a couple of fields from every item in the list. Alternatively, we often have to aggregate data from multiple external sources to gather everything we need to visualize a page. You need to make a request for an article, for example, and then make two more requests to fetch related articles on the same topic and today's most read ones.

This is not only a potential performance problem but also a design one. The front-end is becoming aware of how the data is stored and what parts of the system are responsible. Any change in these endpoints or the data structure must be synced with the UI. Now, if a mobile application also uses these APIs, then the problem exacerbates even more.

To resolve this, you should consider adding an intermediate layer to do the aggregation work for the front-end or strip away unneeded data. In recent years, GraphQL has proven to be a reasonable solution for these two

problems. I abstain from recommending it to teams because running a GraphQL server is not easy. You'll have to deal with schema management, query complexity, and maintain another critical service. But if you are starting to face these problems, then it would be better to handle them in a place better equipped than the front-end.