BUILDING LARGE SCALE WEB APPS

A REACT FIELD GUIDE



ADDY OSMANI

HASSAN DJIRDEH

Building large scale web apps

A React field guide

By Addy Osmani and Hassan Djirdeh

© Addy Osmani and Hassan Djirdeh

Learn tools and techniques to build and maintain large-scale React web applications.

www.largeapps.dev

ISBN: 978-1-304-52088-3

Preface vii

Introduction 9

Managing Software Complexity 15

EXPLAINING COMPLEXITY 17
DETERMINE THE ROOT CAUSES OF COMPLEXITY 19
A PHILOSOPHY OF SOFTWARE DESIGN 20
OUT OF THE TAR PIT 21
SIMPLE MADE EASY 22
NO SILVER BULLET 23
SYSTEM DESIGN AND THE COST OF ARCHITECTURAL COMPLEXITY 25
HOW CAN TEAMS MANAGE COMPLEXITY? 26
THE BEST SOLUTIONS ARE SIMPLE BUT NOT SIMPLISTIC. 27
SOMETIMES, (ESSENTIAL) COMPLEXITY HAS TO LIVE SOMEWHERE 29
WHAT ARE ONGOING CHALLENGES IN MANAGING COMPLEXITY? 30
CONCLUSION 30

Modularity 32

MODULES IN JAVASCRIPT 33
LAZY-LOADING 42
CODE-SPLITTING 47
WRAP UP 52

Performance 53

UNDERSTANDING HOW BROWSERS WORK 54
UNDERSTANDING AND REDUCING THE COST OF JAVASCRIPT 58
OPTIMIZE INTERACTIONS 61
NETWORKING 61
REDUCING THE IMPACT OF THIRD-PARTY DEPENDENCIES 62
RENDERING PATTERNS 64
OPTIMIZING PERCEIVED PERFORMANCE 66
OPTIMIZING PERFORMANCE - THE HIGHLIGHTS 70
PERFORMANCE CULTURE 72

Design Systems 74

CODING STYLE GUIDES 75
DESIGN TOKENS 78
COMPONENT LIBRARIES 83
ACCESSIBILITY 86

PERFORMANCE....87
DOCUMENTATION....88
CASE STUDIES....90
WRAP UP....94

Data Fetching....96

BROWSER APIS AND SIMPLE HTTP CLIENTS....97
MORE SOPHISTICATED DATA-FETCHING LIBRARIES....99
TIPS FOR EFFICIENT DATA-FETCHING....114

State Management....120

MANAGING DATA BETWEEN COMPONENTS....121
PROP DRILLING....124
SIMPLE STATE MANAGEMENT....127
DEDICATED STATE MANAGEMENT LIBRARIES....129
FINAL CONSIDERATIONS....134

Internationalization....137

UTILIZE THIRD-PARTY LOCALIZATION LIBRARIES....140
DYNAMIC LOADING....146
HANDLING PLURALS ACROSS LANGUAGES....148
FORMAT DATES, TIMES, AND NUMBERS....150
CONSIDER RIGHT-TO-LEFT (RTL) LANGUAGES....154
WRAP UP....161

Organizing Code....162

FOLDER AND FILE STRUCTURE....163
NAMING CONVENTIONS....167
BARREL EXPORTS....168
OTHER GOOD PRACTICES....169
WRAP UP....174

Personalization and A/B Testing....175

PERSONALIZATION....176
A/B TESTING....179
FEATURE FLAGS....186
WRAP UP....190

Scalable Web Architecture....191

SCALABILITY....192

CHARACTERISTICS OF A SCALABLE APPLICATION....197
WHERE DO KUBERNETES AND DOCKER FIT IN?....199
WHERE DO COMPANIES LIKE VERCEL AND NETLIFY FIT IN?....202
WRAP UP....203

Testing....205

UNIT TESTS....206
END-TO-END TESTS....212
INTEGRATION TESTS....221
SNAPSHOT TESTS....224
HOW SHOULD WE TEST OUR APP?....228

Tooling....234

VERSION CONTROL (GIT)....235
CONTINUOUS INTEGRATION....237
BUNDLERS....239
LINTING....240
LOGGING AND PERFORMANCE MONITORING....242
WRAP UP....244

Technical Migrations....245

DIFFERENT MIGRATION STRATEGIES....246
MIGRATION STRATEGY....248
CODEMODS....250
THE ROLE OF GENERATIVE AI....255

TypeScript....260

TYPE SAFETY....261
BUILD TOOLS & TYPESCRIPT....263
CONFIGURATION & LINTING....264
REACT + TYPESCRIPT....267
DECLARATION FILES....287
AUTO-GENERATING TYPES FROM AN API....289
MIGRATING AN EXISTING REACT APP TO TYPESCRIPT....296

Conclusions....301

Preface

As web applications become increasingly complex and feature-rich, it becomes important for developers to consider how to build scalable and maintainable systems that don't break under pressure. Building large-scale web applications can be a challenging task, requiring careful planning and implementation to ensure that the application is **scalable**, **maintainable**, and **performs well**.

This book is designed to be a resource that provides a set of tools and techniques to successfully navigate the challenges of building large-scale, maintainable, and scalable **JavaScript web applications using React.** Many of the ideas apply more broadly.

In the first half of the book, we delve into the essentials, adopting a practical, example-led strategy. We start with the basics of understanding **how software complexity can be managed** before addressing key concepts such as **design systems**, **data fetching**, and **state management**, ensuring you understand how to structure and scale your React applications effectively. After closing the first part, we explore other topics like **translation** and **internationalization (i18n)**, and **how code folders and files can be neatly organized**.

In the second half of the book, we dive into more sophisticated areas necessary for full-scale application development. Here, we tackle the challenges of **personalization**, **A/B testing**, **scalable web architecture**, and **caching strategies**. You'll learn how to approach **technical migrations**, ensuring that your application remains cutting-edge and maintainable. Finally, we conclude by discussing the advantages **TypeScript** provides in making React code "safe" and the importance of **testing**.

It's not necessary to read this book from start to finish in one go. Each chapter is designed to discuss and cover a topic in **isolation**, so feel free to jump around to the sections that most interest you or are most relevant to the challenges you're facing in your current projects.

While we've endeavored to create a guide on building large-scale applications, the landscape of web development is vast and ever-evolving. We understand there may be emerging topics or deeper dives into specific areas that you might find valuable for your personal growth or project needs.

If there's a subject you feel we haven't covered in enough depth or a certain topic you'd like us to explore; **we would love your suggestions**. Please let us know by filling out this Google form—https://bit.ly/largeapps-feedback. Your input is crucial in helping us update and expand the book to serve you better.

We'll continuously update and add new content to the book over time. If you're interested in keeping up with these updates, make sure you're subscribed to our Substack newsletter at https://largeapps.substack.com/.

Lastly, the knowledge contained within these pages isn't just theoretical—it's a **practical toolkit** that we hope will empower you to build more robust, efficient, and scalable React applications. With all that said, let's dig in!

- Addy & Hassan

Introduction

With web applications becoming increasingly complex and feature-rich, it's crucial for developers to focus on constructing scalable and maintainable systems. In this book, we'll be exploring some key principles and strategies for building such large applications within the context of JavaScript and, more particularly, React.

First, what do we consider a "large-scale" web application to be?

It would be helpful to first attempt to define what we mean when we refer to a 'large' web application. Many years ago, as part of an experiment, we asked a few intermediate developers to try providing their informal definition of what a large JavaScript application is. One developer suggested 'a JavaScript application with over 100,000 LOC (Lines Of Code)' whilst another suggested 'apps with over 1MB of JavaScript code written in-house'. Whilst valiant (if not scary) suggestions, we consider both of these suggestions as **incorrect** as the size of a codebase does not always correlate to application complexity—those 100,000 LOC could easily represent quite trivial code.

At this moment in time, we think it's easiest to define a large web application as one that is developed by a large team of dozens of engineers or more, or by a small team but evolving over a long time, or both. In summary, large-scale web apps are **non-trivial** applications requiring **significant** developer effort to maintain.

Developing large-scale applications comes with its own set of challenges, thanks to the complexities of managing teams and the impact of group dynamics on software quality over time. To build these applications, it's crucial we pay attention to subtle shifts in software design that can lead to significantly better results. Ultimately, successful development of large applications requires not only good software design but also effective team organization.

How important are the abstractions & fundamentals we work with to this problem?

When building large web applications, **choosing the right abstractions is crucial**. This involves anticipating how others in your field would approach a problem and recognizing when you can't know something. Abstractions can include frameworks, libraries, and other helper layers built on top of a platform. Additionally, embracing the idea that "no abstraction is better than the wrong abstraction" (Malte Ubl) can be important in avoiding the wrong abstractions, which can lead to unnecessary complexity, increased development time, and reduced maintainability.

At the same time, understanding the fundamental foundations beneath the abstractions you use (and their limitations) is valuable. Software engineering involves considering various layers—the core languages, implementation, infrastructure, tools, and people. A surface-level appreciation for these layers can help you build faster, but truly grasping the fundamentals (including topics like O(n) time complexity) can take you even further, especially as languages and frameworks evolve over time.

Does the JavaScript framework or abstractions we use play a role here?

When building large applications, the JavaScript frameworks we use should have strong defaults that throw developers into a "pit of success."

This is for several reasons. First, when developers start using a new framework, they may not be aware of the best practices or optimal settings for certain aspects of large-scale applications (e.g., application performance). Strong defaults help guide developers to make better decisions without requiring them to do extensive research or trial and error. This can save developers time and ensure that the application is following best practices from the start.

Secondly, strong defaults can help balance developer experience and user experience. Developers may prioritize developer experience by making code easy to write and maintain, but this can come at the expense of user experience if the application is slow or bloated. On the other hand,

prioritizing user experience can result in difficult or cumbersome development experiences. Strong defaults can help find a middle ground by providing a good developer experience while also encouraging good performance and user experience.

Continuing on the topic of performance, in several modern web frameworks, such as Next (React), Nuxt (Vue), and Angular, the Chrome team has collaborated on baking in more opinionated components for loading images, fonts, and scripts. Similar features are also present in other solutions, including Astro and Svelte, which can enable teams to concentrate more on refining their application logic. While this does not imply that performance issues have been entirely resolved, it does mean that developers can utilize these built-in features to enhance performance without needing to master every nuance of web performance optimization.

Lastly, having strong defaults can help ensure high-quality applications. When developers are encouraged to use best practices from the start, it can result in fewer issues and less technical debt.

What are considerations of large web applications?

The quality of an application regardless of "size," relates to how well it can satisfy user and business needs without introducing friction. An example may be completing a task with low friction, such as successfully posting a photo or making a purchase. Quality may broadly include having great UX, being usable, accessible, useful, findable, credible, desirable, and valuable. While the degree to which these are critical may vary based on the kind of application, large applications may benefit from keeping these points in mind:

Usable

- **Performance**: speed (fast loading), rendering, interactions, and animations.
- **Task completion**: minimal steps required to complete desired tasks.
- **Stability**: consistent functionality and availability to have pages be usable.

Accessible

- **Accessibility**: have pages be accessed by users with special needs. Considerations may include Screen Reader support, keyboard navigation, and not having issues with low color contrast or other vision deficiencies.

Credible

- **Privacy**: the safe handling of user data with the appropriate level of permission.
- **Security**: the integrity of data and communication on the site.
- **Identity**: characterized by the systems used for authenticating users on the site.
- **Authenticity**: the content on the app is credible.

Findable

- **Discoverability**: the degree to which pages are discoverable by search engines.

Useful

- **Capabilities**: advanced features are similar to those of installed apps.

Desirable

- **Visual experience**: pages are visually and aesthetically appealing.
- **Trustworthiness**: content from trustworthy brands is more desirable.

Valuable

- **Functionality**: content, functions, and use cases that provide value to the user.

In addition to the above points, good quality in a large web application can also include:

1. **Scalability**: The ability to handle increased user traffic and data without a significant impact on performance.

2. **Maintainability**: The ease with which the application can be modified or updated over time without introducing bugs or breaking existing functionality.

3. **Reliability**: The capability to be free from bugs and crashes and provide consistent and accurate results to the user.

Where does this book come in?

The concepts of building and maintaining large-scale web/JavaScript applications are *vast*. Topics from architecture design to performance optimization can be discussed, and this breadth of subject matter can encompass a range of books.

In this book, we've chosen a targeted approach to discuss useful concepts and patterns for managing and developing large-scale React JavaScript applications. In the last decade, React has been perhaps the most influential JavaScript framework in the industry, and having it as the focus point helps streamline how we explain concepts in the book.

With the above in mind, although our primary focus is on React, the majority of topics we cover are applicable to almost any JavaScript framework or library. In every chapter of the book, we discuss an important topic of modern development and use React as the main example to illustrate and demonstrate code.

Descriptive, not prescriptive.

For explaining concepts and patterns, the book follows a **descriptive approach, not prescriptive**. In other words, we don't tell you what specific tools or libraries you have to use to be successful. Rather, we focus on explaining a concept and employ certain libraries or tools to illustrate that concept.

This approach is crucial because we recognize that no single tool or library is the universal solution for a given problem. Instead, we provide

insights and methodologies that allow you to evaluate and choose the best tool or library for your specific needs and context.

With that said, let’s get started! In the next chapter, we'll take a slight detour to first explore how complexity in software development can be identified and managed. Following that, we'll delve into specific topics, patterns, and concepts related to working with large-scale React applications, beginning with the construction of modular React applications.

Managing Software Complexity

One of the most important skills in software engineering is being able to identify *essential complexity* and eliminate *unnecessary complexity*. This can also be worded as minimizing the creation of complexity at all where possible. The longer we do software development, the less tolerance we should have for unnecessary complexity.

"The biggest problem in the development and maintenance of large-scale software systems is complexity–large systems are hard to understand" [Out of the tar pit by Ben Moseley & Peter Marks].



When we build something complex, it can be rewarding when that complexity works. Later down the line, we often regret having created such a monster—the difficulty and time it takes to fix bugs is directly proportional to the resulting code's overall elegance (or lack thereof). Creating "domain experts" for such complexity can lead only to silos within development teams. No single person can understand how the whole system works (much less write down some semantics or any invariants) because of its size and complexity.

Complexity increases when components interact. In a set of complex systems, like at Google, making one part more complex can make all the other parts more complicated as well. Complex requirements can yield complex code, so expecting simple code when complexity is required doesn't make sense. In addition, a person who has been working on the system for a long time will see the code differently from someone new to it.

Complexity, like entropy, tends to increase unless significant effort is applied to remove it from a small area. In the process of doing so, it's not uncommon to move it from one place and move it elsewhere.

In this chapter, we won't discuss any topics specific to React or JavaScript; instead, we will share some thoughts on reasoning about inherent and accidental complexity within the context of software engineering as a whole.

Spoilers:

1. Determine the difference between inherent and accidental complexity. The Cynefin framework can help determine which types of complexity exist in a system.

2. It's often best to tackle the core complexity of a problem (remove, reduce, question) before trying to model it. Avoid doing well that which should not be done at all. Complexity should be looked at holistically across code, architecture, tooling, infrastructure, and even organizationally.

3. Simplicity is not the same thing as simplistic solutions. Don't just consider how to redistribute the burden to another part of the system—think about changing the system so that it does not rely on burdens.

4. Create resilient abstractions. Frameworks, libraries, and patterns can compartmentalize complexity. Be mindful that they meet your needs where flexibility is concerned (e.g., provide defaults, but let me override them if needed).

5. If you work in an organization that has a tendency to complexity, you will inevitably find yourself tasked with solving organizational problems—a Sisyphean task that demands influence to make any progress.

6. Once you've sorted out the organizational complexity and separated the inherent from accidental, you'll be most of the way there. Keep reviewing everything since complexity is a weed that just seems to grow unless constantly tended.

Explaining Complexity

You often have to consider at least two types of complexity: Problem (Inherent) Complexity and Solution (Accidental) Complexity.

Problem Complexity is determined by the nature of your real-world problem. It can include factors that are well outside of your control. If the domain is complex, you must be willing to spend time experimenting with a model before it's fully understood. This is otherwise known as Inherent Complexity—the complexity in solving a particular problem.

The Solution Complexity is about how well you and your team implement a solution. **This part of the complexity is under your control**. The more you know about the problem, the more likely you are to find a solution. You can reduce this type of complexity by spending time working through the problem in various ways. This is otherwise known as Accidental Complexity—the complexity that arises from the implementation of a solution to a problem.

In other words, Inherent Complexity is a fundamental property of the problem itself, while Accidental Complexity is a result of the specific approach taken to solve the problem.

For example, consider the problem of sorting a large dataset. This problem has inherent complexity because many different algorithms can be used to sort the data, and each of these algorithms has its strengths and weaknesses. The specific algorithm chosen to solve the problem will determine the accidental complexity of the solution.

While we often find complex solutions to simple problems, our ideal is to come up with a simple solution that solves all of the issues at hand. We may not be able to avoid a complicated solution for a difficult problem, but the complexity of our solution should never exceed that required by the original problem.

Example of accidental complexity: multiple solutions for the same problem.

We create new solutions to existing problems all the time for a variety of reasons.

1. Inadvertently (since the team is unaware of existing solutions).
2. Because the existing solution does not cover the use case sufficiently.
3. Because it introduces a dependency on another team's work that is outside of our organization's control and agility.

It is often difficult to determine whether or not it's better to reinvent the wheel or adopt an external dependency. But one thing is obvious: both options increase system complexity to different degrees.

Each option may appear to be simpler and more straightforward than the other in the short term. Because we don't have good metrics to measure system complexity, it is not clear how to factor in the increase of such when making decisions.

Inherited complexity

For completeness, it is also worth distinguishing complexity that is inherited vs. purely accidental from a greenfield project. Inherited complexity is often closely related to legacy code, which is code that has been in use for a long time and has been built upon and modified by

many different people over the years. Aggregate accidental complexity over time can affect it.

As a result of this constant change and evolution, legacy code can often become very complex, with many different layers and dependencies that can make it difficult to understand and modify. This complexity can be a major challenge for software developers, as it can make it difficult to add new features or make changes to the system without introducing errors or breaking existing functionality.

Determine the root causes of complexity

The complexity of a system can stem from a number of inherent and accidental factors, including:

- The amount of things we have to consider.
- Interdependencies between different elements of the system.
- Mismatches between the product and the code used to build it.

There has been a wealth of literature and models written about how to navigate this kind of complexity. We'll briefly provide an overview of how some of these can be helpful tools in your complexity toolkit.

The Cynefin framework

The Cynefin framework helps people understand and manage complex systems. It was developed by IBM researchers, including Dave Snowden, to provide a way of identifying different types of complexities in any given system. It is based on the idea that there are five different domains of complexity: simple, complicated, complex, chaotic, and disorder. Each of these domains has its own characteristics and requires a different approach to decision-making and problem-solving.

In the clear (simple) domain, the relationships between the elements of the system are clear and well-defined. This makes it relatively easy to predict the outcomes of our actions, and we can use best practices and expert knowledge to make decisions.

In the complicated domain, the relationships between the elements of the system are more complex, but they are still knowable and

predictable. In this domain, we can use analysis and expertise to make decisions, but we also need to take into account the specific context of the situation.

In the complex domain, the relationships between the elements of the system are not fully known or understood, and the outcomes of our actions are difficult to predict. In this domain, we need to use a more exploratory and iterative approach to decision-making, and we need to be prepared to learn from our experiences and adapt as we go.

In the chaotic domain, the relationships between the elements of the system are highly unpredictable, and it is difficult to know what will happen as a result of our actions. In this domain, we need to act quickly and decisively to stabilize the situation, and we need to be prepared to make decisions without all the information we would ideally like to have.

In the disorder domain, it is not clear which of the other domains the system belongs to, and we need to take steps to create order and clarity before we can make effective decisions.

Overall, the Cynefin framework provides a useful tool for understanding the complexity of a system and for determining the appropriate approach to decision-making and problem-solving in that context.

A Philosophy of Software Design

The book "A Philosophy of Software Design" by John Ousterhout identifies **dependency** and **obscurity** as the main sources of complexity in software design.

Dependency occurs when one piece of code cannot be understood or modified in isolation and must be considered in relation to other parts of the system. Obscurity arises when important information is not obvious, either due to a lack of clarity and documentation at the time of development or because the design becomes so large and complex that it is difficult to keep track of everything. Complexity also grows over time and with the size of the design and is exacerbated by the iterative and extensible nature of the design process.

Complexity can be recognized at both a high level and a low level. At a high level, it manifests in decreased velocity or agility, inefficiency,

increased instability and uncertainty, and overall developer discontent and frustration. At a low level, it manifests in change amplification, cognitive load, and unknown unknowns.

- **Change amplification**: A seemingly small change can require editing multiple files.
- **Cognitive load**: The volume of knowledge required to complete a task is reflected in the time it takes to learn that knowledge and, subsequently, create software.
- **Unknown unknowns**: The code that needs to be changed or information a programmer needs to successfully complete a task.

One should eliminate complexity when possible (i.e., minimize complexity); otherwise, encapsulate complexity effectively.

Minimizing complexity

Minimizing complexity involves writing software in a way that is easy to understand and improve. In the book, we describe the nature of certain issues and present techniques that can be used during the development process to address them, such as high-level design principles and red flags to watch out for.

Encapsulating complexity

In the next chapter, we'll discuss the use of modularity in software design, arguing that it can help to decompose systems into manageable components. Modularity allows for an incremental and extensible approach to design and can replace global complexity with local complexity. However, modularity also introduces dependencies through the interfaces between modules, which adds complexity. New modules should only be introduced if the benefits outweigh the added complexity.

Out of the Tar Pit

In the paper "Out of the Tar Pit," Ben Moseley and Peter Marks define complexity as something that makes a system hard to understand. The authors argue that complexity is the root cause of most problems in

software and identify state and control as the main causes of complexity. The authors note that the behavior of a system in one state tells us nothing about its behavior in another state, and this can make it difficult to understand and reason about a system. Additionally, control, or the order in which things happen, can add to the complexity of a system.

The paper also discusses other contributing factors to complexity, such as sheer code volume and the fact that complexity can breed more complexity. To address these issues, the authors propose several remedies for reducing complexity in software development. These include using functional programming languages, which can help tame the complexity of state, and separating state and logic to make systems easier to understand and reason about. The authors also recommend keeping the amount of code needed to solve a problem as small as possible and using abstraction to hide unnecessary details.

To deal with complexity in the real world, the authors recommend avoiding accidental complexity whenever possible and separating complexity from the pure logic of the system. This involves splitting the logic from the complexity so that the system can still function correctly even if the "accidental but useful" parts are removed. This approach can help make our code easier to understand and reason about and can improve the overall quality of our software.

A good summary of this paper is also available.

Simple Made Easy

"Simplicity is measured by how interconnected a program is. Whether or not it's interleaved is something that anyone can see. In other words, it's objective." [Simple Made Easy, a blog post by Paul Cook on Rick Hickey's talk of the same name].

In the now well-known talk "Simple Made Easy," Rich Hickey, creator of the Clojure programming language, discusses the importance of simplicity in the development of systems. He argues that simplicity is often mistaken for "easy," but the two are not the same. While "easy" is subjective and refers to the familiarity or ease of access to something, "simple" is objective and refers to the level of interconnectedness or complexity in a system.

From the talk Simple Made Easy:

- Choose simple constructs over complexity-generating constructs (it's the artifacts, not the authoring).
- Create abstractions with simplicity as a basis.
- Simplify the problem space before you start.
- Simplicity often means making more things, not fewer.
- Reap the benefits!

According to Hickey, our primary concern when choosing tools and languages should not be their ease of use but rather the resulting complexity of the systems we build with them. While it is natural to prefer easy tools, we should also consider the long-term impact on the ease of changing and maintaining the system. Easy tools may not always result in complexity, but we should not prioritize ease of use above all else. Instead, we should focus on the overall simplicity of the resulting system.

Hickey argues that simplicity is essential in order to effectively analyze and change a system over time, as complexity can hinder our ability to do so. He also emphasizes the importance of minimizing accidental complexity, which is the complexity introduced by the tools and constructs used to build systems.

Hickey suggests using simple constructs, applying the correct level of abstraction when designing components, and using functional composition to construct systems in order to reduce complexity. Hickey suggests that to be efficient, we should focus on both simplicity and ease, with a higher emphasis on simplicity.

No Silver Bullet

In the paper "No Silver Bullet," Fred Brooks argues that there is no single development, in either technology or management technique, that by itself promises even one order of magnitude improvement in productivity, reliability, or simplicity. He identifies four essential difficulties that make building software hard: complexity, conformity, changeability, and invisibility.

Brooks argues that complexity is the most difficult of these four properties and that it is caused by the interactions between the parts of a

system. He suggests that the use of abstraction, modularity, and hierarchy can help to reduce complexity by allowing developers to focus on a smaller part of the system at a time.

Problems resulting from complexity:

- difficult team communication
- product flaws; cost overruns; schedule delays
- personnel turnover (loss of knowledge)
- unenumerated states (lots of them)
- lack of extensibility (complexity of structure)
- unanticipated states (security loopholes)
- the project overview is difficult

Complexity also comes from the numerous and tight relationships between heterogeneous software artifacts such as specs, docs, code, test cases, and so forth.

In software development, the conceptual components of the task often require the most time and effort. To address this, Brooks offers three recommendations that remain relevant today: reuse, incremental development, and investing in your software developers. It's worth noting that these recommendations were made nearly 30 years ago, showing their lasting value in the field.

Additional literature on complexity

While we are talking about programming complexity here, there are many more specific kinds of complexity in software engineering, which include:

- Time complexity (O(log(n)), etc.)
- Cyclomatic Complexity
- Computational complexity
- Kolmogorov complexity

Understanding the impact of your decisions on the business can also help to minimize complexity. It may be tempting to build something that is highly reusable or generic, but from a business perspective, it may not add any value.

System design and the cost of architectural complexity

Many modern systems have become so large and complex that they defy a complete understanding by any one individual or team. A 2013 paper, System design and the cost of architectural complexity published by MIT, explores the link between software architectural complexity and various cost drivers, including productivity, defect density, and staff turnover, and highlights the potential financial benefits of refactoring efforts aimed at improving software architecture.

The role of architectural patterns in controlling complexity

Architectural patterns, including hierarchies, modules, and abstraction layers, play a critical role in managing the complexity of large-scale software systems. These patterns allow for the organization and separation of system components, making it easier for distributed teams to work independently while still contributing to a coherent whole. This organization also facilitates system evolution and maintainability, enabling development teams to better respond to changing requirements and technologies.

The link between architectural complexity and cost

In the paper, a study conducted within a successful software firm measured architectural complexity across eight versions of their product using techniques developed by MacCormack, Baldwin, and Rusnak.

The results of the study, which were widely discussed on Hacker News, revealed that increased architectural complexity could account for 50% drops in productivity, three-fold increases in defect density, and order-of-magnitude increases in staff turnover. These findings resonate with many software developers who have experienced the challenges of working on projects with convoluted codebases and patchwork architectures.

The financial impact of architectural complexity and the value of refactoring

With the techniques developed in the study, organizations can estimate the financial cost of architectural complexity by assigning monetary values to decreased productivity, increased defect density, and higher staff turnover. This information enables firms to more accurately assess the potential dollar-value benefits of refactoring efforts aimed at improving software architecture.

In many cases, investing in refactoring can lead to significant long-term savings by reducing the costs associated with complexity. Improved architecture can enhance productivity, decrease the occurrence of defects, and reduce staff turnover due to frustration. Management teams that recognize these potential benefits are better positioned to make informed decisions about allocating resources to refactoring initiatives.

The importance of balancing speed and quality in software development

The aggressive and rapid nature of the software industry often pressures development teams to prioritize immediate results over long-term stability. However, as many contributors to the Hacker News thread pointed out, the adage "going slower today means we can go faster tomorrow" highlights the importance of carefully considering the long-term implications of architectural decisions. By slowing down and focusing on quality, teams can create systems that are more maintainable and scalable, ultimately resulting in faster and more efficient development cycles.

How can teams manage complexity?

As software systems continue to grow in size and complexity, developers have found that **breaking up large systems into smaller, modular components can be an effective way to manage this complexity**. By using modularity, well-defined interfaces, and tiered architectures, developers can create higher-level modules that are composed of smaller, lower-level components. This allows for a high degree of creative autonomy and flexibility while still imposing the necessary constraints to

ensure that the various components of the system work together effectively.

However, this approach also introduces additional "accidental complexity" in the form of the overhead associated with interfaces and abstractions. This additional complexity is necessary in order to break up the essential complexity of the system into smaller bits that can be managed by smaller, autonomous teams of developers. This allows for better scaling of development teams, but it also increases the overall complexity of the system.

One advantage of this approach is that it allows for the consolidation of common requirements across different systems, which can be achieved through the use of service-based or microservices-oriented architectures.

Additionally, the use of open-source software can help to amortize the costs of developing and maintaining these modular components across multiple organizations.

Modularity, well-defined interfaces, and tiered architectures allow for the creation of larger, more complex systems without sacrificing the autonomy and creativity of individual developers. By breaking up large systems into modular components and imposing the necessary constraints, developers can maintain their creative autonomy while still producing high-quality, scalable software.

The best solutions are simple but not simplistic.

Simplicity makes code easier to write, debug, and maintain. It makes products easier to use. However, it's important to keep in mind simplicity is not the same as being simplistic. Simplicity is about removing unnecessary complexity and focusing on what really matters. A simple solution is easy to understand but not necessarily easy to implement.

Simplicity is about getting to the point quickly without sacrificing accuracy or completeness so that you can focus on other tasks that are more valuable than additional editing or research. The goal is to remove anything that is unnecessary or confusing. When we say "remove," we mean taking things out of the user's way (whether that user is your

customer or another developer). It would be remiss to talk about simplicity without noting Occam's razor.

Occam's Razor

The principle of Occam's razor states that "entities should not be multiplied without necessity." In other words, when we have multiple possible solutions to a problem, we should choose the simplest option with the fewest assumptions and moving parts.

This razor is often mistaken to strictly advocate simplicity, but Einstein's quote, "Everything should be made as simple as possible, but no simpler." points to a more nuanced recommendation. Occam's razor is useful in making rapid decisions and establishing truths, especially when evidence is unavailable or limited. It works best as a mental model for initial conclusions—before all the facts are known.

Let's consider how this principle can be applied to software engineering. In web development, we might be faced with multiple approaches to solving a problem. Let's say we need to quickly build a web application that allows users to create and manage their own profiles. We could consider two different approaches:

1. Implement the profile feature using a custom solution that we develop ourselves. This approach would require a lot of work, as we would need to design and implement the database schema, user authentication, and all the necessary functions for creating and managing profiles.

2. Implement the profile feature using an existing user management system, such as OAuth or a third-party option like Firebase Authentication. This approach would require less work, as we would only need to implement the necessary functions to connect to the existing system and integrate it with our website. It does, however, have other trade-offs.

In this situation, **Occam's razor suggests that the second approach is the better one**, as it requires fewer assumptions and has fewer moving parts. It might not be the absolute simplest solution (there could be other existing systems that are even easier to use), but it's the simplest solution that is reasonable given the current state of the project.

Occam's razor is a problem-solving principle, not a theorem. It's important to remember that simplicity is not always the most important factor in choosing between solutions. There may be times when other considerations, such as effectiveness or performance, outweigh simplicity. It's important to consider all factors and make an informed decision based on the specific situation and needs of the project.

Sometimes, (essential) complexity has to live somewhere

"There is an essential complexity to all meaningful software. We can never eliminate that complexity; we can only control it." [Grady Booch]

We also want to cover the other side of this coin. "Essential Complexity," complexity that is inherent to the problem we're trying to solve can sometimes not just simply be "removed" or "simplified." It has to live somewhere. This is the opposite of accidental complexity, which doesn't have to exist.

Focusing on simplicity for simplicity's sake can be a dangerous trap because it's sometimes impossible to remove complexity from code. In such cases, the only way is to move the complex parts around in such a way that less-experienced programmers of your system will find them easier—not harder—to use. There's a great quote from Fred Hebert that says:

"Complexity has to live somewhere. If you embrace it, give it the place it deserves, design your system and organization knowing it exists, and focus on adapting, it might just become a strength." [Fred Hebert]

Complexity has an unfortunate tendency to spread beyond the places where we originally introduced it. If you are lucky, though, that essential complexity will only exist in well-defined locations: abstractions in your code base, your documentation, and training materials for new engineers on the team.

You give essential complexity its rightful space without trying to hide all of it. You create ways to manage it, but you know where and when you need more complex approaches.

What are ongoing challenges in managing complexity?

As software systems continue to grow in size and complexity, managing their development and maintenance has become an increasingly challenging task. Developers have traditionally relied on a variety of tools and techniques, such as module systems, abstraction, open-source, and distributed architectures, to manage this complexity. However, these approaches also introduce additional "Accidental Complexity" that can be difficult to manage.

One major challenge is the increasing reliance on third-party modules, which can add complexity to a system without providing corresponding benefits. Additionally, the use of microservices and distributed architectures can make it difficult to manage and coordinate the various components of a system. As a result, many organizations are turning to DevOps and SRE roles to help keep their software systems running smoothly and prevent costly outages.

Despite these challenges, many developers are optimistic that new tools and techniques will continue to emerge that can help manage accidental complexity and enable the creation of even larger and more complex systems. However, it is important to recognize that essential complexity will always increase and that managing this complexity will remain a key challenge for the foreseeable future.

Conclusion

In conclusion, managing software complexity is essential for creating effective and efficient systems. By understanding the difference between inherent and accidental complexity and using tools like the Cynefin framework, software teams can attempt to tackle complexity in their systems effectively.

As software developers, we strive to create high-quality code that is clean, robust, and reliable. However, the inherent complexity of software development often makes this more of an art than a science. We can try to minimize the impact of this complexity by thoroughly understanding the problem we are trying to solve and by testing and refining our code, but ultimately, we may never be able to fully eliminate it. This lack of control can be frustrating, and it leaves us wondering if there is a better way to approach the problem.

It is important to focus on removing unnecessary complexity and finding ways to simplify the system rather than simply redistributing the burden to another part of the system. Additionally, it is crucial to constantly review and tend to the system to prevent the growth of unnecessary complexity.

From the next chapter onwards, we'll focus on and discuss specific topics on identifying and managing complexity in large React JavaScript applications, and we'll begin with a discussion on **Modularity**.

Additional reading

- A Philosophy of Software Design — John K. Ousterhout
- Complexity and Strategy — Terry Crowley
- Complexity and Software Engineering — Matthew Sackman
- Summary of "Out of the Tar Pit" — Kyle M. Douglass
- No Silver Bullet — Frederick P. Brooks, Jr.
- There's still no silver bullet — Sean McBride
- Simple Made Easy — Rich Hickey

Modularity

“The secret to building large apps is never to build large apps. Break your applications into small pieces. Then, assemble those testable, bite-sized pieces into your big application.” [Justin Meyer]

One of the key principles of building large JavaScript applications is to modularize and componentize code. This can be described as **dividing an application into small, independent modules or components that can be developed and tested independently.**



Modularity (and componentization) makes our code more reusable by allowing us to share modules and components between different projects and teams. This can save time and reduce costs by avoiding the need to reinvent the wheel for every new project. In addition, by making it easier to manage and maintain our application, we also reduce the likelihood of bugs and make it easier to add new features in the future.

The best way to build a modular application is to start with small, self-contained pieces of functionality that we can test and debug independently. We can write that code as a module and make sure it works as expected before adding more modules or components.

Modules in JavaScript

JavaScript modules allow us to break up code into separate files that can export and import functionality. The native modules syntax provides:

- The export declaration: for exporting anything - functions, objects, primitives, etc.
- The import declaration: for importing from other modules.

Modules encourage reusability and maintainability. We can write a module once and reuse it across different parts of our application and in different projects. As a result, updates to a module won't require changes across the entire codebase. Modules also enable encapsulation. By only exporting the functionality we need publicly, we can hide internal implementation details in private module scopes.

For example, we can have a UI module that exports reusable UI components:

ui.js

```
export function Button({text}) {
  // button component
}

export function Header({title}) {
  // header component
}
```

And import just the parts we need in other files:

page.js

```
import { Header } from './ui.js';

function Page() {
  return <Header title="My Page"/>;
}
```

As our JavaScript applications grow in size and complexity, the importance of modularity becomes even more pronounced. Modules allow for organized, reusable, and maintainable codebases.

With this brief understanding of JavaScript modules under our belt, let's now see how the concept of modules translates when working within a React application.

Componentization in React

In the context of React, modules are often applied in a pattern called "componentization." React, by its nature, encourages developers to think in terms of **components**. Each component represents a distinct piece of the UI and, when built correctly, can be reused across different parts of an application, much like how modules allow us to reuse code.

The beauty of componentization is that it aligns with the modular approach we discussed. For instance, we might have a `Button` component in one file and a `Header` component in another. These components, much like JavaScript modules, can be imported and used wherever needed in our React application. This ensures a consistent look and behavior throughout, while also centralizing the logic and state management for each component.

With a growing React application, the need arises to organize components in a scalable and maintainable manner. This is where the concepts of component libraries, atomic design, or even domain-driven design might come into play. We discuss some of these concepts throughout the book, but for now, we'll delve deeper into some common

and important strategies for approaching componentization in a large-scale React application:

Identify reusable components

The first step in componentizing our React application is to identify reusable components.

Good opportunities for componentization include:

- Repeated elements like buttons, menus, and cards.
- Sections of a page like headers, content areas, and footers.
- Logical chunks of functionality.

By identifying reusable components, we can create a library of components that can be reused throughout our application and even in other projects. This can save us a lot of time and effort in development and testing.

Here's an example of a `Post` component that contains functionality for a post made by a certain author on a social network site. This includes information like the author's name and the post's title, text, date, and other post details.

A post element that contains all the functionality in one component

```
function Post({ post }) {
  return (
    <div>
      <img
        src={post.profileUrl}
        alt={`${post.author}'s profile`}
      />

      <h1>{post.title}</h1>

      <p>{post.text}</p>

      <div>Author: {post.author}</div>

      <div>Date: {post.date}</div>
```

```
        <p>{`${post.numLikes} likes`}</p>

        <p>{`${post.numComments} comments`}</p>

        <p>{`${post.numShares} shares`}</p>

        <button>Like</button>
        <button>Share</button>
        <button>Comment</button>
      </div>
    )
}

export default Post;
```

In this example, we're rendering all the content of the post element in a single component. The `post` prop is passed to the component as a parameter, and we're using it to display the title, text, author, date, and other information of the post. We're also including buttons to like, share, and comment on the post.



Figure 3-1. Post component

The approach of having all the UI of a certain element be kept within one component can be simple and quick to implement, but it can also make

the component more difficult to maintain and test as the codebase grows. Modularity and componentization can help with maintaining and testing, as well as making the component more reusable and flexible.

Divide your application into smaller components

Breaking our application down into smaller, more manageable components is a key aspect of componentization. Instead of building large, monolithic components, we can instead divide our application into smaller components that are easier to develop, test, and maintain.

Smaller components are also more flexible and can be reused in different contexts, making our application more adaptable and scalable.

For example, we can break the `Post` component into smaller components as follows:

- PostHeader
- PostContent
- PostFooter

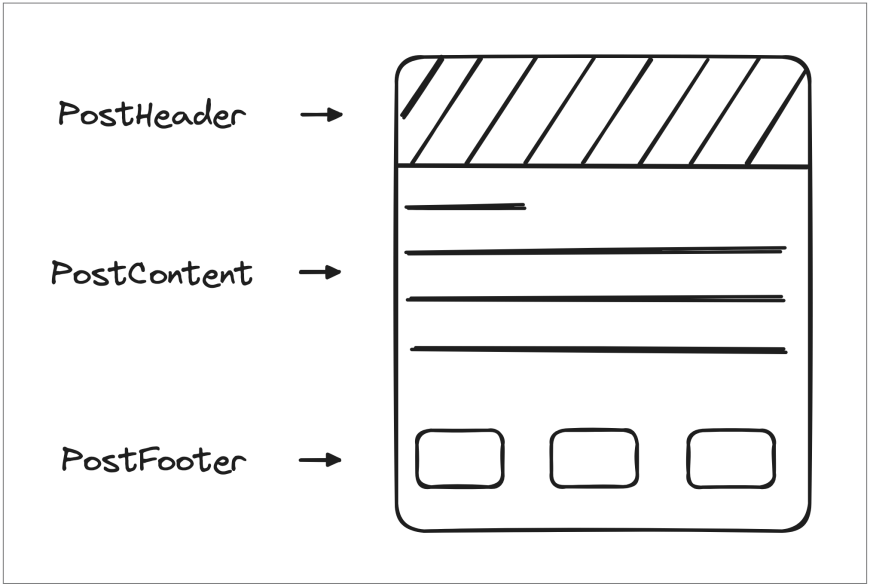


Figure 3-2. PostHeader, PostContent, and PostFooter components

PostHeader

The `PostHeader` component can hold the responsibility for displaying the header of the post. It can take props for the author's name, profile picture, and timestamp.

The `PostHeader` component

```
function PostHeader({
  authorName,
  profileUrl,
  timestamp
}) {
  return (
    <div>
      <img
        src={profileUrl}
        alt={`${authorName}'s profile`}
      />
      <p>{authorName}</p>
      <p>{timestamp}</p>
    </div>
  );
}

export default PostHeader;
```

PostContent

The `PostContent` component will be responsible for displaying the main content of the post. It can take a prop for the post's text and perhaps also an array of media elements (e.g., images, videos).

The `PostContent` component

```
function PostContent({ text, media }) {
  return (
    <div>
      <p>{text}</p>
      {media.map((element, index) => (
        <img
          key={index}
```

```
            src={element.url}
            alt={element.alt}
          />
        ))}
      </div>
    );
}

export default PostContent
```

PostFooter

The `PostFooter` component would be responsible for displaying the footer of the post and can take props for the number of likes, comments, and shares.

The `PostFooter` component

```
function PostFooter({
  numLikes,
  numComments,
  numShares,
}) {
  return (
    <div>
      <p>{`${numLikes} likes`}</p>
      <p>{`${numComments} comments`}</p>
      <p>{`${numShares} shares`}</p>

      <button>Like</button>
      <button>Share</button>
      <button>Comment</button>
    </div>

  );
}

export default PostFooter
```

Post

The `Post` component will finally be responsible for combining all of these smaller components into a cohesive post element. It can take props for the post's author, content, and footer and pass these props down to each of the child components that require them.

The parent `Post` component

```
function Post({
  author,
  content,
  footer,
}) {
  return (
    <div>
      <PostHeader
        authorName={author.name}
        profileUrl={author.profileUrl}
        timestamp={author.timestamp}
      />

      <PostContent
        text={content.text}
        media={content.media}
      />

      <PostFooter
        numLikes={footer.numLikes}
        numComments={footer.numComments}
        numShares={footer.numShares}
      />
    </div>
  );
}

export default Post
```

By dividing the post into smaller components like this, we make the code more modular and easier to maintain. We can also reuse these smaller components in other parts of the application, making it more scalable and adaptable.

While it's tempting to make everything a component, it's essential to strike a balance. Too granular, and you might end up with a codebase that's hard to navigate. Too broad, and you miss out on the benefits listed above. How we decide to break down components can be based on multiple factors:

- **Reusability**: Do we want a specific UI element or functionality to be repeated in various parts of the application? If so, it might be a good candidate to be turned into its own component.

- **Simplicity and readability**: How readable is the component's code? Would it be easier to read and understand the code if the component was broken down into smaller sub-components with their own focused responsibilities?

- **Improved testability**: Can the component be tested more effectively when it's smaller and has a focused responsibility? Smaller components often have less internal state and fewer side effects, making them easier to isolate in tests and ensuring that each piece of functionality works as intended.

- **Performance considerations**: Would breaking down the component optimize rendering or reduce unnecessary operations? In frameworks like React, smaller components can sometimes lead to fewer re-renders with the capability to memoize components and computations. This could sometimes improve the app's overall performance.

Implement a Design System

A design system is a collection of reusable components, guidelines, and assets that help teams build cohesive products. Many popular design systems exist today, such as Material by Google, Polaris by Shopify, Human Interface Guidelines by Apple, Fluent Design System by Microsoft, and more.

Building a design system can help us standardize the design and development of components in our React application. Furthermore, using an existing open-source design system, like Material by Google, can expedite the development process by providing a well-documented set of

components and patterns. This allows us to focus on application-specific logic and functionality.

We go into more detail discussing how reusable components play a role in building and maintaining design systems in the upcoming chapter—**Design Systems**.

Lazy-loading

Lazy-loading is a technique for loading resources only when they are needed. This can be useful for improving the performance of an application by reducing the amount of resources that need to be loaded initially.

In React, we can optimize the performance and responsiveness of applications by judiciously loading components only when they are required (i.e., lazily). We're able to achieve this with the help of React's lazy function and Suspense component.

- **lazy**: a function that allows us to load components on demand.
- **Suspense**: a component that can be used to display a fallback component while the lazy component is being loaded.

Let's go through an example of how we can use lazy-loading in a React application to load the `Post` component we created earlier. We'll first start with a basic example of importing the `Post` component statically, and then show how to use dynamic imports with `React.lazy()` to load the component dynamically.

Static import

With a standard static import, the `Post` component is imported at the top of the file with the import declaration:

Static import of the `Post` component

```
import React from 'react';

// importing the Post component
import Post from './components/Post';
```

```
function App() {
  return (
    <div>
      <Post />
    </div>
  );
}

export default App;
```

If the `Post` component is large and takes a long time to load, this can impact the initial loading time of its parent `App` component, especially if the `Post` component is not immediately needed on the initial render. This is because static imports will ensure that the entire `Post` component and its dependencies are fetched and executed before the `App` component is initialized.

Dynamic import

To avoid this issue, we can attempt to load the Post component lazily (i.e., dynamically) with React's `lazy()` function.

Dynamic import of the `Post` component

```
import React, { lazy, Suspense } from 'react';

// dynamically importing the Post component
const Post = lazy(
  () => import("./components/Post"),
);

function App() {
  return (
    <div>
      {/* using Suspense to render fallback while
          Post is dynamically loading */}
      <Suspense fallback={<div>Loading...</div>}>
        <Post />
      </Suspense>
    </div>
  );
}
```

```
export default App;
```

In the example above, we leverage the dynamic import() syntax to asynchronously load the `Post` component. The `import()` function returns a promise that resolves to the imported module, allowing us to utilize it in conjunction with React's `lazy()` function to create a lazily-loaded component.

With this approach, the `Post` component is only loaded when it's actually needed, reducing the initial bundle size and improving the load times of the `App` component.

The `Suspense` component is used to display a loading message or placeholder while the `Post` component is being loaded.

Lazy-load on interaction

We can also use this lazy-loading pattern to dynamically load components on interaction/click.

Here's an example of how we can import the `Post` component with a click instead of dynamically loading the component as the parent is being rendered.

Lazy-loading the `Post` component on button click

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

function App() {
  const [Post, setPost] = useState(null);

  const handleClick = () => {
    import("./components/Post").then((module) => {
      setPost(() => module.default);
    });
  };

  return (
    <div>
      {Post ? (
        <Post />
```

```
      ) : (
        <button onClick={handleClick}>
          Load Post
        </button>
      )}
    </div>
  );
}

export default App;
```

In this example, we use the `useState` Hook to initialize a `Post` state variable as `null`. When the button is clicked, we use the dynamic `import()` function to load the `Post` component and then set it to the value of the `Post` state. Once `Post` is set, it will be rendered to the screen.

Note that using React's `lazy()` function with dynamic imports only works with default exports, so if the `Post` component has named exports, we'll need to adjust the import statement accordingly. Also, keep in mind that dynamic imports with React's `lazy()` function should only be used for large components that are not needed immediately, as it adds some complexity to the code and can cause issues with server-side rendering.

Lazy-load with the Intersection Observer API

Intersection Observer is a JavaScript API that allows us to detect when an element is visible in the viewport. This can be useful for implementing on-demand code splitting, where code is loaded when the user scrolls to a specific section of the page.

To use the Intersection Observer API in your React application, we can create our own custom functionality or import this functionality from a third-party library like react-intersection-observer.

Here's a rough example of how we could use a custom `useIntersectionObserver()` Hook to lazily load the `Post` component when it enters the viewport:

Lazy-loading with Intersection Observer

```
import React, {
  useState,
  useRef,
  lazy,
  Suspense,
} from "react";
import useIntersectionObserver from "./hooks";

const Post = lazy(
  () => import("./components/Post"),
);

function App() {
  const [shouldRenderPost, setShouldRenderPost] =
    useState(false);
  const postRef = useRef(null);

  const handleIntersect = ([entry]) => {
    if (entry.isIntersecting) {
      setShouldRenderPost(true);
    }
  };
  useIntersectionObserver(
    postRef,
    handleIntersect,
    { threshold: 0 },
  );

  return (
    <div>
      <div style={{ height: "1000px" }}>
        Some content before the post
      </div>

      <div ref={postRef}>
        {shouldRenderPost ? (
          <Suspense
            fallback={<div>Loading...</div>}
          >
            <Post />
          </Suspense>
        ) : (
```

```
            <div>Loading...</div>
          )}
        </div>

        <div style={{ height: "1000px" }}>
          Some content after the post
        </div>
      </div>
    );
}

export default App;
```

In the above example, we're using a `useIntersectionObserver()` Hook to watch for changes in the visibility of the `postRef` element, and trigger the `handleIntersect()` callback when it enters the viewport. The `shouldRenderPost` state property is set to a value of `true` when the element is intersecting, which triggers the rendering of the `Post` component inside a `Suspense` component.

Note that this example assumes that the `useIntersectionObserver()` Hook is defined somewhere in our codebase.

By loading resources only when they are needed, lazy-loading components can significantly minimize the initial load of an application, thereby improving user experience and resource utilization. While lazy-loading focuses on loading components only when necessary, the concept of **code-splitting** takes this a step further by breaking down an entire application into smaller chunks that can be loaded independently.

Code-splitting

Code splitting is a technique for optimizing the performance of large applications by splitting the application's code into smaller, more manageable chunks.

By having a modular or componentized application structure, we naturally pave the way for more efficient code-splitting. When components are designed to be self-contained and independent, it becomes easier to separate them into distinct chunks that can be loaded

on demand. This modular approach aligns perfectly with the core idea behind code-splitting, where the aim is to load only the necessary code for the user at any given moment rather than loading the entire application upfront.

In React applications, code-splitting patterns commonly include:

- **Splitting by route**: Load page modules as user navigates.
- **Splitting by component**: Lazy-load large components like graphs and tables.
- **On-demand loading**: Load code when a user clicks buttons, dropdowns, etc.

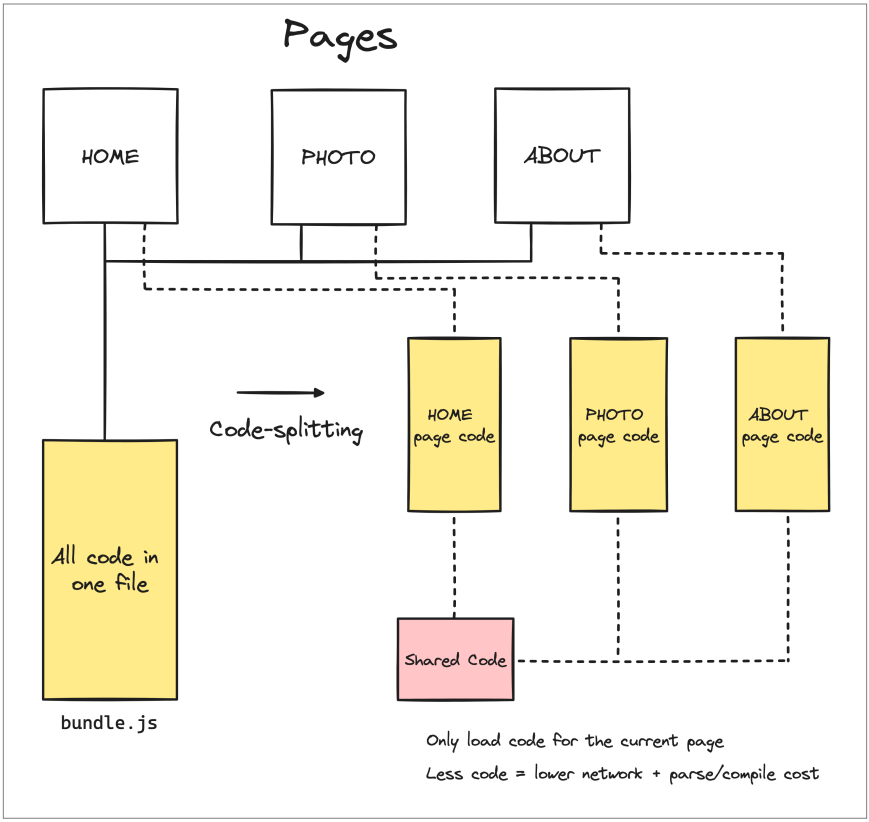


Figure 3-3. Code-splitting a monolithic bundle.js into granular chunks.

The first step in implementing code splitting is to identify the critical path of our application. The critical path is the sequence of resources that must be loaded before our application can be displayed to the user. By

identifying this critical path, we can determine which resources should be loaded first and which resources can be loaded later using advanced code-splitting techniques.

In the next few sections, we'll discuss some strategies for approaching advanced code splitting in a large React application.

Entry point splitting

The entry point is the initial JavaScript file that is loaded when a user visits a website. **With entry point splitting, we break up the initial JavaScript file into smaller chunks that are only loaded when needed**, reducing the initial load time for the page.

For example, imagine we have a website with:

- A home page
- A product page
- And a contact page

Each of the pages has its own unique JavaScript code. If we load all of the code at once, the initial page load time could be slow. With entry point splitting, we can break up the code for each page into separate chunks. With this type of splitting, when a user visits:

- The home page: only the code for the home page is loaded.
- The product page: only the code for the product page is loaded.
- The contact page: only the code for the contact page is loaded.

This type of code-splitting results in faster load times and a better user experience since only the necessary code is loaded for each specific page, reducing the amount of redundant or unnecessary data being fetched.

Vendor splitting

Vendor splitting is a technique used to separate out third-party dependencies from your own code. When we use a third-party library or framework, the code for that library is included in our JavaScript bundle. This can make the bundle larger and slower to load and can also cause cache invalidation issues when the library is updated.

With vendor splitting, we can break out the code for these third-party dependencies into a separate chunk that can be cached independently. This means that when we update our own code, the end user won't need to re-download the entire library since it's already cached. This can result in faster load times and a better user experience by optimizing caching and reducing unnecessary data downloads.

Dynamic splitting

Dynamic splitting is a technique used to load JavaScript code on demand, as needed. This is useful for large-scale JavaScript applications where different parts of the code are only needed in certain situations. For example, if we have an application with a dashboard and a settings page, the code for the dashboard might not be needed when the user is on the settings page, and vice versa.

With dynamic splitting, we can load the code for each page or component only when it's needed. This can reduce the initial load time for the page and improve performance overall. It can also help to keep the size of the JavaScript bundle under control, which is important for large-scale applications.

Dynamic splitting differs from entry-point splitting in that it doesn't rely solely on predefined entry points. Instead, it leverages tools and patterns like React's `lazy` and `Suspense` or the dynamic `import()` function to split code at specific modules or components. This allows developers to granularly control when different parts of the codebase are loaded based on user interactions or other runtime conditions.

Component-level splitting

In component-level code-splitting, each component is lazy-loaded only when it's needed, which means that the application loads only the components that are required for the current page. This technique can lead to more efficient use of bandwidth, but it can sometimes increase latency due to the need to load components on demand.

Route-based splitting

In route-based code-splitting, the application is split into separate bundles based on routes. When a user navigates to a different route, the appropriate bundle is loaded on demand, reducing the amount of code that needs to be downloaded initially. This technique can help reduce the initial load time of an application, but it may not be as efficient as component-level code-splitting in terms of bandwidth usage.

Trade-offs with aggressive code-splitting

Aggressive code-splitting refers to the practice of extensively breaking down the application's JavaScript into numerous small chunks. While code-splitting has clear benefits in terms of loading only the necessary code for a given view or action, there are some difficulties associated with aggressive code-splitting.

1. **Granularity trade-off**: When we aggressively code-split, we end up with a large number of smaller chunks of code. This can be good for caching and de-duplication but bad for compression and browser performance. Smaller chunks compressed individually get lower compression rates, and loading performance can be impacted, even with as low as 25 chunks and very severely at 100+ chunks.

2. **Interoperability**: Different browsers, servers, and CDNs may implement code-splitting differently, which can sometimes lead to compatibility issues.

3. **Overhead**: While code-splitting can improve loading performance, it can also introduce additional overhead due to the need to process, fetch, and parse multiple files. This could sometimes slow down an application, especially on slower devices or networks.

4. **Debugging**: With a large number of smaller chunks, it can be difficult to debug the code and identify issues as the code is spread across multiple files.

5. **Build complexity**: Aggressive code-splitting can make the build process more complex and time-consuming, as the codebase is broken down into multiple smaller chunks that need to be managed and sometimes built separately.

Wrap up

Modularity, through componentization, not only makes our applications more maintainable and scalable but also enhances the developer experience by providing a clear structure and reusability of components.

As applications grow in complexity, there's an ever-increasing need to optimize for performance and user experience. Code-splitting allows us to break down applications into manageable chunks, ensuring users load only the necessary code at the right time. Having our application broken down into components makes implementing code-splitting efficient, as we can dynamically load individual components based on user interaction or the current view.

In the next chapter, we'll spend a bit more time discussing and sharing helpful resources on the topic of **Performance**.

Performance

Web performance is the measure of the speed and efficiency with which web pages load and operate. For large-scale web applications, maintaining optimal performance is important in ensuring a positive user experience because as an application grows, the challenges to performance often intensify due to growing traffic, heavier resource usage, and more extensive functionalities.



Whether we're focused on optimizing for the Core Web Vitals metrics, our own custom User Timing metrics, or otherwise, there are a number of important topics in performance worth keeping in mind. With that said, performance is a *very* vast topic with numerous nuances, intricacies, and methodologies, each deserving its own deep dive.

In today's chapter, we'll explore some key concepts of performance optimization within the realm of web development and JavaScript. For those looking to delve deeper into specific aspects, we'll share relevant links and resources where applicable.

Understanding how browsers work

We want our web applications to load quickly, be responsive, and provide a seamless user experience. To achieve this, it first helps to understand how browsers render, paint, and load content.

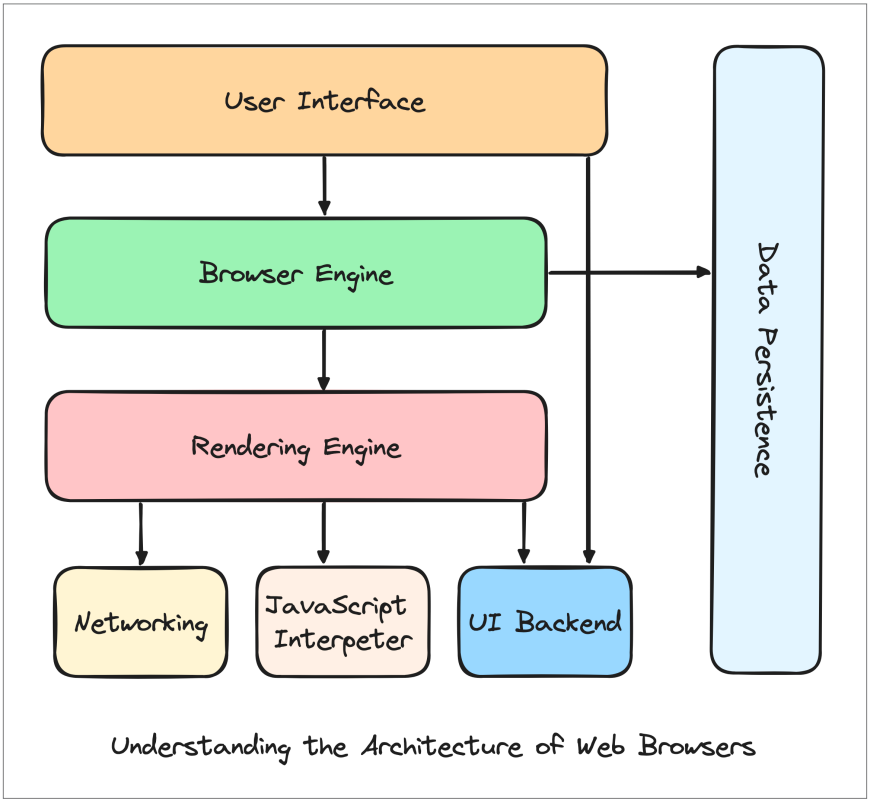


Figure 4-1. A high-level picture of how modern browsers work

Rendering

Let's start with **rendering**. When a browser receives an HTML document, it goes through a process called **parsing**. The parser reads the

document and creates a DOM (Document Object Model) tree, which represents the structure of the document. For a simple HTML document like the following:

HTML document

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Sample Page</title>
</head>
<body>
  <header>
    <h1>Welcome to My Page</h1>
  </header>
  <section>
    <p>This is a paragraph in a section.</p>
    <ul>
      <li>List item 1</li>
      <li>List item 2</li>
      <li>List item 3</li>
    </ul>
  </section>
  <footer>
    <p>Contact us at contact@example.com</p>
  </footer>
</body>
</html>
```

The DOM tree can be visualized to look something like this:

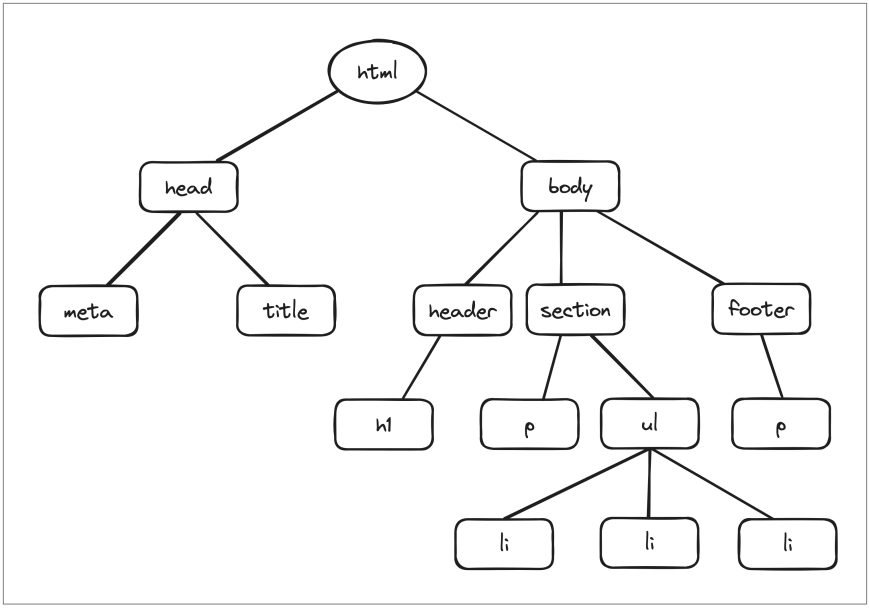


Figure 4-2. DOM (Document Object Model) tree

The browser uses this DOM tree to render the content on the page. As the browser parses the HTML and creates the DOM, it also processes the CSS and builds the CSSOM (CSS Object Model)—a representation of the CSS styles that will be applied to the DOM elements.

However, rendering is more than just displaying the content. There are two main phases to rendering: **layout** and **paint**. During the layout phase, the browser calculates the size and position of each element on the page based on its style and the layout rules defined in the CSS. Once the layout is complete, the browser enters the paint phase, during which it draws each element on the screen.

Understanding this process is crucial because it helps us optimize our application's performance. For example, if we have a large number of elements on our page, it can take a long time for the browser to calculate the layout. This can lead to a slow and unresponsive application. By minimizing the number of elements on the page or by using CSS grid or Flexbox to simplify the layout, we can oftentimes improve our application's performance.

Loading

Another important aspect of browser performance is **loading**. When a browser receives an HTML document, it starts loading the resources referenced in the document, such as images, stylesheets, and scripts. This process can be a bottleneck, especially if we have a large number of resources to load.

Understanding the loading process can also help us optimize our application's performance. For example, we can use lazy-loading to only load resources when they are needed. We can also use resource hints, such as prefetching or preloading, to give the browser a heads-up about resources that will be needed in the future.

Identifying performance bottlenecks

Perhaps the most important reason to deeply understand browser engineering is to be able to identify and resolve performance bottlenecks. By using browser developer tools, such as Chrome DevTools, we can inspect our application's performance and identify areas that are slowing it down. This can help us optimize our code and improve our application's performance.

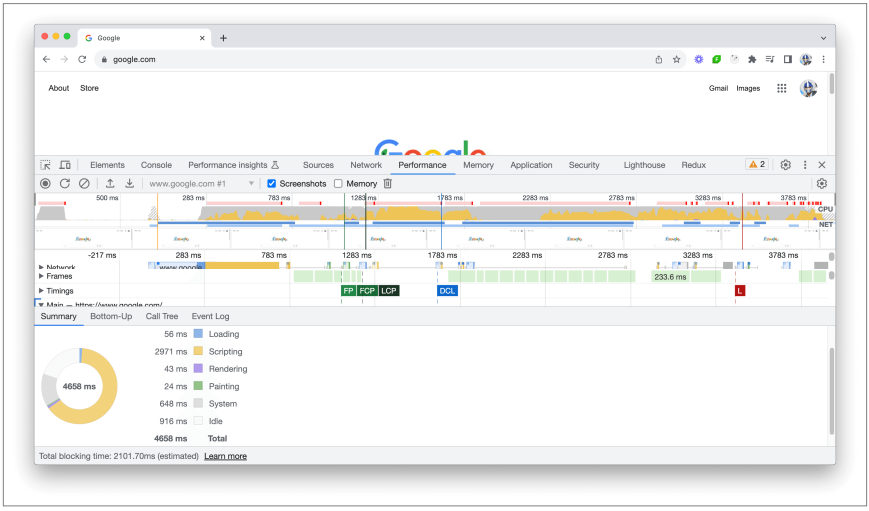


Figure 4-3. The Performance tab in Chrome DevTools

To dive deeper into browser engineering, we recommend reading the "Inside the Browser" series on the Google Chrome Developers blog. This series provides a comprehensive look at the various components of a browser and how they work together to render web pages. For a more detailed dive into the inner workings of the DOM and CSSOM, be sure to check out the "Constructing the Object Model" and "Render-tree Construction, Layout, and Paint" articles from web.dev.

We also recommend the free book "Web Browser Engineering" by Pavel Panchekha and Chris Harrelson. This book provides a deep dive into browser engineering, covering topics such as parsing, layout, rendering, and networking. It's a great resource for any developer who wants to understand how browsers work from the ground up.

Understanding and reducing the cost of JavaScript

JavaScript has become a critical component of modern web development, powering interactive user experiences and dynamic content. However, the cost of using JavaScript can negatively impact performance, particularly in the areas of download time and CPU execution time. These factors can lead to slower page load times, reduced interactivity, and a poor user experience, especially for users on laptops and phones with slower CPUs.

Optimizing JavaScript CPU execution often enhances interactivity metrics, such as Total-Blocking-Time (TBT) and Interaction-to-Next-Paint (INP). In the following section, we'll delve into the cost of JavaScript in 2024, exploring the latest techniques and best practices for improving JavaScript performance and delivering fast, responsive web experiences.

Download time (network transfer)

One of the biggest challenges of JavaScript is reducing download time. Despite the growth of high-speed networks like 5G, many users still experience slow connection speeds, particularly when on the go. To optimize download time, it's important to **keep JavaScript bundles**

small, especially for mobile devices. Smaller bundles improve download speeds, lower memory usage, and reduce CPU costs.

It's also important to **avoid having a single large bundle**. If a bundle exceeds around 50-100 KB (kilobytes), it should be split up into separate smaller bundles. This is because, with HTTP/2 multiplexing, multiple request and response messages can be in flight at the same time, reducing the overhead of additional requests.

We discuss ways to code-split a single large bundle into smaller bundles in the previous chapter: **Modularity**.

Execution time

Once downloaded, script execution time will now be a dominant cost in 2024. User interaction can be delayed if the browser's main thread is busy executing JavaScript, so optimizing bottlenecks with script execution time and network can be impactful. To optimize for this:

- **Avoid long tasks** that can keep the main thread busy and push out how soon pages are interactive. Long tasks monopolize the main thread, so it's important to break them up into smaller tasks. By splitting up code and prioritizing the order in which it is loaded, we can get pages interactive faster and hopefully have lower input latency.

- **Avoid large inline scripts** (as they're still parsed and compiled on the main thread). A good rule of thumb is: if the script is over 1 KB (kilobyte), avoid inlining it. This is also because 1 KB is when code caching kicks in for external scripts.

Parse and compile

One large change to the cost of JavaScript over the last few years has been an improvement in how fast browsers can parse and compile script. In 2024, parse and compile costs are no longer as slow as they once were. The raw JavaScript parsing speed of V8, Google's JavaScript engine, has increased since Chrome 60+, and the raw parse and compile cost has become less visible/important due to other optimization work in Chrome that parallelizes it.

V8 has reduced the amount of parsing and compilation work on the main thread by an average of 40% by parsing and compiling on a worker thread. This is in addition to the existing off-main-thread streaming parse and compile. V8 can parse and compile JavaScript without blocking the main thread, which is a significant improvement over older versions of Chrome that would download a script in full before beginning to parse it.

On repeat visits, V8's (byte)code-caching optimization helps. When a script is first requested, Chrome downloads it and gives it to V8 to compile. It also stores the file in the browser's on-disk cache. When the JS file is requested a second time, Chrome takes the file from the browser cache and once again gives it to V8 to compile. This time, however, the compiled code is serialized and attached to the cached script file as metadata. The third time, Chrome takes both the file and the file's metadata from the cache and hands both to V8. V8 deserializes the metadata and can skip compilation. Code caching kicks in if the first two visits happen within 72 hours.

Mobile devices

In addition to the above-mentioned factors, JavaScript can also impact the battery life of mobile devices. Since JavaScript requires a lot of CPU resources to execute, it can drain the battery life faster, especially on low-end devices. This makes JavaScript execution time important for phones with slow CPUs.

Due to differences in CPU, GPU, and thermal throttling, there are huge disparities between the performance of high-end and low-end phones. This matters for the performance of JavaScript, as execution is CPU-bound.

To address these issues, web developers need to focus on optimizing the download and execution time of their JavaScript code. This can be achieved by:

- Reducing the size of JavaScript bundles.
- Breaking down large bundles into smaller ones.
- Avoiding long tasks that can block the main thread.

Optimize interactions

Interaction-readiness is a measure of web performance that assesses how quickly a web page can respond to user interactions.

It is a key aspect of web performance because users expect web pages to respond immediately to their input, such as clicks, scrolls, and typing. If a web page is slow to respond, users may become frustrated and abandon the site. Interaction-readiness can be influenced by a variety of factors, such as network latency, server processing time, and browser rendering performance. You may have come across a variety of metrics that measure different aspects of interaction-readiness, including some lab metrics like Time To Interactive (TTI) or Total Blocking Time (TBT).

Interaction to Next Paint (INP) is also a metric that measures the responsiveness of a web page to user interactions. However, INP measures the responsiveness from the moment the user initiates the interaction until the next frame is painted on the screen. It provides a more accurate estimate of a web page's load and runtime responsiveness.

React.js time slicing, implemented through startTransition and Suspense, allows developers to opt-in to selective or progressive hydration, enabling hydration to be done in small slices that can be interrupted at any point. This approach helps improve INP, making React apps more responsive to keystrokes, hover effects, and clicks, even during large transitions such as auto-complete. Next.js is working on a new routing framework called App Router that will use `startTransition` by default for route transitions, helping Next.js site owners adopt React time-slicing and improve the responsiveness of their route transitions.

Networking

HTTP/3 is a new version of the Hypertext Transfer Protocol that is designed to improve performance and security over the internet. It uses the QUIC transport protocol, which is designed to reduce latency and congestion and supports multiplexed streams and connection migration. All major browsers like Google Chrome, Mozilla Firefox, and Microsoft Edge support HTTP/3 out of the box or by enabling it in their settings.

Streaming refers to the ability to send and receive data in chunks or segments rather than waiting for the entire file or resource to be loaded. This can improve performance by allowing users to start consuming content more quickly.

Flushing refers to the ability to send data to the browser before the entire response has been generated. This can be useful for streaming content, as well as for improving perceived performance by starting to render the page before all of the resources have been loaded.

For large-scale JavaScript web applications, these technologies and techniques can be important for improving performance, particularly for users with slower internet connections or who are accessing the application from remote locations. By using HTTP/3, streaming, and flushing, you can reduce latency and improve perceived performance, which can lead to increased engagement and retention.

Reducing the impact of third-party dependencies

When working in large JavaScript and React applications, third-party dependencies can sometimes be a big contributor to performance bottlenecks. These dependencies can range from libraries and frameworks to widgets, analytics, and scripts. While they can provide significant functionality and save development time, they also introduce additional overhead.

Reducing the impact of third-party dependencies in React applications can be a challenging task, but here are some steps that can be followed to reduce the cost of using these dependencies:

Identify the most expensive dependencies

When auditing the use of third-party dependencies and their impact on performance, the first step is to identify the third-party dependencies that are taking the most time to load and execute. To help us here, we can leverage tools like Chrome DevTools, Lighthouse, or WebPageTest to identify the slowest-loading third-party dependencies.

Evaluate the necessity of each dependency

Once we've identified the slowest dependencies, evaluate their necessity. Are they essential for our application's core functionality? Can they be replaced with smaller or faster alternatives?

It's important to evaluate the necessity of each dependency regularly to determine if it's worth the performance cost.

Consider alternative libraries

If a dependency is not essential for our application's core functionality, consider replacing it with an alternative library that is smaller and faster. Oftentimes, there are many alternative libraries available for popular third-party dependencies that can achieve the same outcome.

Optimize the loading of dependencies

We can optimize the loading of third-party dependencies by using techniques like lazy-loading, code-splitting, and tree shaking. Lazy-loading can help load a dependency only when it's needed, and code-splitting can break up code into smaller chunks and load them only when necessary.

Tree shaking is a technique used by bundlers and build tools like Vite and Webpack to eliminate dead code or unused exports from the final bundle. With tree shaking, we can eliminate unused code from dependencies that are being used.

Use CDN hosting

Using CDN hosting for third-party dependencies can help reduce the loading time and improve the performance of our application. CDNs have servers located all around the world, which means that users can download the dependencies from a server that is closest to them.

Keep in mind that double-keyed caching can heavily reduce the benefits of CDN caching. However, there may still be benefits from serving scripts from the edge.

Analyze your bundle

Tools like Webpack Bundle Analyzer and Lighthouse Treemap can help us analyze our bundle and identify the dependencies that are taking up the most space.

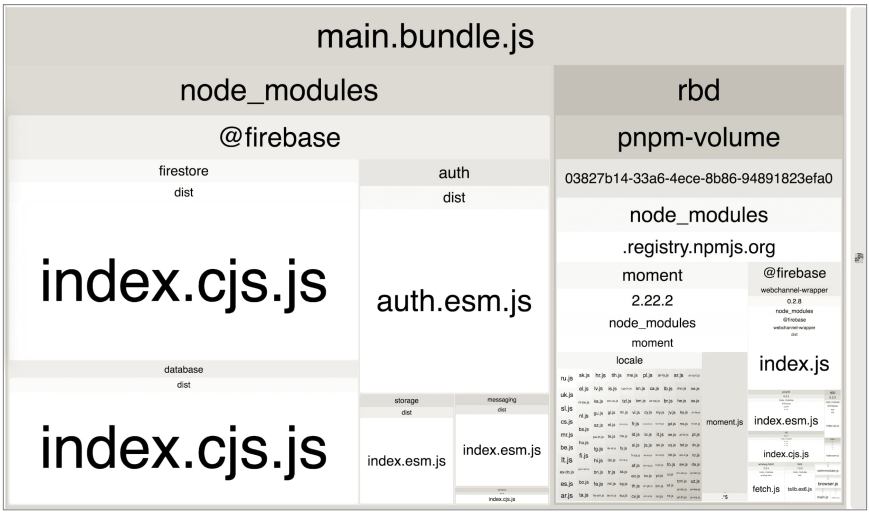


Figure 4-4. A visualization of an app's bundle by Webpack Bundle Analyzer

Once we've identified the largest dependencies, we can try to replace them with smaller or faster alternatives.

Optimize third-party dependency configurations

Some third-party dependencies may have configurations that can be optimized to improve their performance. For example, if we're using a data visualization library, we can reduce the amount of data it is rendering by limiting the number of items displayed.

Rendering patterns

Understanding rendering patterns is crucial for building large-scale JavaScript and React applications that deliver the best possible user experience. Rendering patterns have evolved significantly over time,

with server-side rendering (SSR) and client-side rendering (CSR) giving way to more complex patterns being discussed and evaluated on various forums today. With so many options available, it can be overwhelming to choose the right one for our project. However, it's essential to remember that each pattern is designed to solve a specific use case, and what works for one may not be suitable for another.

Additionally, different types of pages on the same website may require different rendering patterns. The Chrome team recommends using static or server-side rendering instead of a full rehydration approach. Progressive loading and rendering techniques, when used with modern frameworks, can help strike a balance between performance and feature delivery over time.

Here's a summary of useful rendering patterns inspired by patterns.dev and StateOfJS:

- Client-Side Rendering (CSR) a Single Page Application (SPA)
  - Apps that run entirely in the browser.

- Multi-Page Application (MPA)
  - Apps that run entirely on the server, with minimal client-side dynamic behavior.

- Static Rendering for Static Site Generation (SSG)
  - Pre-rendered static content, with or without a client-side dynamic element.

- Server-Side Rendering (SSR)
  - Dynamically rendering HTML content on the server before rehydrating it on the client.

- Partial Hydration
  - Only hydrating some of your components on the client (e.g., React Server Components).

- Progressive Hydration
  - Controlling the order of component hydration on the client.

- Islands Architecture
  - Isolated islands of dynamic behavior with multiple entry points in an otherwise static site (Astro, Eleventy).

- Incremental Static Generation
  - Being able to dynamically augment or modify a static site even after the initial build (Next.js ISR, Gatsby DSG).

- Streaming SSR
  - Breaking down server-rendered content into smaller streamed chunks.

- Resumability
  - Serializing framework state on the server so the client can resume execution with no duplicated code.

- Edge Rendering
  - Altering rendered HTML at the edge before sending it on to the client.

Optimizing perceived performance

Perceived performance is the subjective experience of how quickly a website or app loads and responds, as opposed to the actual time it takes to load. It's influenced by a variety of factors, including visual cues, feedback, and animations. By improving perceived performance, we can oftentimes create a better user experience even if the actual load times are slower. While optimizing the perceived performance of an app can be important, it should not detract from the work done to make the core user experience actually load faster.

One technique for improving perceived performance is progressive loading. Rather than waiting for an entire page or app to load before displaying anything, progressive loading prioritizes the most crucial content and loads it first while continuing to load additional content in the background.

One popular approach to progressive loading is the use of **skeleton screens**.

Skeleton screens and placeholder UI

Skeleton screens, also known as skeleton loaders or content placeholders, are a design pattern used to improve the perceived performance of web applications. These screens consist of simple outlines or "skeletons" of the content that will eventually be loaded on the page, such as images, text, or other media.

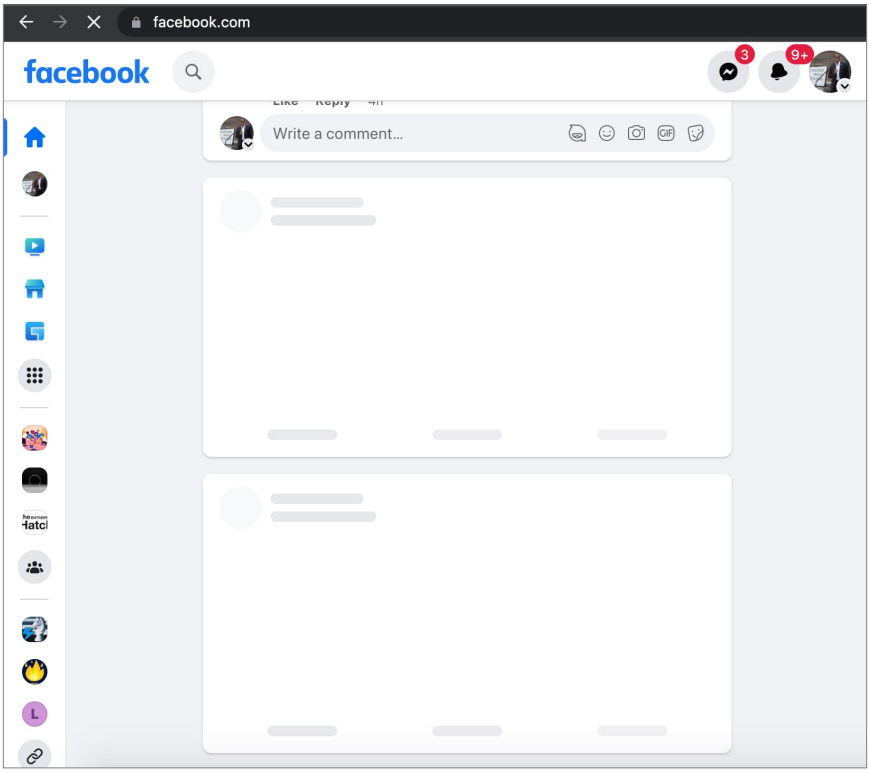


Figure 4-5. Skeleton loading behavior for the facebook.com Feed

The idea behind skeleton screens is to give the user immediate visual feedback that content is being loaded, even if it is not yet available. This technique can help reduce perceived loading times and increase the user's engagement and satisfaction with the application.

Skeleton screens can be implemented in various ways, but they typically involve using HTML and CSS to create a placeholder structure that

resembles the final content. This structure is then displayed on the page while the actual content is being fetched or loaded in the background.

Another related technique is to use skeleton loaders for individual components or UI elements within a page. For example, a complex data table might have a skeleton loader that displays a simplified table structure with empty cells while the actual data is being loaded. This technique helps users understand the layout of the table and anticipate the data that will eventually be displayed.

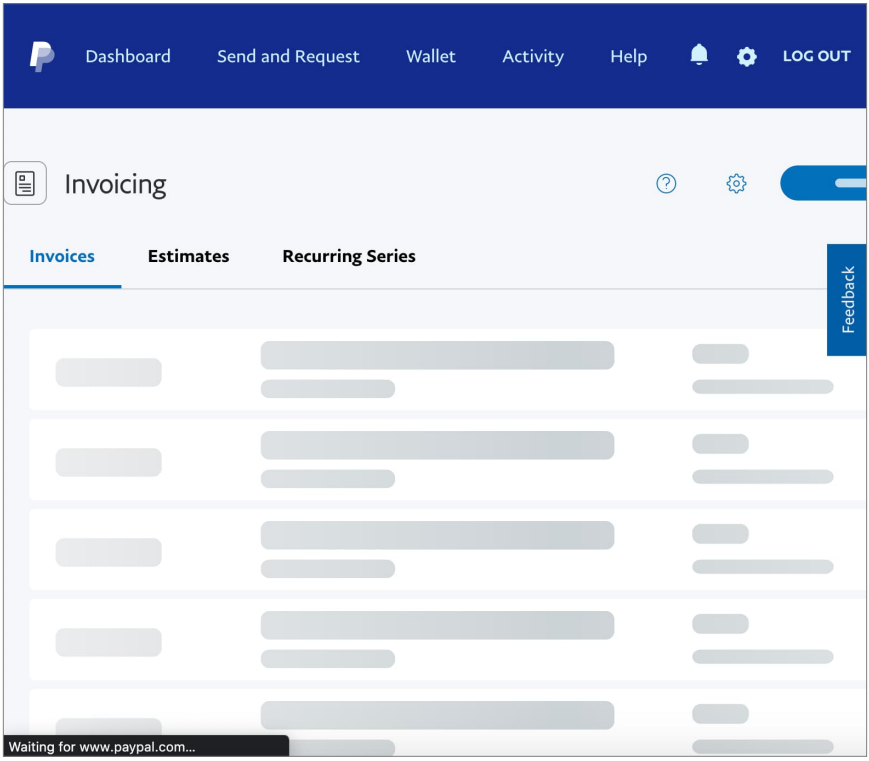


Figure 4-6. Skeleton loading behavior for paypal.com's Invoicing table

Placeholder UI is another approach that involves using pre-existing visual elements to represent content that is not yet available. For example, a news application might use a default image placeholder for articles that do not have an associated image. This technique can be effective in reducing perceived loading times, but it is not as visually informative as a skeleton screen.

While skeleton screens and related techniques can be effective in improving perceived performance, they also have some potential drawbacks. For example, if the skeleton screen does not accurately reflect the final content, it can create confusion or frustration for the user. Similarly, if the actual content takes too long to load, the skeleton screen can become a source of annoyance rather than a helpful indication of progress. The pros and cons of using skeleton screens and other progressive loading techniques can be summarized as follows:

Pros:

1. **Improved perceived performance**: As mentioned, skeleton screens can help reduce the perceived wait time and improve the overall perceived performance of the website or app.

2. **Better engagement**: By displaying content as soon as possible, progressive loading can help keep users engaged and interested in the content rather than making them wait for the entire page to load.

3. **More efficient use of resources**: By prioritizing the most important content, progressive loading can reduce the load on servers and networks, making the website or app more efficient.

Cons:

1. **Increased complexity**: Progressive loading can sometimes require more complex coding and design, which can be challenging to implement and maintain.

2. **Risk of visual distraction**: Skeleton screens can be distracting or even confusing if not implemented properly, potentially leading to a worse user experience.

3. **Limited functionality**: Progressive loading may not be suitable for all types of content, particularly interactive elements or complex interfaces.

Finally, it is important to ensure that progressive loading techniques are implemented in a way that is accessible and compatible with a variety of devices and platforms.

Optimizing Performance - The Highlights

Besides the concepts discussed in this chapter, there is a wide array of techniques and strategies for enhancing web performance. While this write-up touches only the surface of this extensive topic, we encourage further exploration into the following areas and resources to gain deeper insights into optimizing web performance for large-scale web applications:

Service Workers, Precaching, and Navigation Preload

1. Introduction to Service Workers
2. Precaching with Workbox
3. Navigation Preload: Speeding up Service Worker Start-up

Streaming

1. Stream API: A guide to streams
2. Streaming HTML with the Fetch API and ReadableStreams
3. Streams for the Web

V8 Code Caching and Bytecode Cache

1. V8 Engine: A deep dive into V8 code caching
2. Understanding JavaScript Engine: V8
3. Optimizing V8 bytecode caching in Electron apps

Stale While Revalidate

1. Using the Stale-While-Revalidate Cache Strategy

2. Improve Web Performance with Stale-While-Revalidate

Brotli Compression

1. Brotli Compression: How it works and why you should use it
2. Better Compression with Brotli
3. Enabling Brotli Compression on Your Web Server

Early Hints, Priority Hints, and HTTP/3

1. HTTP/3: From Root to Tip
2. Early Hints: Improving Web Performance
3. Priority Hints: A First Look

Client Hints

1. An Introduction to Client Hints
2. Responsive Images with Client Hints
3. Client Hints: Device-aware content delivery

Adaptive Loading

1. Adaptive Loading: Improving Web Performance Based on Network and Device Conditions
2. Adaptive Loading Patterns with React
3. Adaptive Loading for a Faster Web

List Virtualization

1. List Virtualization - Patterns.dev

2. Infinite Scrolling without layout shifts
3. Rendering large-lists with React Virtualized

Content Visibility

1. CSS Content Visibility on web.dev
2. Content Visibility on MDN

Image Optimization

1. Learn Images web.dev course
2. Optimize your images
3. Optimizing for Largest Contentful Paint

By understanding and implementing the techniques listed above, we can significantly improve the web performance of our large-scale applications, resulting in a better user experience and increased engagement.

Performance Culture

Alex Russell has an authoritative article on performance management maturity that is worth reading. As you can imagine, shifting such culture in large organizations takes time, but it is worth being aware of as "getting fast" is one challenge, and "staying fast" is another.

Performance culture refers to the attitude and approach that an organization takes towards web performance, where performance is a critical metric for business success. A healthy performance culture involves incorporating scientific methods into processes and approaches, acknowledging the complexity of modern systems, and working together to learn and investigate the unknown.

Having a performance culture is crucial because web performance is directly linked to business outcomes, including customer satisfaction, user engagement, and revenue. Therefore, performance should be a

priority, with management of latency, variance, and other performance attributes being incorporated into OKRs (Objectives and Key Results) to ensure that performance is always on the agenda.

To approach performance culture effectively when building large-scale JavaScript applications, it's important to have a clear understanding of the levels of performance management maturity. These levels include:

- **Level 0** (Bliss)
- **Level 1** (Fire Fighting)
- **Level 2** (Global Baselines & Metrics)
- **Level 3** (P75+, Site-specific Baselines & Metrics)

The approach to performance culture will differ at each level. At Level 0, the organization is not aware of the problem and lacks any performance management strategies. At Level 1, managers are aware of the performance problem and are attempting to fix it. At Level 2, teams are moving towards global baseline metrics and benchmarks to assess performance, while at Level 3, teams have a deep understanding of site-specific performance metrics and have incorporated scientific methods into their approach.

In summary, building a healthy performance culture involves prioritizing web performance, incorporating scientific methods into processes, and using data-driven approaches to improve performance over time. This helps organizations achieve consistent velocity and maximize their potential while also providing a better user experience and driving business success.

Design Systems

A design system is a collection of reusable components, guidelines, and assets that help teams build cohesive products. Many organizations choose to build and maintain their own custom design systems tailored to their specific needs and branding. This approach allows for greater control and uniqueness in the design language used across their products.



It's also common for teams, especially those with limited resources or seeking industry-standard practices, to leverage popular open-source design systems. Examples of such widely recognized systems include Material by Google, Polaris by Shopify, Human Interface Guidelines by Apple, Fluent Design System by Microsoft, and many others. These open-source systems offer robust, well-tested design

principles and components and are sometimes actively used within the organizations that created them.

Design systems play an important role in large web applications as they provide a centralized library of design elements, patterns, and guidelines that help ensure consistency and efficiency in the development and design process.

"Design systems are an incredibly useful tool for building large-scale applications. When you're building a suite of enterprise-level applications that have the same brand identity, your two biggest concerns should be accessibility and consistency. Design systems solve those problems at the core.

Having a suite of accessible, robust, and scalable components like buttons, dropdowns, and modals on top of a set of design tokens for spacing, color, and typography allow your engineering teams to more quickly build applications without worrying about each individual pixel. Additionally, when updates are made to the brand identity or designs, components only need be updated in one central location and the changes propagate to each integrated platform."
[Emma Bostian | Engineering Manager @ Spotify]

In the following chapter, we'll delve deeper into the core elements of what makes a design system by exploring their benefits and some best practices for their implementation.

Coding Style Guides

Coding style guides are sets of rules and guidelines that ensure consistency in code across a team or organization. Coding style guides play a crucial role in the context of design systems by providing and facilitating:

- **Uniformity**. A standardized set of rules and conventions helps ensure uniformity, which promotes consistency in the development environment.

- **Modular code structure**. Coding style guides encourage the creation of modular and reusable components. This is central to the efficiency of design systems by allowing the seamless sharing and reusing of components across different projects and teams.

- **Cohesive aesthetics**. In conjunction with design principles and visual guidelines, coding style guides can help in rendering components with consistent aesthetics and behavior by always attempting to align visual elements with their code structure.

As a simple example, one widely recognized style guide is the BEM (Block, Element, Modifier) methodology used for structuring CSS classes. BEM stands for Block, Element, Modifier and is a naming convention that helps developers create modular and reusable CSS classes.

Using BEM, class names are structured like the following:

BEM (Block, Element, Modifier)

```
.block {}
.block__element {}
.block--modifier {}
```

The block is the main component or container, the element is a part of the block, and the modifier is a variation of the block or element. An example of using BEM to structure how CSS classes are applied can look something like this:

Using BEM methodology to name classes for a simple card

```
<div class="card card--featured">
  <h2 class="card__title">Featured Product</h2>
  <p class="card__description">
    Lorem ipsum dolor sit amet
  </p>
  <a href="#" class="card__link">Learn More</a>
</div>
```

BEM is only one example of a coding style guide for how to structure HTML elements and CSS properties. Many others exist, such as Scalable and Modular Architecture for CSS (SMACSS), Object-Oriented CSS (OOCSS), etc.

Coding style guides extend beyond CSS structuring and can also include guidelines for naming conventions, indentation, formatting, commenting, and more. When building a design system of your own, you and your

team can help craft these guidelines from scratch or rely on established frameworks and conventions already in use by the wider development community.

An example of an established framework is Tailwind—a tool that has gained traction in the development community as a utility-first CSS framework. Unlike traditional methodologies like BEM, which prioritize structured naming conventions, Tailwind offers developers a set of utility classes that can be applied directly to HTML elements to style them.

Styling with Tailwind's utility classes

```
<div class="p-3 bg-white shadow rounded-lg">
  <h3 class="text-xs border-b">font-mono</h3>
  <p class="font-mono">
    The quick brown fox jumps over the lazy dog.
  </p>
</div>
```

The goal behind Tailwind is to provide a flexible and customizable way to design interfaces, and its utility-first approach allows developers to write less CSS to achieve a more consistent and scalable design. Each class represents a specific style or a set of CSS properties. For example:

- p-3 represents `padding: 0.75rem`
- bg-white represents `background-color: white`
- rounded-lg represents `border-radius: 0.25rem`
- etc.

When it comes to design systems, the choice of a style guide should be guided by the project's requirements, team preferences, and the overall goals of the design system. Key considerations include:

1. **Flexibility and Scalability**: A chosen style guide should be flexible enough to accommodate changes and scalable to handle the project's growth over time. It should allow for the easy integration of new components and styles without disrupting the existing system.

2. **Team Collaboration and Efficiency**: A good coding style guide should facilitate efficient collaboration among team members. It

should be clear, concise, and easily understandable by all team members, regardless of their experience level.

3. **Alignment with Design Principles**: The coding style should align with the overarching design principles of the system. This ensures that the code not only looks clean but also reflects the design intentions and user experience goals.

Design Tokens

Design tokens are reusable pieces of design information, such as colors, typography, and spacing, that are defined in a central location and can be accessed and reused across multiple projects. They are an essential part of modern design systems, providing a way to manage and maintain design values across different platforms and technologies.

The main purpose behind design tokens is to abstract design values *away from their application*, which means a single set of values can be used across web, mobile, and other platforms, regardless of the underlying technology.

For example, a design system might define a set of color tokens:

Design tokens for colors

```
$color-primary: #0088cc;
$color-secondary: #3d3d3d;
$color-success: #47b348;
$color-warning: #ffae42;
$color-error: #dc3545;
```

Design tokens can also define other aspects of design, such as font sizes, font families, line heights, spacing, and more.

Design tokens for font sizes and spacing

```
/* Font sizes */
$font-size-xs: 12px;
$font-size-sm: 14px;
$font-size-md: 16px;
$font-size-lg: 20px;
$font-size-xl: 24px;
```

```
/* Spacing */
$spacing-xs: 4px;
$spacing-sm: 8px;
$spacing-md: 16px;
$spacing-lg: 32px;
$spacing-xl: 64px;
```

One of the significant advantages of design tokens is their capacity to **simplify updates within the design system**. For instance, should a designer opt to modify the primary color, updating the respective design token (like `$color-primary`) will automatically reflect this change across all associated components and styles. This approach is much more efficient than manually adjusting each element.

In web applications, design tokens can be defined in various formats, such as SASS variables:

Using SASS variables to define design tokens

```
/* Colors */
$color-primary: #3498db;

/* Spacing */
$spacing-large: 20px;

/* Fonts */
$font-family-default: "Arial, sans-serif";
```

Or CSS custom properties (often referred to as CSS Variables):

Using CSS custom properties to define design tokens

```
:root {
  /* Colors */
  --color-primary: #3498db;

  /* Spacing */
  --spacing-large: 20px;

  /* Fonts */
  --font-family-default: "Arial, sans-serif";
}
```

Or even JSON or YAML.

Using JSON to define design tokens

```
{
  "color": {
    "primary": "#3498db"
  },
  "spacing": {
    "large": "20px"
  },
  "fontFamily": {
    "default": "Arial, sans-serif"
  }
}
```

Using YAML to define design tokens

```
color:
  primary: '#3498db'
spacing:
  large: '20px'
fontFamily:
  default: 'Arial, sans-serif'
```

For web projects, we'd need to use a build tool or script to convert design tokens defined in JSON or YAML into usable styles.

Using design tokens in a project involves more than just defining them. There are many different ways we can import and use design tokens in our app, which is dependent on how we've defined them in the first place.

Using design tokens in a SASS/LESS setting

For projects using SASS or LESS, which are preprocessor scripting languages that extend CSS with features like variables, nested rules, and mixins—tokens can be imported as variables and used throughout different stylesheets.

Importing design tokens in a SASS-based stylesheet

```
// Importing design tokens
@import 'path-to-your-design-tokens-file.scss';

.button {
  background-color: $color-primary;
  padding: $spacing-large;
  font-family: $font-family-default;
}
```

Using design tokens defined as CSS variables

If design tokens are represented as CSS variables within an application that's being worked on, these tokens are declared within the `:root` pseudo-class, which makes them globally accessible:

CSS Variables

```
:root {
  --color-primary: #3498db;
  --spacing-large: 20px;
  --font-family-default: "Arial, sans-serif";
}
```

Once defined, we can use these CSS variables throughout our styles with the var() CSS function.

Inserting the value of CSS custom properties

```
.button {
  background-color: var(--color-primary);
  padding: var(--spacing-large);
  font-family: var(--font-family-default);
}
```

Using design tokens within a React component

In a React environment, design tokens can be imported and used in both traditional CSS ways or with CSS-in-JS libraries such as Styled Components or Emotion.

For example, if we have our tokens as CSS variables, we can use them just like we'd do in regular CSS:

Using CSS variables to access design tokens

```
// Component.css
.button {
  background-color: var(--color-primary);
  padding: var(--spacing-large);
  font-family: var(--font-family-default);
}

// Component.jsx
import './Component.css';

function Button() {
  return (
    <button className="button">Click Me</button>
  );
}
```

In a CSS-in-JS approach, we can define our tokens as JavaScript objects and access them in our component in a CSS-in-JS manner.

Importing values of design tokens in JS

```
import styled from 'styled-components';
import { tokens } from './designTokens.js';

const Button = styled.button`
  background-color: ${tokens.colorPrimary};
  padding: ${tokens.spacingLarge};
  font-family: ${tokens.fontFamilyDefault};
`;

function App() {
  return <Button>Click Me</Button>;
}
```

The above only touches the surface of the many different ways design tokens can be created and used within the context of working with design systems.

In addition, third-party tools exist that can make the process of creating

and managing design tokens easier. Tools such as Theo by Salesforce can convert design tokens from JSON or YAML into various formats, including SASS, LESS, CSS custom properties, and even native formats for iOS and Android. Tools like Style Dictionary by Amazon can help generate platform-specific styles for iOS, Android, and more from a single set of tokens.

Component libraries

Another core element of design systems is component libraries: collections of pre-designed and pre-coded UI components that can be reused across various parts of a digital product or across multiple products. These libraries are often the building blocks for a web app's design by ensuring that every component adheres to the defined design system.

Modern JavaScript libraries and frameworks, like React, make it easy to create reusable components that can be easily shared and used across different projects by allowing us to build encapsulated components that use a combination of props and state to control behavior and rendering. For example, here's a simple React `Button` component that uses certain class names established in our design system and can be modified through props:

A `Button` component within a component library

```
import React from "react";
import "./Button.css";

function Button(props) {
  const { children, primary, ...rest } = props;

  return (
    <button
      className={`button ${
        primary ? "button--primary" : ""
      }`}
      {...rest}
    >
      {children}
    </button>
  );
```

```
}

export default Button;
```

The above `Button` component takes in a `primary` prop that controls whether or not the button is to be styled as a primary button (i.e., has the class of `button--primary`).

Elsewhere in our app, we can use the `Button` component to render the button element we expect to show from our component library.

Rendering different variations of the `Button` component

```
import React from "react";
import Button from "@design-system/Button";

function App() {
  return (
    <div>
      <Button primary>
        Click me. I am the primary button!
      </Button>
      <Button>But click me too.</Button>
    </div>
  );
}

export default App;
```

By building and using component libraries in this manner, we ensure consistency across our application, which saves time and effort in development and makes it easier to maintain and update the component library in our design system over time.

"If we think about design systems as a toolbox, we wouldn't encourage people to use a screwdriver for a nail or a hammer for a screw. [Additionally], the more complicated the tool, the more complex the quality control processes have to be on the manufacturer's side. In most cases, the best-designed tools are efficient but simple: they do a few things really well and don't try to do too much to the point where they're unreliable or cumbersome to use.

*The same can be said about design system components. Components that try to do too much or are too smart for their own good end up being the hardest to maintain and the most brittle tools in the system. On the other hand, baking too many opinions into a component's API can make it unnecessarily rigid. Building a design system is all about finding the right balance between competing priorities and adapting those calculations as new information comes up." [**Francine Navarro** | **UX Developer @ Assembled**]*

When using component libraries in a larger ecosystem, several considerations come into play:

Scalability of the component library

As projects grow and more components get added, the component library should be structured in a way that it remains maintainable. Organizing components into folders based on functionality or UI category, ensuring clear naming conventions, and maintaining comprehensive documentation are all vital.

Theming and customization

Oftentimes, the same component library can be used across different products under one brand umbrella. Consider supporting theming options in your component library to allow easy branding changes without altering the component logic.

Dependency management

It's important to always ensure that components within a design library have minimal dependencies. This not only reduces the library's size but also reduces the chance of dependency conflicts in projects using the library.

Accessibility, performance, and documentation are also important when working within and building component libraries and design systems. We'll discuss these considerations in the next few sections.

Accessibility

Ensuring accessibility in UI design, and subsequently in the elements within a design system, is paramount. UI components should be accessible to everyone, including those with disabilities. Accessibility guidelines such as Web Content Accessibility Guidelines (WCAG) help provide detailed information on how to create accessible web content.

An important aspect of accessibility is providing alternative text for non-text content such as images, videos, and audio. These descriptions aid assistive tools like screen readers in narrating the content to users who might have visual or auditory limitations. For pictures, the alt attribute can be employed to offer a concise description, which assistive technologies can then relay.

Using the `alt` attribute to describe an image

```
<img
  src="cat.jpg"
  alt="A fluffy white cat with green eyes"
/>
```

Besides alternate descriptions, utilizing semantic HTML plays a crucial role in fostering accessibility. Leveraging semantic HTML ensures that tools like screen readers can interpret and narrate the content in a user-friendly manner. Semantic HTML can be described as HTML written in a manner that gives meaning to the structure of web content. Here are two code snippets demonstrating semantic and non-semantic approaches to building a header:

Semantic and non-semantic approaches to building a header

```
<!-- Non-semantic heading -->
<div class="header">
  <div class="logo">Company Logo</div>
  <div class="title">Page Title</div>
</div>


<!-- Semantic heading -->
<header>
  <img src="logo.png" alt="Company Logo">
```

```
  <h1>Page Title</h1>
</header>
```

In the above code snippet, the initial block uses non-semantic tags, making it challenging for assistive tools to deduce the main content's intent. Conversely, the latter block embraces semantic HTML by clearly defining the content's structure by using the img and h1 elements to display an image and a title within a `header`.

The above only touches the surface of ensuring good accessibility in UI components. Many other practices can be done, such as:

- Ensuring sufficient contrast between foreground and background hues can be especially vital for those with vision challenges. Digital tools like WebAIM's Contrast Checker can validate if color combinations align with accessibility standards.

- Incorporating keyboard-friendly navigation for all user-interactive components. This ensures users who rely solely on keyboards can seamlessly navigate and interact with the interface.

- Refrain from integrating high-intensity, rapidly flashing content or swift color transitions, as these might trigger seizures in susceptible individuals.

The Accessibility Fundamentals section of W3C provides useful information on getting started with accessibility, while the Accessibility Guidelines provide a more detailed framework that encompasses a wide range of recommendations for making web content more accessible.

Performance

Components in design systems should be optimized for high performance since they're often reused across an application (or applications) and can be rendered multiple times on a page. Any performance issues or bottlenecks in these components can have a significant impact on the overall performance of the application, which would directly affect the user experience.

Implementing lazy loading for UI components can enhance the performance of the app that uses these components. This technique

ensures resources, like images or videos, are loaded only when necessary, boosting a page's initial loading speed. By loading resources on-demand, the browser can focus on displaying crucial content first. Similarly, this on-demand approach can be extended to UI components, loading them exclusively when called upon.

Utilizing images that are adapted for varied screen dimensions can further boost performance. When designing UI components that integrate or anticipate images, it's beneficial to present images suited to a user's device screen. This practice ensures browsers download the most fitting image dimension, shrinking the file size and accelerating an app's loading process.

Reducing the HTTP requests for UI component assets can further refine performance. Consolidating files, such as scripts or styles, into singular files can diminish the requests a browser sends to a server. Likewise, image sprites consolidate several images into one, minimizing the requests required for loading multiple images.

Documentation

Documentation is the backbone of a good design system and/or component library. It serves as a guidebook that provides clarity, context, and instructions on how to effectively use and integrate the components and design tokens present in the system.

The purpose and significance of clear documentation are multifaceted. It streamlines the onboarding process for new team members, minimizes miscommunication or potential misunderstandings related to component usage, guarantees consistent deployment across different projects, and serves as a touchstone for both designers and developers to ensure alignment.

Writing clear documentation encompasses several elements:

- **Component descriptions**: A brief overview of each component, its purpose, and when to use it.

- **Usage guidelines**: Step-by-step instructions on how to implement the component, with code snippets and visual references.

- **Props and API References**: A detailed list of all the properties a component accepts, their types, and their impact on the component's behavior and appearance.

- **Examples**: Real-world application of the component, showcasing different configurations or prop combinations.

- **Accessibility notes**: Specific details on how a component adheres to accessibility standards and any special instructions for ensuring consistent accessibility.

- **Versioning and changelog**: Information about different versions of components and the updates or changes made in each version.

Documentation that offers an interactive playground and search functionality allows developers to quickly find specific components or guidelines and tinker with component properties in real-time and see the result. Tools like Storybook have made this possible, allowing for an interactive and dynamic documentation experience.

Storybook is often used in conjunction with design systems to help developers create and test components in isolation. It provides a visual interface for building and testing components, allowing developers to see how their components will look and behave in different contexts without having to navigate through the full application. This can be especially useful for large teams, as it allows developers to work independently on different components while still ensuring that they all work together seamlessly.

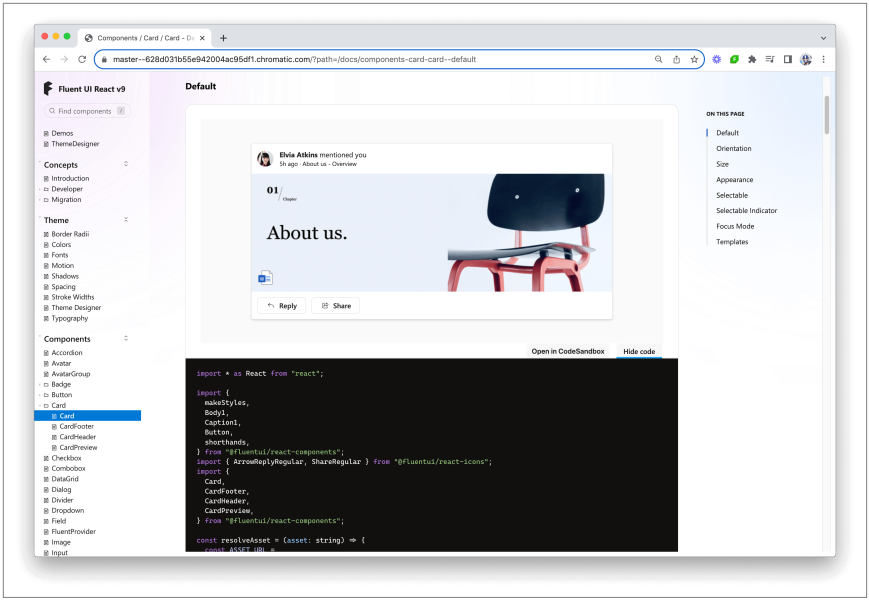


Figure 5-1. Storybook view of Microsoft's Fluent UI Card component (React)

Case Studies

Airbnb

Airbnb Design | Building a Visual Language

Airbnb Design | The Way We Build

In its rapid growth years, Airbnb faced challenges with guiding and leveraging a collective effort in designing and building software. As their design department expanded, consisting of a myriad of functions and outcome teams, it underscored the need for a more systematic approach to harness their collective potential.

To address these challenges, Airbnb formed a dedicated team of designers and engineers to develop a **Design Language System (DLS)**. The DLS aimed to establish a unified, universal, iconic, and conversational design language that would cater to Airbnb's global audience.

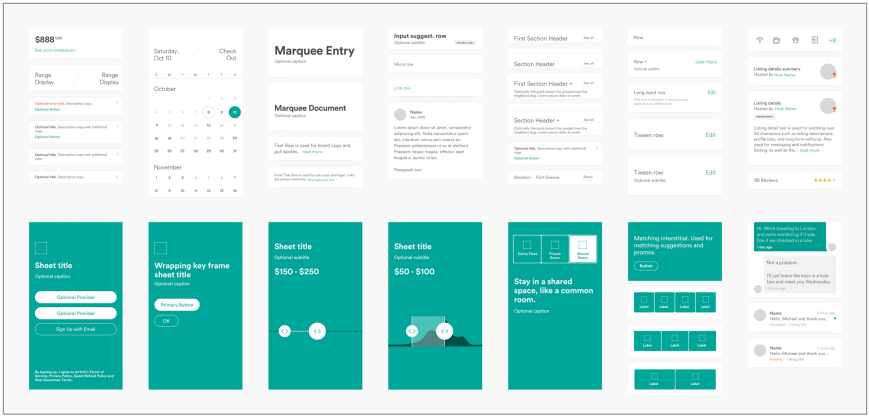


Figure 5-2. A sample of Airbnb DLS components from the airbnb.design article The Way We Build

Airbnb's DLS encompasses a systematic set of components seen as elements of a living organism. These components are not just static rules but an evolving ecosystem that adapts to the changing needs of the platform.

By prioritizing a unified design language, Airbnb was able to streamline design processes, enhance user experience, and foster better collaboration between designers and engineers. This approach not only improved the consistency across their digital platforms but also made product development more efficient and cohesive.

Government Digital Service (GDS) in the UK

GDS Podcast #16 | GOV.UK Design System

GOV.UK | Government Design Principles

During its digital transformation phase, the UK Government recognized the pressing need to provide user-friendly and accessible services to its citizens. Furthermore, the digital sprawl across various government bodies meant that there was a lack of uniformity and consistency in user experience.

In response, the Government Digital Service (GDS) took the lead in centralizing the digital strategy. GDS initiated a comprehensive

evaluation of current digital assets, aiming to pinpoint discrepancies and unify the diverse design elements. This resulted in the birth of the GOV.UK Design System.

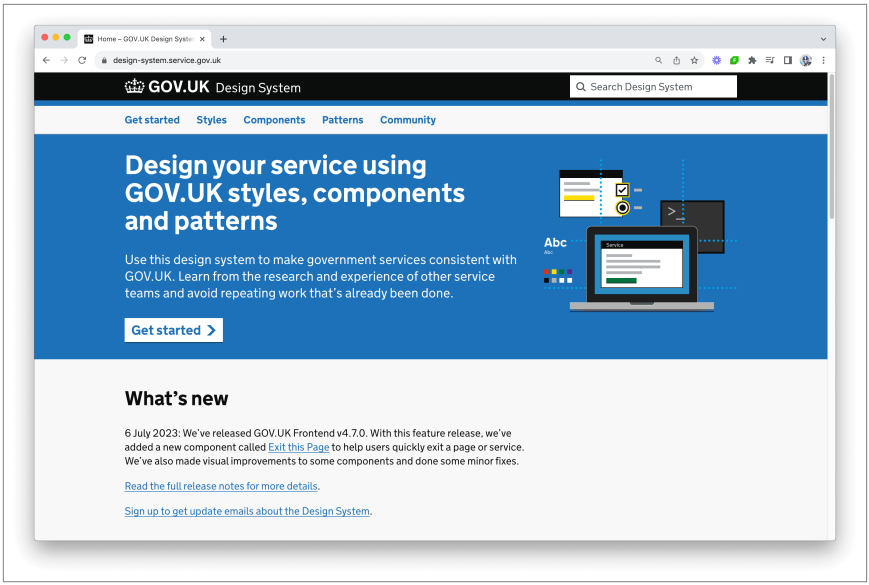


Figure 5-3. GOV.UK Design System

The GOV.UK Design System can be described as a combination of tools: the main Design System, the GOV.UK Prototype Kit for high-fidelity prototypes, and the GOV.UK Frontend. This frontend framework ties everything together, ensuring a cohesive experience across various services.

These services together help provide styles, components, and patterns to enable the creation of high-quality, accessible digital services. They are widely used across central and local government, with the design system's influence even reaching international governments.

Material

Google, recognized globally for its influence in shaping the digital landscape, identified the need for a unified design language across its array of products and services. As the ecosystem of Android apps and Google services grew, so did the diversity in user interfaces. This led to

the challenge of maintaining a consistent user experience across applications and even platforms (Android, Flutter, Web).

To address this, Google introduced Material Design in 2014—a design language that takes inspiration from the physical world and its textures to create interfaces that are intuitive, consistent, and responsive across devices.

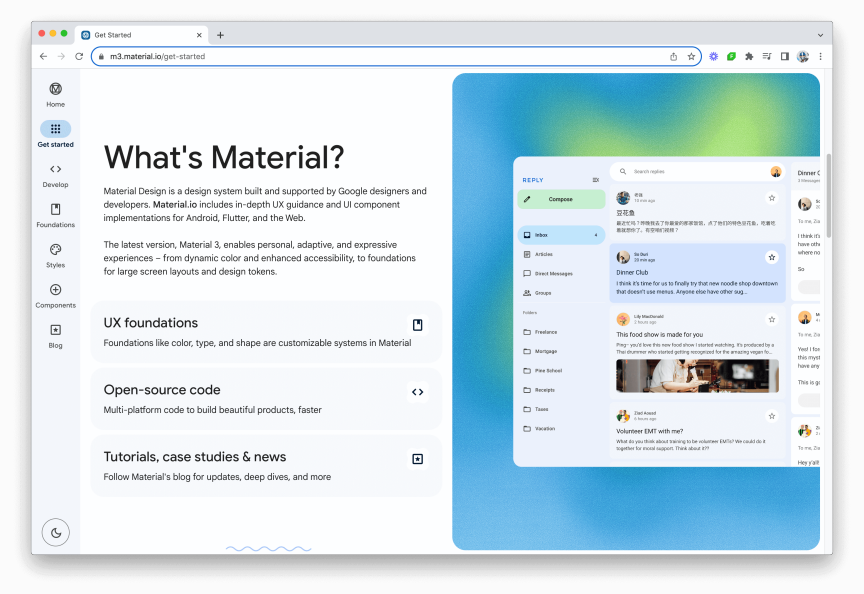


Figure 5-4. Material | Get Started

Material Design provides a comprehensive set of guidelines, resources, and tools. It defines everything from the basics like, typography, icons, and color schemes, to more complex components, like navigation drawers, cards, and animations.

A significant achievement of Material Design is its adaptability. Google ensured that Material Design is not just for web apps but also for Android and Flutter. This cross-platform design system helps developers and designers maintain consistency while still allowing for unique brand expression across different products.

Wrap up

In the ever-evolving landscape of web development, design systems have emerged as an invaluable tool. They streamline the development process, foster collaboration, ensure consistency, and enhance overall user experience. Whether it's through coding style guides that bring a unified codebase, design tokens that centralize and standardize design values, or component libraries that encapsulate reusable elements, design systems bridge the gap between designers and developers.

"Having a consistent suite of [design] products saves a company an immense amount of time and money. Nathan Curtis, a prominent speaker in the Design Systems space, published a great article titled 'And You Thought Buttons Were Easy?'.

In the article, he breaks down the cost of developing, designing, and testing one of the most foundational components: buttons. Once you create all the permutations of buttons (primary, secondary, tertiary, active, disabled, focused, active, icon, large, small, medium, etc.), the variations are staggering.

He goes on to explain that with all these button permutations, if you combine a designer, engineer, and tester to design, build, and test these buttons, assuming the staff costs $100 per hour of work and combined, it takes about 200 hours to design, build, and test these permutations, then buttons alone cost a company about $20,000. Well, if your company has more than one team creating buttons, let's say 50 teams, you're spending $1,000,000 to create buttons. This, he states, is why we need a design system.

*Design systems are incredibly powerful in the right circumstances. You likely don't need an entire system for creating a simple website for a restaurant, for example, but when you're in the business of building large-scale applications, a design system becomes invaluable." [**Emma Bostian | Engineering Manager @ Spotify**]*

As you forge ahead in your digital journey, whether you're building from scratch or integrating an existing system, remember that the primary goal is to enhance the user experience. The tools, guidelines, and components are simply means to that end. In the confluence of design and

development, a design system is your compass, guiding you and your team toward creating useful digital experiences for your users.

Data Fetching

Modern web applications are heavily reliant on data. Whether it's a social media feed, the latest stock prices, or a content management system, the heart of many client-side applications is data that typically resides on a server. As a result, having the client fetch this data from a server becomes a fundamental task when working in large-scale JavaScript applications.



Regardless of what form of data exchange happens between the client and the server (RESTful APIs, GraphQL, WebSockets, etc.), there are several approaches that can be taken in React and JavaScript applications to fetch this data from the server. This includes using built-in browser APIs (like the Fetch API), simple promise-based HTTP clients

(like Axios), or even more sophisticated query libraries like React Query or SWR.

Browser APIs and simple HTTP clients

Before diving into specialized libraries, it's worth noting that modern browsers already offer built-in tools to fetch data. One of the most prominent capabilities to achieve this is the Fetch API.

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global `fetch()` method that makes it easy to make web requests. Here's a basic example of using the Fetch API in a React component's `useEffect()` Hook.

Using the Fetch API to make a GET request

```
import { useEffect } from "react";

const App = () => {
  // ...

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch(
          "https://api.example.com/data",
        );
        if (!response.ok) {
          throw new Error(
            "Network response was not ok",
          );
        }
        const data = await response.json();
        console.log(data);
      } catch (error) {
        console.error(
          "There was a problem:",
          error.message,
        );
      }
    }
```

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

  // ...
};
```

Above, we're defining an asynchronous function inside the `useEffect` Hook called `fetchData()`. This function is responsible for making an HTTP GET request to a specified API endpoint. If the response is successful, we parse the JSON data from the response and log it to the console. Otherwise, we throw an error.

If there's a need for some additional capabilities like better error handling, adding request interceptors, or built-in Cross Site Forgery (XSRF) protection, we could leverage a minimal HTTP library like Axios that provides more features than just the standard Fetch API.

Using the Axios HTTP library to make a GET request

```
import { useEffect } from "react";
import axios from "axios";

const App = () => {
  // ...

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await axios.get(
          "https://api.example.com/data",
        );
        console.log(response.data);
      } catch (error) {
        console.error(
          "There was a problem:",
          error.message,
        );
      }
    }

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

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

While the Fetch API and Axios are somewhat flexible, they're quite low-level. As applications grow and requirements become intricate, developers might find themselves reinventing the wheel, writing repetitive code for things like error handling, caching, or retries. This is where more sophisticated data-fetching libraries come in.

More sophisticated data-fetching libraries

Specialized libraries like React Query and SWR have advanced the way developers approach data fetching in modern web applications. These libraries provide a set of tools for fetching, caching, and synchronizing data, which help abstract away many of the complexities associated with these tasks.

For this chapter, we'll use the React Query library to illustrate some of these more advanced patterns and techniques. It's important to note, however, that while we're focusing on React Query, the principles and discussions presented in the sections below are broadly applicable to other similar libraries. These libraries often share common features and philosophies, albeit with minor differences.

The following exploration is intended to not only highlight the specific capabilities of React Query but also to demonstrate the power and efficiency of modern data-fetching libraries in general.

Fetching data

With React Query, the useQuery Hook is at the heart of fetching data. The `useQuery` Hook accepts a distinct identifier and an asynchronous function that fetches the data from the API.

To begin using the `useQuery` Hook, we'll first need to initialize React Query in our React app. This provides the necessary context for our React components to access the data-fetching capabilities provided by React Query.

To initialize React Query, we'll import QueryClient and QueryClientProvider from React Query, create an instance of `QueryClient`, and wrap our React app with the `QueryClientProvider`.

Initializing React Query

```
import { createRoot } from "react-dom/client";
import {
  QueryClient,
  QueryClientProvider
} from "@tanstack/react-query";
import { App } from './App';

const rootElement = document.getElementById("root");
const root = createRoot(rootElement);

// Create a client
const queryClient = new QueryClient();

root.render(
  // Provide the client to your App
  <QueryClientProvider client={queryClient}>
    <App />
  </QueryClientProvider>
);
```

With React Query now initialized in our app, we can effectively use the `useQuery` Hook to fetch data and manage its state across our application. Here's an example of using the `useQuery` Hook to fetch a list of mock todo data from the publicly available https://jsonplaceholder.typicode.com/todos endpoint.

Using the useQuery Hook to fetch data

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

const fetchTodoList = async () => {
  const response = await axios.get(
    "https://jsonplaceholder.typicode.com/todos",
  );
  return response.data;
```

```
};

export function App() {
  const { data, isLoading, isError } = useQuery({
    queryKey: ["todos"],
    queryFn: fetchTodoList,
  });

  // ... further code to display/utilize the data
}
```

Notice in the declaration of the `useQuery` Hook we've supplied a unique key for the query, `'todos'` and an asynchronous function that triggers a GET request to the endpoint with the help of the `axios` library.

The `useQuery()` Hook returns a `result` object that contains different attributes reflecting the query's status and result. Besides holding the retrieved `data` upon a successful query, this `result` object also includes indicators like `isLoading` and `isError`. `isLoading` indicates the request's progress, while `isError` signals if any issues arose during the fetch.

We can use these query attributes to have our component render different UI depending on the state of the API request.

Rendering UI based on the returned values from useQuery

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

const fetchTodoList = async () => {
  const response = await axios.get(
    "https://jsonplaceholder.typicode.com/todos"
  );
  return response.data;
};

export function App() {
  const { data, isLoading, isError } = useQuery({
    queryKey: ["todos"],
    queryFn: fetchTodoList,
  });
```

```
  if (isLoading) {
    return <p>Request is loading!</p>;
  }

  if (isError) {
    return <p>Request has failed :(!</p>;
  }

  return (
    <div>
      <h1>Todos List</h1>
      <ul>
        {data.map(todo => (
          <li key={todo.id}>{todo.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

In the code above, the `App` component renders a loading message while the data is being fetched, and if there's an error in fetching the data, it displays an error message. Otherwise, it renders the fetched list of todos.

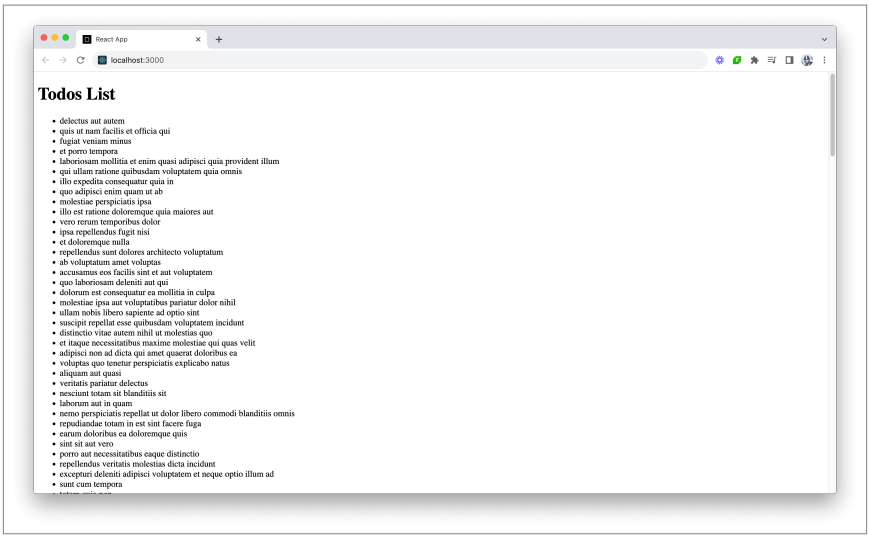


Figure 6-1. Fetched list of todos

With this example, we're able to see how the `useQuery` Hook provides built-in request states to manage the different phases of an API call when fetching data. In addition to this, the Hook provides a suite of additional capabilities that make it a powerful tool for developers. Let's dive deeper into some of these features:

Caching

One of the standout features of libraries like React Query is **caching**. Caching, in the context of data fetching, means storing fetched data so that future requests can be served faster by retrieving the data from the cache instead of making another network request. React Query uniquely manages this caching process not on the server or directly within the browser's local storage but in memory (i.e., its own JavaScript object/store).

React Query's caching mechanism is sophisticated, as it not only caches the data but also automatically invalidates and refetches it based on specific triggers or a set period. This ensures that the data users see is always fresh while also benefiting from the performance advantages of caching.

When using React Query, when a piece of data like the one below is fetched from the server, it is *automatically* stored in a cache.

Fetching data with the useQuery Hook

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

const fetchTodoList = async () => {
  const response = await axios.get(
    'https://jsonplaceholder.typicode.com/todos'
  );
  return response.data;
}

export function App() {
  const { data, isLoading, isError } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodoList
  });
```

```
  // ... further code to display/utilize the data
}
```

The next time a component needs this same data, it will be fetched from the cache, ensuring fast, immediate access, and in the background, a fresh request will be made to the server. If the server's data has changed, the cache will be updated, and the component will re-render to show the latest change.

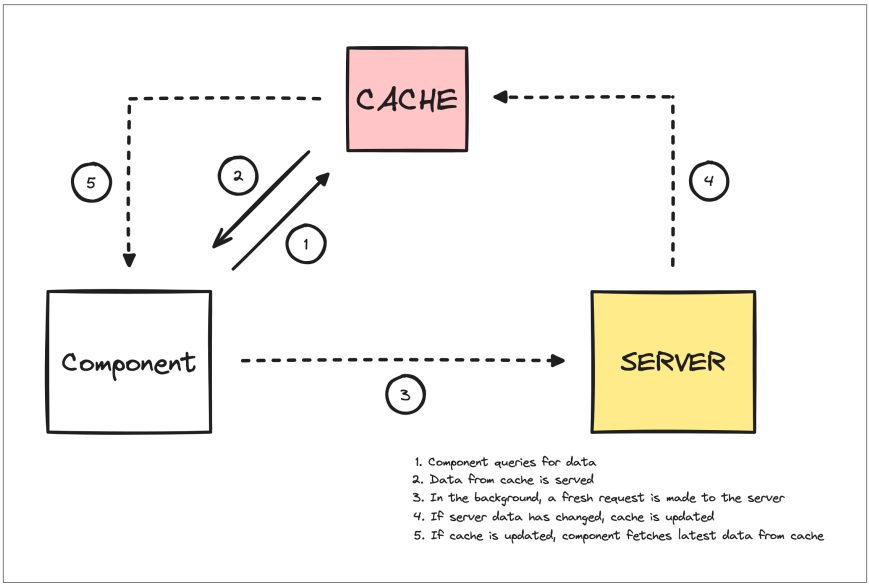


Figure 6-2. React Query's default cache behavior

Fetching data from the cache while still making a network request in the background for up-to-date data drastically reduces the visible loading times for the user. Information is automatically shown to the user, and if the network data differs from the cached version, the UI seamlessly updates with the freshest data.

staleTime

For every request we make with React Query, we're able to use the `staleTime` option to dictate the amount of time (in milliseconds) fetched data can be considered "fresh". By default, React Query sets this

option to a value of zero, but here's an example of us stating the fetched data of the following query will have a `staleTime` option of 1 minute.

Configuring the `staleTime` option

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

const fetchTodoList = async () => {
  const response = await axios.get(
    'https://jsonplaceholder.typicode.com/todos'
  );
  return response.data;
}

export function App() {
  const { data, isLoading, isError } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodoList,
    staleTime: 1000 * 60, // 1 minute
  });

  // ... further code to display/utilize the data
}
```

For the above query, when 1 minute hasn't passed after the initial query request, and another query is made, React Query will consider the data fresh, and a background request to fetch new data will not occur. Instead, the data will simply be fetched from the cache for the second request.

However, if the query request is made again and the staleTime of 1 minute has surpassed, React Query will recognize the data as stale. While it will still instantly serve the cached data to any component requesting it, a background request will be initiated to fetch the most recent version of the data from the server.

Updating the cache

React Query also provides a series of utilities to interact and update the cache directly. A common example of it being helpful to interact with the cache directly is when we may want to update the cache when a mutation

(i.e., a request to change data on the server) has been successful or when it has failed.

To handle mutations, React Query provides us with the useMutation Hook. Here's an example of triggering a mutation to add a new todo item to a list of todo items saved on our server when a button is clicked:

Triggering a mutation to add a new todo item

```
import {
  useMutation
} from "@tanstack/react-query";
import axios from "axios";

const createTodo = async (newTodo) => {
  const response = await axios.post(
    "https://jsonplaceholder.typicode.com/todos",
    newTodo,
  );
  return response.data;
};

export function App() {
  const { mutate, isLoading, isError } =
    useMutation({
      mutationFn: createTodo,
    });

  const triggerAddTodo = () => {
    mutate({
      title: "Groceries",
      description:
        "Complete the weekly grocery run",
    });
  };

  return (
    <div>
      <button onClick={triggerAddTodo}>
        Add Todo
      </button>
      {isLoading && <p>Adding todo...</p>}
      {isError && (
        <p>Uh oh, something went wrong!</p>
```

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

Like the `useQuery` Hook, the `useMutation` Hook returns the `isLoading` and `isError` properties to manage the state of our mutation request and give feedback to the user during the mutation process. The `useMutation` Hook also returns a `mutate()` function that, when called, triggers the mutation request.

When a mutation is successful, we might want to update our cache immediately so that any components depending on the related query data can have access to the latest information. One way we can achieve this is by using the query client's setQueryData function to update the cache for a certain query based on its key. We can trigger this function in the `useMutation` Hook's `onSuccess()` callback to ensure it's executed right after the mutation has been successfully completed.

Updating the cache after a mutation is successful

```
// ...
import { queryClient } from ".";
// ...

function App() {
  const { mutate, isLoading, isError } =
    useMutation({
      mutationFn: createTodo,
      // On success of the mutation
      onSuccess: (data) => {
        // Fetch the current todos from the cache
        const currentTodos =
          queryClient.getQueryData(["todos"]);

        // Update todos cache with the new todo
        queryClient.setQueryData(
          ["todos"],
          [...currentTodos, data],
        );
      },
    });
```

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

Using this method provides an optimistic update to our cache, making sure our UI is up-to-date with the latest data without having to make an additional fetch request.

This capability to update the cache on our own essentially allows us to update and manage the "state" being tracked and maintained by these data-fetching libraries. When data in one component changes due to a mutation, other components that rely on the same data can immediately reflect those changes, thanks to the centralized cache (i.e., global store). Furthermore, by being able to manually update the cache, developers gain more granular control over their application's state and user experience.

Interacting with the cache is a deep topic in and of itself. React Query comes configured with "aggressive but sane defaults" when working with the cache that can sometimes confuse new users. Be sure to take the time to go through the documentation on Important Defaults and Caching to understand these intricacies and make the most out of the library.

Prefetching data

Prefetching refers to the practice of proactively fetching data before it's actually needed, with the aim of having fresh data readily available when a user interacts with a UI element or navigates to a new screen. By prefetching data, we can oftentimes provide a more seamless user experience by reducing perceived loading times.

React Query offers a method called prefetchQuery that allows us to prefetch data for a specific query key. Once the data is prefetched, it's stored in the cache, and subsequent `useQuery` calls with the same query key can immediately access the cached data without refetching.

Here's an example of creating a function that, when triggered, prefetches the to-do list we saw in our earlier example.

Creating a function to prefetch a query when triggered

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";
import { queryClient } from ".";

const fetchTodoList = async () => {
  const response = await axios.get(
    "https://jsonplaceholder.typicode.com/todos",
  );
  return response.data;
};

const prefetchTodos = async (queryClient) => {
  /*
    Results of this query will be cached
    like a normal query.
  */
  await queryClient.prefetchQuery({
    queryKey: ["todos"],
    queryFn: fetchTodoList,
  });
};

// ...
```

The `prefetchTodos()` function can be triggered with event handlers, a component lifecycle event, or even before a user navigates to a specific section or page.

Placeholder data

There may be times when we have placeholder data we'd want to present to a user while the actual data from a query is still being requested. This can be beneficial in scenarios such as:

- **Dealing with slow network requests**: If a particular query takes a significant amount of time, showing a skeleton or a placeholder rather than showing a blank space or a loader can keep the user engaged.

- **Providing optimistic updates**: When performing mutations, placeholder data can give the illusion of instant updates while the actual request is processed in the background.

- **Ensuring a consistent layout**: It ensures that the layout remains consistent, avoiding jarring layout shifts when data is received, which can disrupt the user's experience.

In React Query, we can display placeholder data for a certain query with the placeholderData option.

Using the `placeholderData` query option

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

const fetchTodoList = async () => {
  const response = await axios.get(
    "https://jsonplaceholder.typicode.com/todos",
  );
  return response.data;
};

function App() {
  const {
    data = [],
    isLoading,
    isError,
  } = useQuery({
    queryKey: ["todos"],
    queryFn: fetchTodoList,
    placeholderData: [
      {
        id: "placeholder1",
        title: "Fetching...",
        completed: false,
      },
      {
        id: "placeholder2",
        title: "Fetching...",
        completed: false,
      },
    ],
  });
```

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

`placeholderData` helps to display initial placeholder data to the user without persisting this data in the cache. In instances where we want to display placeholder data while also having this data saved to the cache, we can use the initialData query configuration option.

Retry mechanism

By default, when a query request fails, React Query will automatically retry the query 3 times before displaying an error. We're able to configure this default at a global or individual query level. As an example, you can disable this retry mechanism completely for a certain query by setting the retry option to `false`:

Disabling query retries

```
const { data } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
  retry: false // This will disable any retries
});
```

Alternatively, we can infinitely retry query requests until the request is successful by setting the retry option to `true`.

Retrying query infinitely

```
const { data } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
  // Will retry until query is successful
  retry: true
});
```

Instead of completely disabling retries or triggering them indefinitely until successful, we'll often find ourselves configuring the number of retries to a certain number.

Setting the number of retry attempts to 10

```
const { data } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodoList,
  // Will retry 10 times before showing an error
  retry: 10
});
```

Configuring the retry mechanism can be useful in cases where our application may face intermittent network issues if the server is momentarily down due to maintenance or unexpected outages or if we expect certain query requests to fail more times than others.

Devtools

React Query comes equipped with a powerful set of developer tools that greatly enhance the debugging and development experience. The Devtools provide real-time insights into the status of queries, mutations, and the cache, making it easier to understand and debug the behavior of your application.

To use React Query Devtools, we need to first install them as a separate package:

Installing react-query-devtools

```
yarn add @tanstack/react-query-devtools
```

We'll then need to integrate the Devtools into our application by importing the `ReactQueryDevtools` component and placing it as high up in the component tree as possible.

Rendering the Devtools component

```
import {
  ReactQueryDevtools
} from '@tanstack/react-query-devtools';
// ...

// ...
```

```
root.render(
  <QueryClientProvider client={queryClient}>
    <App />

    {/* Placing Devtools in the root */}
    <ReactQueryDevtools initialIsOpen={false} />
  </QueryClientProvider>
);
```

Once integrated, React Query Devtools offer a range of features. At the core of its functionality is real-time query and mutation tracking. This provides a dynamic, constantly updated list of all ongoing and recent queries and mutations. For each query or mutation, we can view its status—whether it's loading, has encountered an error, or has been successful—alongside the data returned and any associated errors.

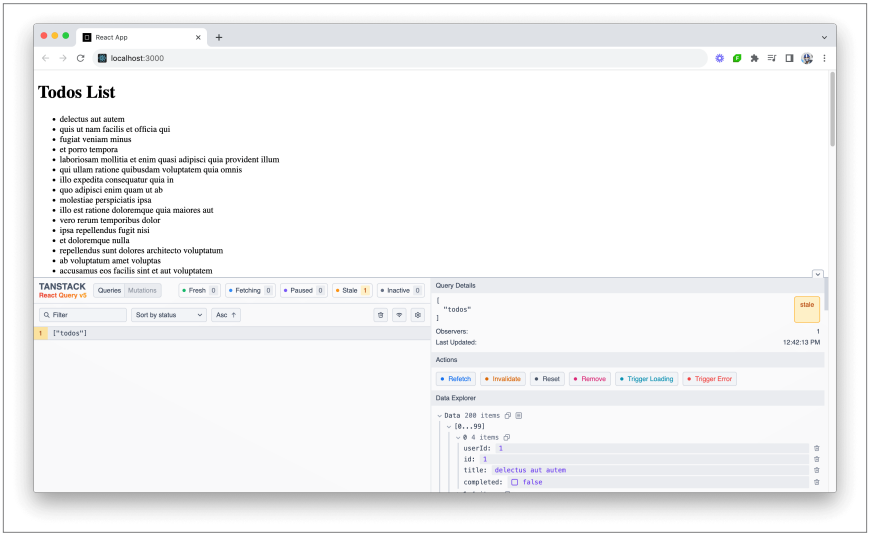


Figure 6-3. Details of a "todos" query

The Devtools also offer manual refetching and cache manipulation capabilities which is particularly useful during the development process, as it allows us to manually trigger query refetches or reset the cache. This enables us to test and debug our application's response to data changes more efficiently.

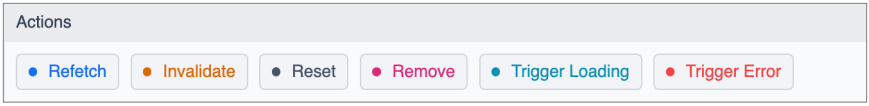


Figure 6-4. Manual triggers through Devtools

And a lot more...

The points we've discussed are just a fraction of the capabilities an advanced data-fetching library like React Query offers. Beyond the mentioned features, React Query also provides:

- Dependent Queries: This is where the execution of one query can be made dependent on the completion of another, ensuring that data is fetched in the right order.
- Parallel Queries: React Query can fetch multiple queries in parallel, ensuring faster data retrieval.
- Pagination and Infinite Queries: React Query supports both paginated and infinite queries out of the box, making it easier to handle large datasets.
- Server Rendering and Hydration support: Integration with server-side rendering frameworks (e.g., Remix or Next.js) where we're able prefetch data on the server and pass it to the client, which can then 'hydrate' the client-side React Query cache with this prefetched data.
- And a lot more!

In conclusion, advanced data fetching libraries like React Query empower us to handle complex data fetching, caching, and state synchronization with ease. This helps allow us to focus on building the user interface and enhancing the overall user experience without getting bogged down by the intricacies of data fetching and management.

Tips for efficient data-fetching

As applications grow and datasets become larger, it's important to employ strategies that optimize data-fetching processes. This not only

ensures optimal resource utilization but also provides a better overall experience for users.

Efficiently approaching data fetching for a large-scale JavaScript application requires careful consideration of the data needs of an application, as well as the performance characteristics of the application's data sources. Here are some strategies for efficiently approaching data fetching:

Design your data model carefully

The first step in efficiently approaching data fetching is to design your data model carefully. This means considering the data needs of your application and designing a data model that is optimized for those needs.

Some key principles to keep in mind when designing your data model include minimizing the number of data sources, optimizing data storage and retrieval, and designing your data model with scalability in mind. Though a lot of this work happens on the database level as opposed to the client, this work is an important precursor to any client-side fetching strategies. **An optimized server-side structure will inherently streamline the data-fetching process on the client side**.

Optimize your endpoints

From a client perspective, it's not just about how you fetch the data but also about how the server provides it. Optimizing your server endpoints to serve data efficiently is crucial. This could mean:

- Utilizing database indexes effectively.
- Reducing the number of database calls.
- Compressing the server responses.
- Filtering and limiting the response data on the server side.

Batch requests where applicable

Consolidating multiple smaller requests into a single batch request can sometimes reduce the overhead introduced by each individual request. This is especially useful when you know that a particular set of data will be required in close succession.

For example, instead of fetching user details and user orders in two separate requests, knowing that these two sets of information would always be fetched together, you can batch them into a single request, reducing the overall latency and processing time.

Prioritize and defer

Being judicious about what data to fetch and when to fetch it is crucial for maintaining a responsive application. Not every piece of data needs to be fetched immediately upon page load.

It can sometimes be helpful to identify which data is critical for the initial user experience and fetch that first. This ensures that users are not waiting unnecessarily for non-essential data before they can interact with the application. For data that isn't immediately necessary, consider using lazy loading.

Use lazy loading

Lazy loading is a technique that delays the fetching of data until it's needed, such as when a user scrolls to a particular section of a page or clicks on a specific feature.

For data that isn't immediately necessary, consider lazy loading this data. When dealing with large amounts of data, lazy loading can sometimes substantially improve the initial load time of a page, as the browser only needs to fetch and render a minimal amount of content at first.

Use caching to minimize data fetching

Caching is the technique for storing frequently accessed data in memory so that it can be retrieved more quickly than if it were retrieved from a data source each time. Earlier in this chapter, we've seen how libraries like React Query provide built-in caching mechanisms to optimize data fetching in React applications. The concept of caching doesn't have to remain only on the client since tools like Redis can be utilized on the server side to create a holistic caching strategy.

Some key principles to keep in mind when using caching include using appropriate cache expiration times, using caching for frequently accessed

data only, and implementing a cache invalidation strategy to ensure that cached data remains up-to-date.

Evaluate the use of GraphQL

GraphQL is a query language and runtime for APIs that provide a more efficient, flexible, and powerful alternative to the traditional REST approach. Instead of multiple endpoints with fixed data structures, GraphQL allows clients to request only the data they need, and in the shape they need it, from a single endpoint.

For example, let's assume we have a to-do list on our server, and we only want to retrieve `title` and `completionStatus` for each todo item.

With REST, we might hit an endpoint:

GET list of todos in REST

```
fetch('/todos')
```

This endpoint can return the `title` and `completionStatus` for each todo but also a lot of unnecessary information such as `description`, `creationDate`, `dueDate`, and more.

Mock response from REST endpoint

```
[
  {
    "id": 1,
    "title": "Buy groceries",
    "description": "Milk, Bread, and Eggs",
    "creationDate": "2024-01-01",
    "dueDate": "2024-01-11",
    "completionStatus": false
  },
  {
    "id": 2,
    "title": "Schedule dentist appointment",
    "description": "Remember to call Dr. Smith",
    "creationDate": "2024-01-03",
    "dueDate": "2024-01-03",
    "completionStatus": false
  },
```

```
  // ... more todos
]
```

With GraphQL, we can be more specific:

Query list of todos in GraphQL

```
{
  todos {
    title
    completionStatus
  }
}
```

This GraphQL query will only return the `title` and `completionStatus` of each todo, leaving out any other potential fields that each todo might have.

Mock response from GraphQL endpoint

```
{
  "data": {
    "todos": [
      {
        "title": "Buy groceries",
        "completionStatus": false
      },
      {
        "title": "Schedule dentist appointment",
        "completionStatus": false
      },
      // ... more todos
    ]
  }
}
```

With GraphQL, clients **get only what they ask for**. This can result in smaller, more efficient, and faster responses compared to traditional REST endpoints. For a deeper dive into GraphQL, be sure to read through the official GraphQL documentation.

Should we always use GraphQL?

While GraphQL offers many benefits from a client perspective, it's not a one-size-fits-all solution. GraphQL can introduce added complexity to your stack, especially if you're starting from scratch. If your data-fetching needs are straightforward, a REST API will suffice.

While GraphQL allows clients to fetch exactly what they need, this can also lead to inefficient queries if not managed correctly. Additionally, setting up a GraphQL server can sometimes introduce performance overhead not often seen when working with REST endpoints (e.g., Solving the N+1 Problem for GraphQL through Batching | Shopify Engineering).

Monitor and analyze performance

Regularly monitoring the performance of your data-fetching operations can provide insights into any inefficiencies or bottlenecks. Tools like Google Chrome's Network tab or Lighthouse tool can offer real-time metrics and optimization suggestions for enhancing the speed and reliability of your data-fetching processes.

For monitoring performance over extended periods of time, services like New Relic and Datadog can provide comprehensive analytics, alerting, and dashboard capabilities tailored for application performance monitoring.

State Management

In React and other modern JavaScript applications, components are akin to Lego blocks. They represent self-contained and reusable units of code responsible for rendering a piece of UI that encapsulates structure (HTML), behavior (JavaScript), and sometimes styles (CSS).

Just as every Lego block serves a unique purpose in building a Lego structure, every component serves a distinct purpose in an application.



Components come equipped with their own internal mechanism for managing state. This state represents any data that might change over the lifecycle of the component and has implications on the component's render or behavior. This concept of a component's 'local state' allows it to be reactive, adjusting its presentation and behavior in response to state changes.

With React, this is made possible with the useState Hook—a function Hook that allows functional components to maintain and update local state.

The `useState` Hook

```
function HelloWorld() {
  const [message, setMessage] = React.useState(
    "Hello World!",
  );

  return (
    <div>
      {/* display value of "message" */}
      <h1>{message}</h1>

      {/*
        When button is clicked, "message" state
        is updated and component re-renders to
        show new "message" value.
      */}
      <button
        onClick={() =>
          setMessage("Hello React!")
        }
      >
        Change Message
      </button>
    </div>
  );
}
```

When the local state of a component changes, React monitors this state, and the affected portions of the component are **re-rendered** to reflect that change.

Managing data between components

Components don't exist in isolation. They often have relationships that dictate how data flows. Typically, data is passed from a parent component to a child component. This is a "top-down" or

"unidirectional" flow, which is made possible with props—special parameters used to pass data from a parent to its child component.

A `HelloWorld` component that receives and uses props in its markup

```
function HelloWorld(props) {
  return (
    <div>
      <h1>{props.message}</h1>
      <button onClick={props.onChangeMessage}>
        Change Message
      </button>
    </div>
  );
}
```

Props provide a way for components to communicate, ensuring that children can receive and display data from their parent.

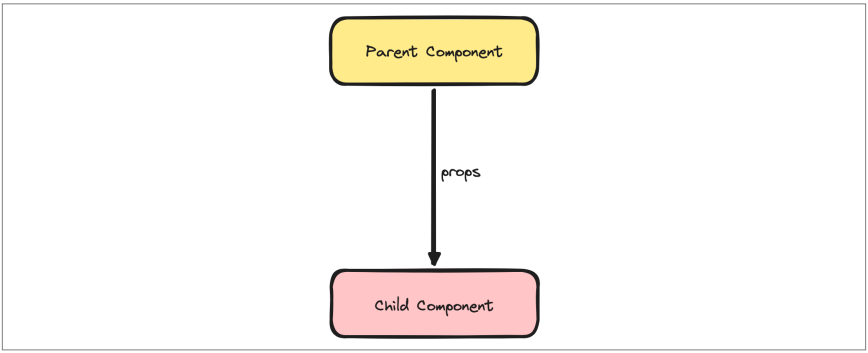


Figure 7-1. Passing data from parent to child with props

While props allow parent components to pass data down to child components, there are situations where child components need to communicate *back* to their parents. This is usually to signal that some action or event has taken place within the child component that the parent component needs to know about.

In React, this "upward" communication is commonly achieved using callback functions. The parent passes a function down to the child via props. The child then invokes this function to communicate back to the parent.

Passing a callback function down as props in React (Parent Component)

```
function App() {
  const [message, setMessage] = useState(
    "Hello World!",
  );

  const changeMessage = () => {
    setMessage("Hello React!");
  };

  // onChangeMessage() passed down as props
  return (
    <HelloWorld
      message={message}
      onChangeMessage={changeMessage}
    />
  );
}
```

Passing a callback function down as props in React (Child Component)

```
function HelloWorld(props) {
  return (
    <div>
      <h1>{props.message}</h1>

      {/*
          onChangeMessage() function comes from
          the parent
      */}
      <button onClick={props.onChangeMessage}>
        Change Message
      </button>
    </div>
  );
}
```

With child components able to signal to their parent that an action has taken place, this establishes a clear and effective means of communication between parent and child components.

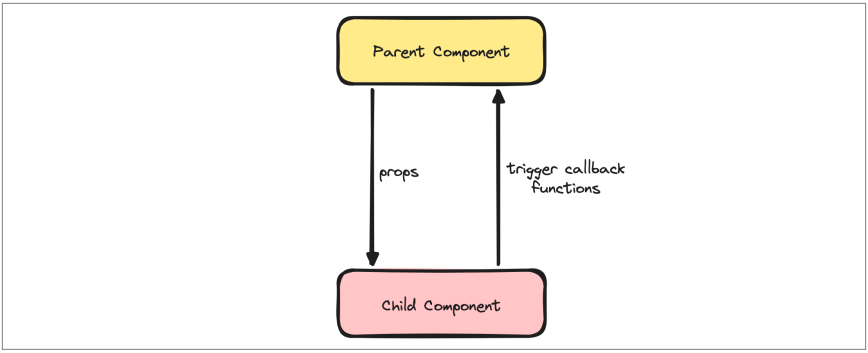


Figure 7-2. Triggering callback functions from the child

Prop Drilling

When working in large applications with a large number of components in the component tree, props can sometimes be hard to maintain since props need to be declared in *each and every component* in the component tree, even if intermediary components don't use the data themselves.

Consider a scenario where a top-level parent component holds a piece of state, and only a deeply nested child needs it. Instead of directly passing the state to the child, we would have to "drill" the state down through every level of component until it reaches the desired child.

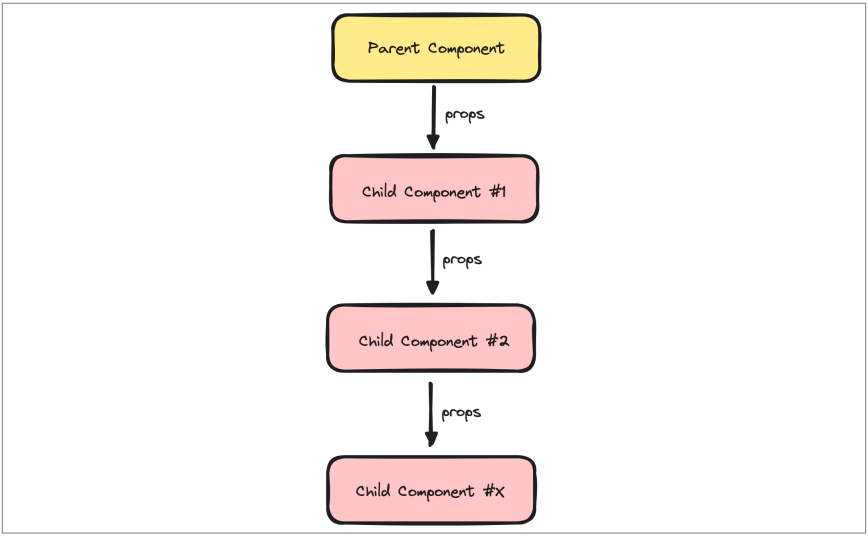


Figure 7-3. Prop drilling

Prop drilling, while functional, can lead to several challenges in large-scale React applications. This approach tends to make the codebase less maintainable, as any modification in the data structure requires changes across *all intermediary components*. This clutters components with unnecessary props, reducing readability and making it difficult for developers to track the flow of data.

The Context API in React offers a solution to this issue as it provides a way to share values between components without having to explicitly pass a prop through every level of the tree.

We can create a context object and then use the context Provider to wrap our components, making the data available to all nested components. Those components can then directly access the data with the useContext() Hook.

Passing data with context in React (Parent component)

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

// Create a context
const MessageContext = createContext();

function App() {
  const [message, setMessage] = useState(
    "Hello World!",
  );

  return (
    // Provide the state to nested components
    <MessageContext.Provider
      value={{ message, setMessage }}
    >
      <Child1>
        <Child2>
          <Child3>
            <DeeplyNestedChild />
          </Child3>
        </Child2>
```

```
      </Child1>
    </MessageContext.Provider>
  );
}
```

Passing data with context in React (Deeply nested child component)

```
function DeeplyNestedChild() {
  /*
    Use the data directly without receiving it
    as a prop
  */
  const { message, setMessage } = useContext(
    MessageContext,
  );

  return (
    <div>
      <h1>{message}</h1>
      <button
        onClick={() =>
          setMessage(
            "Hello from nested component!",
          )
        }
      >
        Change Message
      </button>
    </div>
  );
}
```

This pattern is most suitable for application-wide client data that numerous components can access—such as theme information, locale/language preferences, and user authentication details. These types of data are best managed with Context since any component within the application may require access to them at any given time.

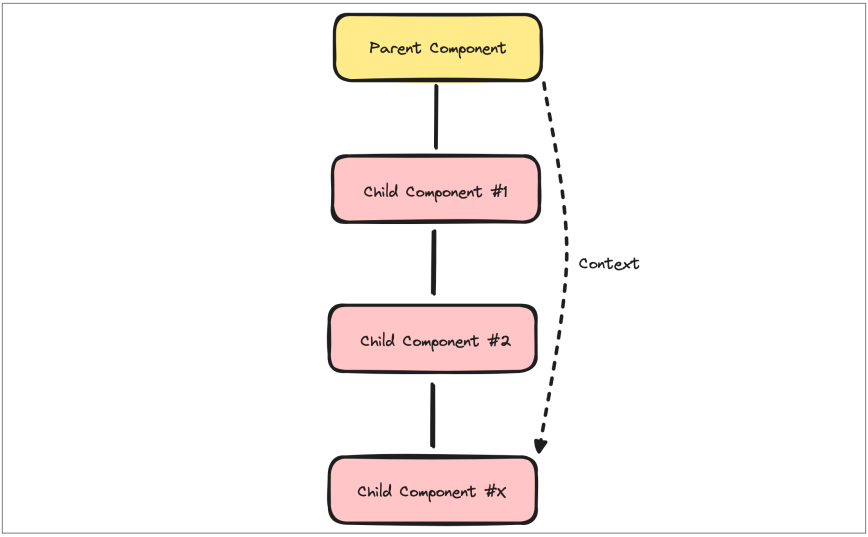


Figure 7-4. Context

Simple State Management

While the Context API in React is excellent for specific scenarios, it isn't necessarily optimal for managing all global or shared state across large applications. The context mechanism helps in avoiding prop-drilling and sharing values in a deep component tree, but for global state management, more sophisticated solutions can sometimes be required where state changes are made in a predictable manner.

The useReducer Hook

Before reaching out to more comprehensive solutions (which we'll discuss shortly), React developers can benefit from the useReducer Hook to manage more complex state logic in components. It works similarly to how reducers work in Redux but is contained either within a component or shared using Context.

The `useReducer` Hook

```
import React, { useReducer } from "react";

const initialState = {
  message: "Hello World!",
```

```
};

function reducer(state, action) {
  switch (action.type) {
    case "CHANGE_MESSAGE":
      return {
        ...state,
        message: action.payload,
      };
    default:
      throw new Error();
  }
}

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

  return (
    <div>
      <h1>{state.message}</h1>
      <button
        onClick={() =>
          dispatch({
            type: "CHANGE_MESSAGE",
            payload: "New Message!",
          })
        }
      >
        Change Message
      </button>
    </div>
  );
}
```

In the above code example, when the button in the component template is pressed:

1. The `dispatch` function is invoked with an action of type `CHANGE_MESSAGE` and a payload containing the new message “New Message!”.

2. This action is passed to the `reducer` function. Inside the `reducer`, the action type is checked, and when it matches the `CHANGE_MESSAGE` case, the `message` state property is updated.

3. Finally, the displayed message in the `h1` tag gets updated to "New Message!".

These additional steps are helpful for separating the logic of how state is updated from the UI components, which makes the management of data flow clearer and more predictable. With `useReducer`, state updates have descriptive action types, which help make it easier to trace *where* and *how* state changes occur.

Moreover, the use of `useReducer` promotes best practices in state management in React, such as ensuring state immutability by returning new state objects rather than modifying the current ones directly. This design pattern facilitates testing, debugging, and reasoning about state changes, especially in more complex components or applications.

Dedicated state management libraries

As applications grow in complexity, developers might find that simple state management solutions, such as React's `useReducer` Hook, might not suffice. The needs of larger applications often necessitate more powerful and versatile state management tools. This is where dedicated state management libraries come into play.

Redux

For React developers, Redux is a commonly reached-for library when more intricate state management is required. Redux centralizes the application's state, allowing for global access and mutations via dispatched actions. Two important concepts exist in Redux.

- **A Central Store**: Redux maintains the entire application state in a *single JavaScript object*, known as the store. This centralized store ensures a single source of truth for the application's state.

- **Immutable State**: In Redux, the state is never modified directly. Instead, every change is described as an action, and state updates are handled by pure functions called reducers.

Let's go through an example of a basic Redux setup. We'll use Redux Toolkit, a recommended toolkit that simplifies Redux development by providing utility functions for common Redux tasks. To get started, we'll need to install the Redux Toolkit package and the React-Redux bindings for use with React.

Installing Redux and Redux Toolkit

```
yarn add @reduxjs/toolkit react-redux
```

With Redux Toolkit, we're able to set up the central store by utilizing "slices" to handle actions and reducers in a more concise manner. In a `store.js` file, we can use the createSlice function from Redux Toolkit to handle how a `message` data property in our store can be updated in a `changeMessage` reducer.

In the same file, we'll import and use the `configureStore` function from Redux Toolkit to create the Redux store.

The `store.js` file

```
import {
  createSlice,
  configureStore,
} from "@reduxjs/toolkit";

const initialState = {
  message: "Hello World!",
};

export const messageSlice = createSlice({
  name: "message",
  initialState,
  reducers: {
    changeMessage: (state, action) => {
      state.message = action.payload;
    },
  },
});
```

```
export const store = configureStore({
  reducer: messageSlice.reducer,
});
```

To make the Redux store available to all components in the React app, we'll need to wrap the root component of the app with the Provider component from React Redux and pass it in the Redux store as a prop to the `Provider`.

Making the Redux store available to all components

```
import { createRoot } from "react-dom/client";
import { Provider } from "react-redux";

import { App } from "./App";
import { store } from "./store";

const rootElement =
  document.getElementById("root");
const root = createRoot(rootElement);

root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

With these changes in place, we can then have a React component leverage the Redux store to manage shared state. From the Redux Toolkit package, we can use the `useSelector` Hook to access state values and the `useDispatch` Hook to dispatch actions.

Managing Redux state from a React component

```
import React from "react";
import {
  useSelector,
  useDispatch,
} from "react-redux";
import { messageSlice } from "./store";

export function App() {
  const message = useSelector(
```

```
    (state) => state.message,
  );
  const dispatch = useDispatch();

  const handleChangeMessage = () => {
    dispatch(
      messageSlice.actions.changeMessage(
        "New Redux Message!",
      ),
    );
  };

  return (
    <div>
      <h1>{message}</h1>
      <button onClick={handleChangeMessage}>
        Change Message
      </button>
    </div>
  );
}

export default App;
```

The above React component displays a message and provides a button to change that message. All the data is managed by Redux, ensuring that state changes are predictable and easy to manage.

When the button is clicked, the `handleChangeMessage` function is invoked, triggering an action to be dispatched to the Redux store.

Once the `changeMessage` action is dispatched, the Redux Toolkit's `createSlice` function has already set up the corresponding reducer logic within the slice. The reducer function defined in the `createSlice` will handle the action. When it receives an action of type `changeMessage`, it will update the `message` property in the state with the new message provided in the action's payload.

After the state is updated, the `useSelector` hook in our `App` component will automatically receive the latest `message` from the Redux store. This is because `useSelector` subscribes to the Redux store and will re-render the component whenever the selected state

changes. Hence, the `<h1>` tag in the `App` component will display the updated message "New Redux Message!".

This process showcases the typical flow in a Redux application:

1. An action is dispatched, often as a result of some user interaction, like clicking a button.

2. The Redux store's reducer processes this action to produce a new state.

3. Components connected to the Redux store, like our `App`, are re-rendered with this new state.

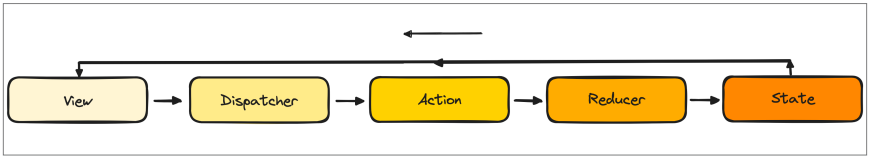


Figure 7-5. Redux Architecture

Though Redux adds more boilerplate to how application state is managed, it does so to provide a more structured and predictable approach to handling complex state interactions, ensuring that as applications scale, developers can maintain clarity and control over state changes.

In addition to a vast ecosystem of plugins, middleware, and utilities, Redux also comes with a powerful DevTools extension that allows for time-travel debugging, action inspection, and state tree visualization for debugging support. Though the devtools can be used as a standalone app or as a React component integrated in the client app, its most often installed and used as a browser extension (Chrome, Edge, and Firefox).

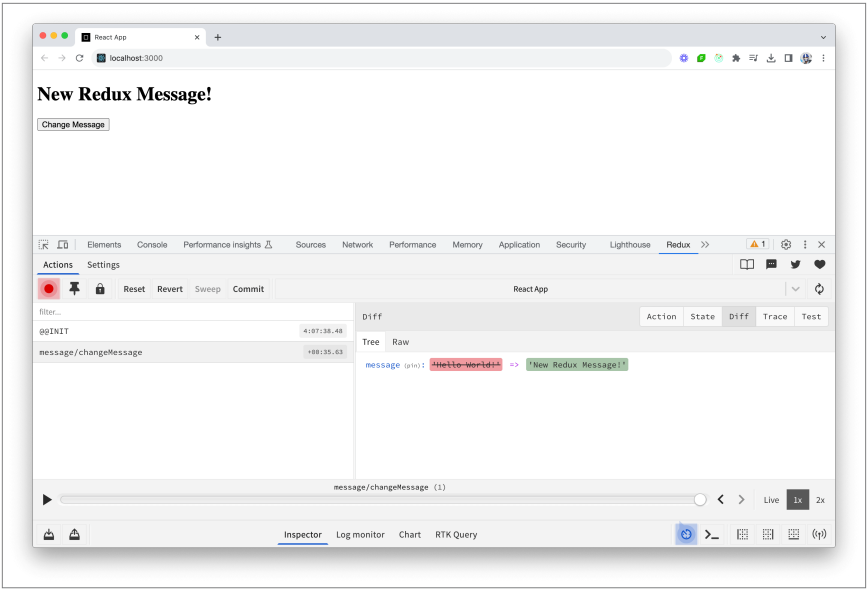


Figure 7-6. Redux Devtools

Redux isn't the only popular state management library available to developers; there are several others, like MobX and Zustand, each with its own unique features and approaches. Despite their differences, these libraries share many similarities towards the main goal of centralizing application state and providing structured mechanisms for updating and accessing that state.

Final Considerations

"While working across various companies and projects of different sizes and stages, I've observed Redux being used as the primary state management tool. However, Redux has often been more than necessary, frequently causing headaches in maintaining legacy code due to the middleware required to support asynchronous server-side logic.

Nowadays, with the advantages offered by powerful server-side rendering frameworks like Next.js, server-side state management is simplified, and complex client-side state management is rarely necessary. Drawing a clear distinction between client-side and server-side states has helped my teams build performant and maintainable state management architecture with React's

*built-in reducers and context providers." [**Jeffrey Peng | Software Engineer @ Doordash**]*

With all the different ways of managing application-level state, how do we determine which is the best approach to take? While there's no universal answer, the following checklist can guide you to find what's best for you and your team.

First, start with data-fetching considerations

Before diving into state management, determine the method of data fetching that your client application will utilize. Modern data fetching libraries such as SWR, React Query, and Apollo come equipped with built-in caching mechanisms, which we've discussed in the previous chapter—**Data Fetching**.

If your application's primary concern revolves around data fetching and the library you choose manages caching well, **you might not need to integrate a dedicated state management library**. Modern data fetching libraries often provide sufficient state management capabilities for handling server state by helping efficiently manage and cache server responses, reducing the need for additional state management layers.

Next, gauge the necessity for more robust custom state management solutions.

Even with a caching mechanism in place, does your application have global state needs that go beyond data fetching? Perhaps you have intricate workflows, mid-level computations, or an application state that needs to persist across sessions. This is where solutions like Redux can help.

Evaluate the merit of simpler state management tools.

If your application doesn't warrant the boilerplate of more robust state management libraries, consider lighter-weight solutions. React's Context API paired with `useReducer` can serve as a straightforward yet effective state management mechanism.

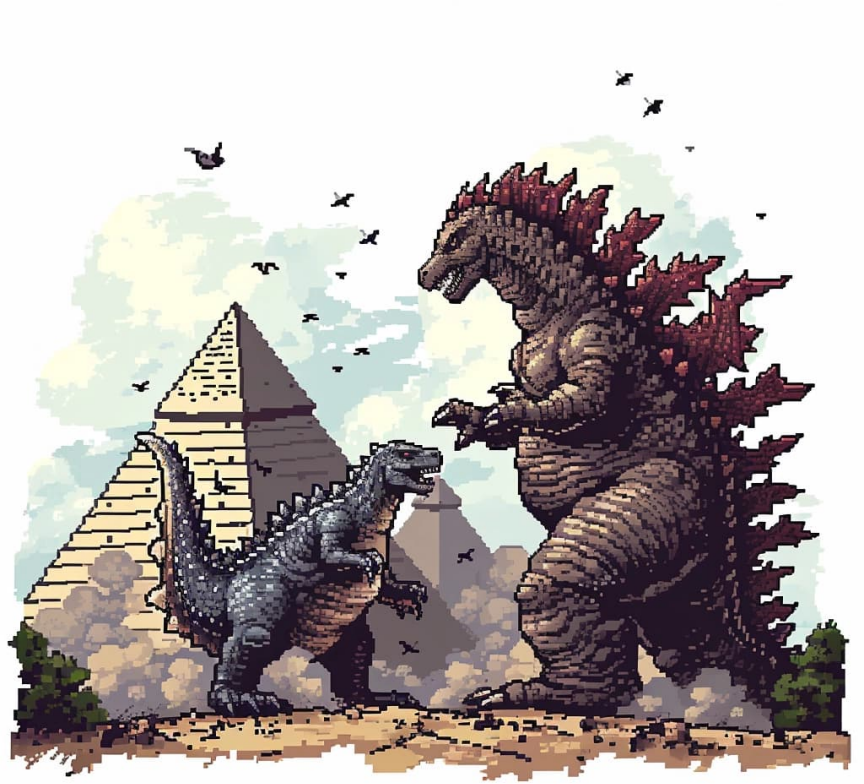
Finally, keep component state at the component level

Not every piece of state requires global management. Component-specific states, like form input values, toggle states, or even local UI animations, can be managed within individual components and shared with closely related components via props.

At the end of the day, the right state management solution depends on your application's unique requirements, the expertise of your team, and future scalability and performance considerations.

Internationalization

As the internet continues to connect people all over the world, it becomes increasingly important to design and develop web applications that cater to users from various regions and locales. This process is known as **internationalization**, and it ensures that our applications are accessible, functional, and culturally appropriate for users all over the world.



Internationalization, often abbreviated as i18n (where 18 represents the number of omitted letters between ‘i’ and ‘n’), is the process of designing and developing an application so that it can be easily localized to different languages, regions, and cultural contexts. It entails decoupling the user interface and content from the application’s codebase to make translation and adaptation easier.

Good internationalization practices lead to:

- **An increased user base**: By adapting our applications for different regions and locales, we can reach a larger global audience, increasing our user base and potential customer base.

- **An enhanced user experience**: By allowing users to access our applications in their preferred language and cultural context, we can provide a more personalized and engaging experience.

- **Compliance with regulations**: In some countries, there are legal requirements for applications to be accessible in the local language. For instance, under Canada's Official Languages Act, federal services (including digital applications) are required to be available in the nation's official languages—English and French.

In the following sections, we'll go over a few key points to keep in mind when attempting to achieve effective internationalization within an application.

Separate text and content from code

One of the most important things we can do to facilitate easy translation and adaptation of text in a large JavaScript application is to **extract all user-facing text strings from our code and store them in external resource files or databases**.

For example, instead of hardcoding a text string in our JavaScript or template code like this:

Hardcoding a text string

```
const greetingVariable = 'Hello, World!';
```

We can store it in a separate resource file like `en.json`, which denotes an English language translation file:

Storing a text string in an `en.json` file

```
{
  "greeting": "Hello, World!"
}
```

With our text strings stored in separate files, we can attempt to access them in our JavaScript/template by referencing the appropriate keys from our resource files or databases.

Accessing text from an en.json file through its key

```
const greetingVariable = {{ greeting }};
```

Once we've separated our text strings from the code and stored them in external resource files, we can easily create translation files for other languages. These translation files will contain the translated versions of the text strings.

For example, to create a translation file for another language, such as French, we can create a new resource file named `fr.json`. We can repeat this process for any additional languages we want to support, creating a new translation file for each language and providing the appropriate translations.

Storing a French-translated text string in a `fr.json` file

```
{
  "greeting": "Bonjour, le monde !"
}
```

Storing a Spanish-translated text string in an `es.json` file

```
{
  "greeting": "¡Hola, Mundo!"
}
```

Storing a German-translated text string in a `de.json` file

```
{
  "greeting": "Hallo, Welt!"
}
```

Each translation file follows the same structure, *with the same keys as the original resource file* (`en.json` in this case), but with translated values specific to the target language.

When incorporating variables within these text strings, we would often want to use placeholders that can be replaced at runtime. For instance, let's say we want to greet a user by name:

Storing a text string with a variable

```
{
  "greeting": "Hello, {{name}}!"
}
```

In our JavaScript or template code, we can interpolate the variable:

Template string interpolation

```
const userName = "Addy";
const greetingMessage = interpolate(
  "{{ greeting }}",
  { name: userName }
);

// greetingMessage === "Hello, Addy!"
```

The method `interpolate` above is a hypothetical function, and the exact implementation would depend on our setup. We're often able to replace placeholders like `{{name}}` with the actual value with utilities provided to us from third-party libraries built for localization. This leads us to the next section of our chapter.

Utilize third-party localization libraries

One of the easiest ways to implement i18n in a large JavaScript application is to use a third-party library like react-intl or i18next. Using a third-party library can save us a lot of time and effort compared to building our own i18n solution from scratch.

Once we've created the translation files for different languages, we can make use of these localization libraries to handle the loading and retrieval of translations in our application. These libraries provide

convenient methods and tools for managing translations (message formatting, pluralization, date/time formatting, etc.) and dynamically switching between different languages.

Let's go through a simple exercise of adding internationalization to an existing React application with the help of the react-intl library. While we'll use react-intl in our examples, the steps we discuss can be broadly applied to other similar libraries and scenarios.

As always, we start with installing the react-intl library:

Installing the react-intl library

```
yarn add react-intl
```

As discussed in the section above, it's often best to have our different language text content stored in separate configuration files:

src/locales/en.json

```
{
  "greeting": "Hello, User!"
}
```

src/locales/fr.json

```
{
  "greeting": "Bonjour, le monde !"
}
```

src/locales/es.json

```
{
  "greeting": "¡Hola, Mundo!"
}
```

src/locales/de.json

```
{
  "greeting": "Hallo, Welt!"
}
```

These files will act as our dictionaries, mapping message IDs to the message in the respective language.

Configuring the localization library

In the root instance of our app, we can wrap our app's component tree with the IntlProvider component, provided by react-intl, which will provide i18n context to all the components in our app. The `IntlProvider` component requires two props: `locale` and `messages`.

- `locale` is a string value that represents the language locale we want to use in our application.
- `messages` is an object that contains the actual text messages (strings) that we want to display in our application.

For the initial values we pass into the `IntlProvider` component, we can default the locale to English ('`en`') and reference the English message dictionary we've created.

Wrapping our app with IntlProvider

```
import { IntlProvider } from "react-intl";

import messages_en from "./locales/en.json";
import messages_es from "./locales/es.json";
import messages_fr from "./locales/fr.json";
import messages_de from "./locales/de.json";

const messages = {
  en: messages_en,
  es: messages_es,
  fr: messages_fr,
  de: messages_de,
};

export default function App() {
  const locale = "en"; // default locale

  return (
    <IntlProvider
      locale={locale}
```

```
      messages={messages[locale]}
    >
      {/* rest of our app goes here */}
    </IntlProvider>
  );
}
```

Rendering text and content

With react-intl now configured in this example situation, we can begin to use a component from react-intl (like the FormattedMessage component) to display text from our translation files.

Displaying text with `FormattedMessage`

```
// ...
import {
  IntlProvider,
  FormattedMessage,
} from "react-intl";

export default function App() {
  const locale = "en";

  return (
    <IntlProvider
      locale={locale}
      messages={messages[locale]}
    >
      <FormattedMessage id="greeting" />
    </IntlProvider>
  );
}
```

When saving these changes, we'll be presented with "Hello, User!" in the browser, as we're currently defaulting to the English locale.

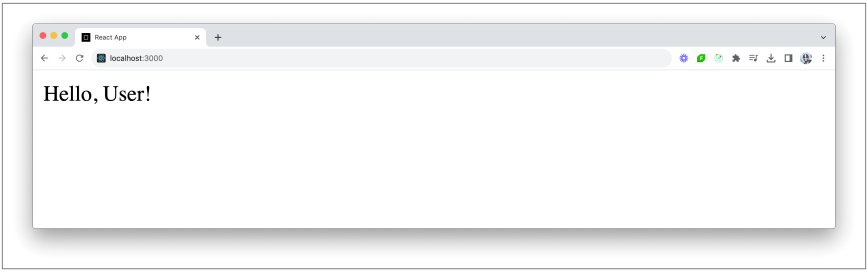


Figure 8-1. Rendering the English "greeting"

To make our application user-friendly for users who speak different languages or who are traveling in different countries, it's important to implement locale switching—the capability to allow users to switch between different languages and locales right within the application.

To implement locale switching in a simple manner from the example we have above, we can first store the locale in component state.

Storing locale in component state

```
import { useState } from 'react';
// ...


export default function App() {
  const [locale, setLocale] = useState('en');


  // ...
}
```

We can then add buttons to the app that allow the user to switch between the different locales available.

Triggering locale changes

```
import { useState } from "react";
import {
  FormattedMessage,
  IntlProvider,
} from "react-intl";
// ...
```

```
export default function App() {
  const [locale, setLocale] = useState("en");

  return (
    <IntlProvider
      locale={locale}
      messages={messages[locale]}
    >
      <FormattedMessage id="greeting" />
      <div>
        <button onClick={() => setLocale("en")}>
          English
        </button>
        <button onClick={() => setLocale("es")}>
          Español
        </button>
        <button onClick={() => setLocale("fr")}>
          Français
        </button>
        <button onClick={() => setLocale("de")}>
          Deutsch
        </button>
      </div>
    </IntlProvider>
  );
}
```

When we click on a certain button, the locale state of our app updates to the selected locale, triggering a re-render of the `IntlProvider` component with the new locale value and data. Consequently, the text in our application that's managed by the react-intl library will be updated to reflect the chosen language.

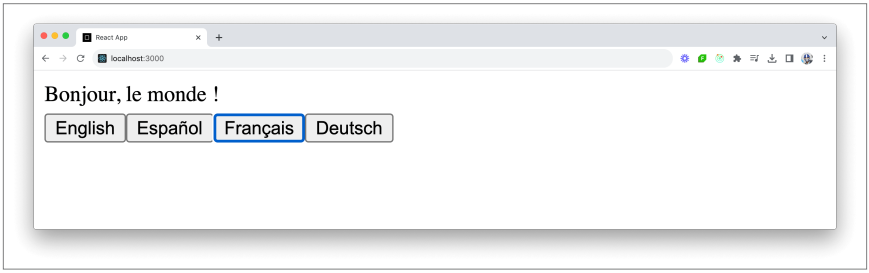


Figure 8-2. Rendering the French "greeting"

Dynamic loading

In the code example above, we're directly importing all language files in the root of our app, which can sometimes result in having all of these files included in the main bundle. While this may be acceptable for applications with a minimal number of relatively small translation files, this could lead to performance concerns for larger applications with extensive and numerous translation files.

To mitigate this concern, translation files can sometimes be dynamically imported or loaded, ensuring that only the necessary files are loaded based on user preference or browser settings. One way we can achieve this is by using dynamic imports (i.e., the `import()` function) to dynamically import the required translation file.

Dynamically importing locales

```
import React, { useState, useEffect } from "react";
import {
  IntlProvider,
  FormattedMessage
} from "react-intl";

export function App() {
  // setting default locale in initial state
  const [locale, setLocale] = useState("en");
  const [messages, setMessages] = useState({});

  useEffect(() => {
    /*
      Dynamically import required translation file
      based on locale
    */
    const loadMessages = async () => {
      switch (locale) {
        case "en":
          await import(
            "./locales/en.json"
          ).then((m) => setMessages(m.default));
          break;
        case "es":
          await import(
```

```
            "./locales/es.json"
          ).then((m) => setMessages(m.default));
          break;
        // and so on...
      }
    };

    loadMessages();
  }, [locale]);

  return (
    <IntlProvider locale={locale} messages={messages}
>
      {/* ... */}
    </IntlProvider>
  );
}
```

In the example above, we're using the useEffect Hook to watch for changes in the `locale` state. When `locale` changes, the Hook helps dynamically import the required translation file with the import() function. Since the `import()` function is a promise, we use `.then()` to update the `messages` state once the translation file has been successfully imported.

Keep in mind that the above is a simple example to illustrate only the capability of how dynamic imports can sometimes be achieved. Depending on the build tooling of your app and what module bundler you may be using, such as Webpack or Vite, the exact implementation or configuration might differ. Some bundlers support dynamic imports out of the box, while others may require additional plugins or configuration settings. We discuss dynamic imports and code-splitting in more detail in the chapter—**Modularity**.

The pattern of steps we've taken to enable localization and internationalization is generally applicable regardless of the third-party library being used. The typical workflow involves:

- Installing the desired library.
- Defining language messages in distinct configuration files.
- Configuring the library at the root instance of the application.
- Rendering content using components or functions provided by the library.

- Finally, locale switching is implemented to allow user-friendly language transitions.

Handling plurals across languages

When building a web application for a global audience, it's important to recognize that simple pluralization rules, like adding an 's' to denote more than one in English, don't apply universally. Different languages have distinct rules for plurals, and some can be more complex than others. For example, the Arabic language often has different plural rules for quantifying zero, one, two, few, many, etc.

Third-party libraries like react-intl provides tools for handling plurals. The `FormattedMessage` component we used earlier has a plural format to help handle multiple plural forms for different languages.

Assume we wanted to display the message "You have {itemCount} items" in our app. In the English locale, we'll want to display three separate sentences to convey this:

- When `itemCount` is 0: **You have no items**
- When `itemCount` is 1: Y**ou have 1 item**
- When `itemCount` is more than 1: **You have {itemCount} items**

When attempting to use the `FormattedMessage` component's built-in plural format, we can first define the plural rules for the English locale:

Defining the plural rules for the item count message

```
{
  "itemCountMessage": "{itemCount, plural, =0
  {No items} one {1 item} other {You have
  {itemCount} items}}"
}
```

Here, we're using react-intl's ICU Message Syntax that builds on ICU Message Formatting—a standardized way to build and format messages in software, especially in the context of internationalization and localization.

For a different locale that has more plural rules like Arabic, we may want to structure our sentences like the following:

- When `itemCount` is 0: **ليس لديك أي عناصر** (You don't have any items)
- When `itemCount` is 1: **لديك عنصر واحد** (You have one item)
- When `itemCount` is 2: **لديك عنصرين** (You have two items)
- When `itemCount` ranges from 3 to 10: **لديك ٥ عناصر** (You have five items)
- etc.

This will lead to more intricate plural rules when defining a similar message in the Arabic locale:

Defining the plural rules for the item count message in Arabic

```
{
  "itemCountMessage": "{itemCount, plural, =0
  {ليس لديك أي عناصر} one {لديك عنصر واحد}
  two {لديك عنصرين} few {{itemCount} عناصر ديك}
  many {لديك العديد من العناصر} other
  {{itemCount} عنصرًا}}"
}
```

When using the `FormattedMessage` component in our React application, we can then retrieve the appropriate message based on the user's locale and the specific item count.

Rendering the item count message

```
import { FormattedMessage } from 'react-intl';

function PluralComponent({ itemCount }) {
  return (
    <FormattedMessage
      id="itemCount"
      values={{ itemCount }}
    />
  );
}
```

The `FormattedMessage` component will now render the correct translation based on the `itemCount` and the active locale. If `itemCount` is 2 and the active locale is Arabic, for instance, the component will display "لديك عنصرين."

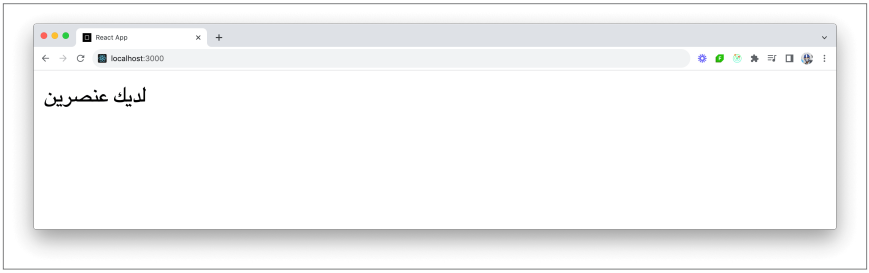


Figure 8-3. Rendering the Arabic text for "You have two items"

Format dates, times, and numbers

Dates, times, and numbers often have different formats across locales. As a result, we should always make use of localization libraries or built-in language features available in our programming language to ensure accurate and localized formatting of dates, times, and numbers.

In JavaScript, the Intl object provides a set of features for formatting dates, times, and numbers according to different locales. Here's an example of formatting a date in the `en-US` locale.

Formatting dates with the Intl JavaScript object

```
const date = new Date();
const options = {
  year: "numeric",
  month: "long",
  day: "numeric",
};
const formattedDate = new Intl.DateTimeFormat(
  "en-US",
  options,
).format(date);

console.log(formattedDate);
// Output: June 30, 2023
```

Here's another example of using the `Intl` object to format numbers differently between the United States and Germany (i.e., the `de-DE` locale).

Formatting numbers with the Intl JavaScript object

```
const number = 1004580.89;
const formattedNumberDE = new Intl.NumberFormat(
  "de-DE",
).format(number);
const formattedNumberUS = new Intl.NumberFormat(
  "en-US",
).format(number);

// Output: 1.004.580,89
console.log(formattedNumberDE);

// Output: 1,004,580.89
console.log(formattedNumberUS);
```

Third-party libraries, like the react-intl library, also provide components and an API to format dates, times, and numbers. Below is an example of how we can use the FormattedDate component from the react-intl library to help render a formatted date in our component.

Formatting dates with the react-intl FormattedDate component

```
import { FormattedDate } from "react-intl";

// ...

// Rendering component
<FormattedDate
  value={new Date()}
  locale="en-US"
/>
```

For a `new Date()` value of `Mon Dec 25 2023`, the above will output a formatted date in the `en-US` locale format of mm/dd/yyyy.

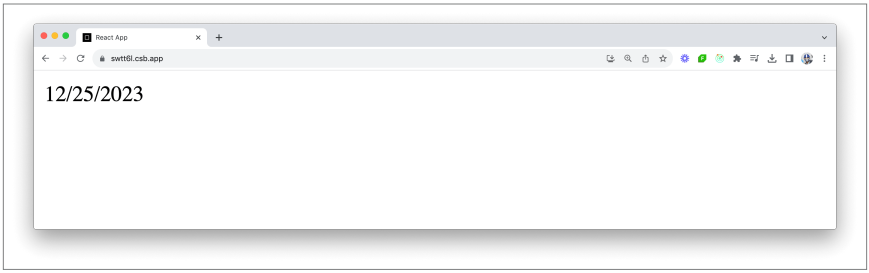


Figure 8-4. Formatted date in the `en-US` locale

If we instead want to render the date in the United Kingdom format of dd/mm/yyyy, we can specify a locale of `en-GB`.

Formatting date in the `en-GB` locale

```
import { FormattedDate } from "react-intl";

// ...

// Rendering component
<FormattedDate
  value={new Date()}
  locale="en-GB"
/>
```

We'll then be presented with a formatted date in the `en-GB` locale format of dd/mm/yyyy, reflecting the standard date presentation in the United Kingdom.

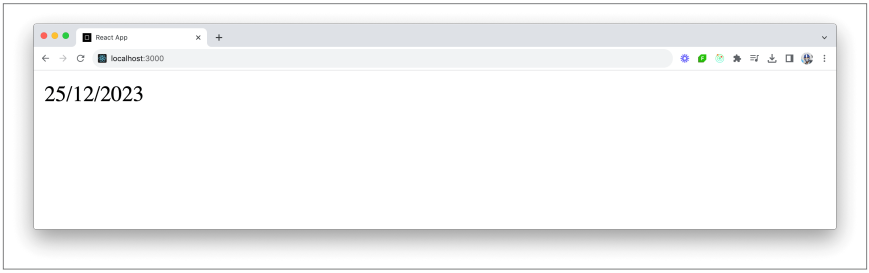


Figure 8-5. Formatted date in the `en-GB` locale

The `FormattedDate` component also allows us to render dates in longer or more descriptive formats beyond the standard numeric

representations. This is more useful for contexts where a more verbose date format is preferred, such as in formal documents, user interfaces that emphasize readability, or when the day of the week is important.

Rendering a long format date

```
import { FormattedDate } from 'react-intl';

// Rendering component
<FormattedDate
  value={new Date()}
  locale="en-US"
  weekday="long"
  year="numeric"
  month="long"
  day="numeric"
/>
```

For a new `Date()` value of `Mon Dec 25 2023`, the above would output something like "Monday, December 25, 2023" in the `en-US` locale.

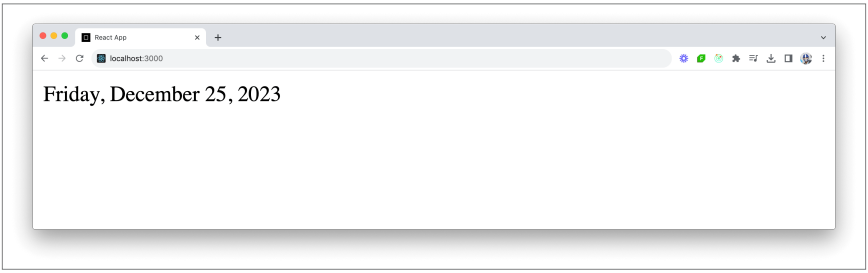


Figure 8-6. Long formatted date in the `en-US` locale

In addition to dates, `react-intl` also offers components like FormattedTime and FormattedNumber for localizing times and numbers, respectively. Here's an example of using the FormattedNumber component to format numbers differently between the United States and Germany.

Formatting numbers with the react-intl FormattedNumber component

```
import { FormattedNumber } from "react-intl";

// ...
```

```
// Rendering component for German locale
<FormattedNumber
  value={1004580.89}
  style="currency"
  currency="EUR"
  locale="de-DE"
/>

// Rendering component for US locale
<FormattedNumber
  value={1004580.89}
  style="currency"
  currency="USD"
  locale="en-US"
/>
```

The above will render the same number twice in a different currency and locale.



Figure 8-7. Formatting the same number in different currencies/locales

The react-intl library also provides the useIntl Hook to allow us to imperatively format dates, times, and numbers outside of the context of rendering JSX and components. For a detailed list of what the react-intl library offers, be sure to check their API and Components documentation.

Consider Right-to-Left (RTL) Languages

In addition to supporting different languages, it's essential to consider the layout, CSS styles, and text alignment for languages that are written from right to left (RTL), like Arabic, Hebrew, and Urdu. Adapting our

application's layout and text direction for RTL languages is important for providing a seamless experience to users in those regions.

Text direction (`dir`)

HTML contains a dir property that we can utilize to handle RTL text. By setting the `dir` property to `rtl` (or `auto`, which lets the user agent decide), we can ensure that text and elements within our application are correctly aligned for RTL languages.

The HTML `dir` attribute

```
<p dir="ltr">
  This paragraph is in English and correctly
  goes left to right.
</p>

<p dir="rtl">
  هذه الفقرة باللغة العربية ، لذا يجب الانتقال
  من اليمين إلى اليسار.
</p>
```

In the above code example, we align the English and Arabic text in their appropriate alignment.

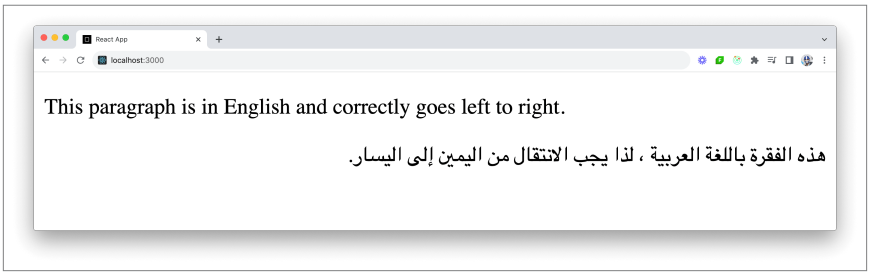


Figure 8-8. Handling RTL text with the HTML `dir` attribute

Text align (`text-align`)

When setting text alignment with the text-align CSS property, there exists the convenience of using values like `start` and `end` in addition to the traditionally used values of `left` and `right`.

The `start` and `end` values help align text based on the document's directionality. For instance, in an LTR (Left-to-Right) context, `text-align: start;` would align text to the left, while in an RTL context, it would align text to the right.

Aligning text based on document directionality

```
/*
  This will align text to the left when
  dir="ltr" and to the right when dir="rtl".
*/
.text-start {
  text-align: start;
 }

/*
  This will align text to the left when
  dir="rtl" and to the right when dir="lrt".
*/
.text-end {
  text-align: end;
}
```

In our HTML, we'll be able to use the above classes for dynamic alignment. As an example, assume we wanted to have the English and Arabic texts below aligned at the end of the document:

Aligning text at the end of the document

```
<p dir="ltr" class="text-end">
  This paragraph is in English and is aligned
  at the end.
</p>

<p dir="rtl" class="text-end">
  هذه الفقرة مكتوبة باللغة الإنجليزية وتمت
  محاذاتها في النهاية.
</p>
```

With the `text-align: end` property, we're able to see how each of the texts is aligned at the end of the document without relying on stating directionality like left or right.

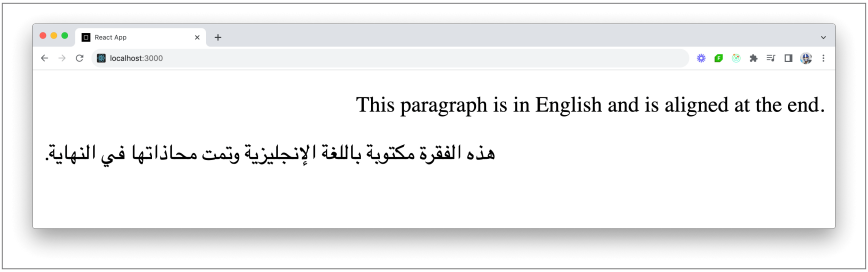


Figure 8-9. Aligning text with the `text-align` *CSS property*

Font

When designing for RTL languages, font selection becomes important. The fonts we select should not only be visually appealing and align with our brand but also adequately support the characters and nuances of the RTL scripts. Some fonts might lack certain characters or have incomplete support, which could lead to a broken or inconsistent user experience.

It's always best practice to provide a list of fallback fonts in our CSS. If the primary font fails to load or a particular character isn't supported, the browser will attempt to use the next font in the list.

Fallback fonts

```
body {
  font-family: 'Noto', 'Monotype SST', '...';
}
```

Furthermore, we can take this a step further by providing a series of fallback fonts for different language directions or different languages.

Fallback fonts for different languages

```
/* Default font stack */
body {
  font-family: 'Noto', 'Monotype SST';
}

/* Font stack for Arabic language content (RTL) */
body[lang="ar"] {
  font-family: 'Noto Naskh Arabic', 'Tahoma';
}
```

```
/* Font stack for Hebrew language content (RTL) */
body[lang="he"] {
  font-family: 'Frank Ruhl Libre', 'Arial Hebrew';
}
```

By setting up language-direction specific font stacks, we cater to the nuances and unique characters of each script, ensuring the text is both legible and aesthetically fitting with our app's design principles.

Layout

When working with RTL languages, we shouldn't only be concerned about text direction and alignment. Other CSS properties, like margin, padding, and positioning, can also be affected by the change in directionality.

As an example, assume we had a piece of text written in English and positioned alongside an icon on the left side in an LTR context. To have some spacing between the icon and the text, we can use the standard margin-right CSS property to achieve this.

Icon positioned to left of text in English (HTML)

```
<div class="icon-text-ltr">
  <img
    src="icon.png"
    alt="Icon"
    class="icon-ltr"
  />
  <p>
    This is some English text with an icon on
    the left.
  </p>
</div>
```

Icon positioned to left of text in English (CSS)

```
.icon-text-ltr {
  display: flex;
  align-items: center;
}
```

```
.icon-ltr {
  width: 20px;

  /* Icon is to the left of the text */
  margin-right: 10px;
}
```

This will have an icon placed on the left of the English text, with the margin appropriately applied to the right of the icon.



Figure 8-10. Using the `margin-right` CSS property

Upon translating the text to Arabic (or any other RTL language), the icon margin will still be applied to the right of the icon.

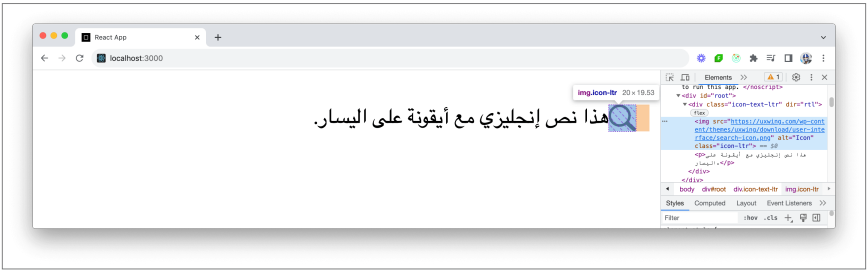


Figure 8-11. Using the `margin-right` CSS property in an RTL setting

To achieve a similar user experience in both the RTL and LTR language settings, we'll want the icon margin to always be applied between the text and the icon. We could simply create separate classes that can be applied for different language settings, but a more effective approach here would be to leverage CSS logical properties.

CSS logical properties

CSS logical properties provide a way to control layout through logical, rather than physical, direction and dimension mappings. Traditional CSS is based on the physical dimensions and directions (top, right, bottom, left). However, with logical properties, we describe styles in terms of their *relation* to the flow and direction of the content (start, end, inline, block).

The main advantage of using logical properties is that they automatically adapt to the writing mode and directionality of the element they're applied to. This makes them extremely powerful for building multi-directional layouts without requiring separate classes or styles for different text directions.

Revisiting the icon example we discussed above, instead of using the `margin-right` CSS property, we would use margin-inline-end. In an LTR context, `margin-inline-end` is equivalent to `margin-right`, and in an RTL context, it translates to `margin-left`.

Icon positioned with the margin-inline-end CSS property

```
.icon-text-ltr {
  display: flex;
  align-items: center;
}

.icon-ltr {
  width: 20px;

  /*
    Margin applied after the icon
    in the inline direction
  */
  margin-inline-end: 10px;
}
```

Now, regardless of whether the text direction is LTR or RTL, the `margin-inline-end` ensures that the margin is applied between the icon and the text, which helps make the layout remain consistent across different languages and text directions.

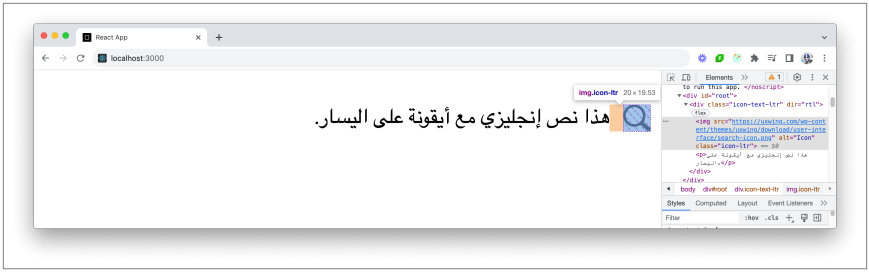


Figure 8-12. Using the `margin-inline-end` *CSS property in a RTL setting*

Wrap Up

In conclusion, internationalization (i18n) is a crucial process in web development that allows applications to cater to users from different regions and locales. By separating text from code, utilizing localization libraries like react-intl, formatting dates, times, and numbers appropriately, and considering right-to-left (RTL) languages, we can create applications that are more accessible, functional, and culturally appropriate for a global audience.

Organizing Code

As a web application grows, so does the complexity of its architecture, the potential for bugs, and the difficulty in introducing new features or making changes. A well-organized codebase is akin to a well-organized library of books; everything has its place, making it easier for developers to find what they need quickly and efficiently. This helps in the maintainability, scalability, and collaboration of a codebase as it grows over time.



In this short chapter, we'll go over some practices we've found that can be helpful when it comes to organizing a React/JavaScript codebase. Note that these are just guidelines, not strict rules. What's important is to have *some* agreed-on best practices within your team for

code organization versus having to follow someone else's conventions rigidly.

"Some of the biggest challenges I had to face when maintaining a large codebase have to do with code organization. There are questions that always come up when adding new code to a project–'where should we put our shared components?', 'Should we put these two modules in the same folder?', 'Should we split our data fetching and rendering logic into different files?', [etc.].

These are always hard questions, but they're even harder when the codebase doesn't have a solid structure with strong opinions about how it should be organized.

I've learned that organizing code using established patterns such as MVC [Model-View-Controller] and strategies like Domain-Driven Design can have a massive impact on the maintainability and scalability of a codebase. These patterns might seem too rigid for smaller projects, even over-engineered–but the larger a project grows, and the more people contribute to it, the more you'll reap their benefits. A solid structure allows developers to add new code without thinking too much about the best place to put it. More importantly, it enables developers to add new features while keeping a consistent and organized code structure throughout the project's life cycle." ***[Maxi Ferreria, Staff Software Engineer @ HelpScout]***

Folder and file structure

A well-organized folder and file structure is essential for managing a large-scale React application. Below is a guide on some ways we can organize an application's folder and file structure:

Root-level folders

In the context of a project directory, root-level folders are the main directories we see when we first open the project. They're usually found at the highest level in our project's hierarchy and serve as categorical dividers for our project's contents.

One simple but effective way of structuring the root-level folders is by segmenting them as something similar to the following:

A simple structure for root-level directories

```
src/
public/
tests/
build/
docs/
```

The following folders would have responsibilities as follows:

src/

This is where the "meat" of our application resides. It's the main directory where we spend most of our time and contains all our source code.

Contents: Components, assets, utilities, configuration files, etc.

public/

Contains all the static files that our app might need but doesn't necessarily go through the build process (i.e., the process of minification, transpilation, and bundling, which transform development code into optimized, production-ready files).

Contents: The main `index.html` file, images, and other non-source code files that are served directly by the web server.

tests/

For projects that adhere to testing (and most should), this directory is dedicated to test configurations and often global or integration test suites.

Contents: Test configuration files.

build/ (generated)

Typically generated by our build tool (like Vite or Webpack) and contains the compiled code ready for deployment.

Contents: Minified JavaScript, optimized images, compiled CSS.

docs/

Purpose: A handy place to store any project-level documentation. This is not just for external users but also for other developers. This can include architectural decisions, component usage guides, and more.

Contents: `README.md` for each significant part of the app, architecture diagrams, API usage guides, etc.

The `src/` directory

The `src/` directory, often the heart of any web application, is where our source code lives. Its structure is crucial because it determines how easy it is for developers to navigate through the code and understand the application's architecture.

Organizing the `src/` directory in a way that's logical and reflects the application's domain and concerns can impact the speed and efficiency of the development process.

How content in the `src/` directory can be arranged

```
src/
  components/
  pages/
  hooks/
  services/
  store/
  utils/
  assets/
  constants/
  types/
```

In the above `src/` directory structure:

- `components/` are all reusable components organized by features or domains.
- `pages/` are top-level components representing individual routes or views in our application.

- `hooks/` contain the custom hooks that encapsulate reusable logic, such as data fetching or state management.
- `services/` represent external services, such as API clients or other integrations.
- `store/` contains the centralized state management setup, including actions, reducers, and middleware (e.g., for Redux).
- `utils/` contains utility functions and common helper modules.
- `assets/` represent static assets like images, icons, fonts, and styles.
- `constants/` contain constants like API endpoints, configuration values, or enums.
- `types/` contain shared type definitions and interfaces For a TypeScript codebase.

Keep in mind not everything shown above needs to be in a codebase. Depending on the complexity, goals, and specific requirements of your project, you may find that you need only a subset of these directories or perhaps even additional ones.

Feature/domain-based organization

Another approach to organizing content within the main `src/` can be based on feature and domain. This involves grouping related components, hooks, utilities, and assets in the same folder, **organized by feature or domain**.

Here's an example of structuring content in the `src/` directory by grouping them within certain features/domains:

Arranging content in the `src/` directory within features

```
src/
  features/
    Authentication/
      components/
        LoginForm/
```

```
      LoginForm.js
      LoginForm.module.css
    SignUpForm/
      SignUpForm.js
      SignUpForm.module.css
  hooks/
    useAuth.js
  services/
    authService.js
UserProfile/
  components/
    ProfileCard/
      ProfileCard.js
      ProfileCard.module.css
  hooks/
    useUserProfile.js
  services/
    userService.js
# ...
```

In the above directory breakdown, we have content arranged within the "Authentication" and "UserProfile" feature domains. Developers working on the "Authentication" feature, for example, can find all necessary components, hooks, and services within its dedicated folder, minimizing the need to search through unrelated parts of the codebase.

Naming conventions

Naming conventions in a codebase ensure consistency, clarity, and predictability, making it easier for other developers to understand the purpose and function of files, directories, and variables at a glance. Here's one helpful way of naming files within certain directories:

- Components: Use UpperCamelCase for component names and their respective file names (e.g., `Header.js`).
- Hooks: Prefix custom hooks with `use` and follow camelCase naming (e.g., `useFetchData.js`).
- Services: Use camelCase and include the service or domain name (e.g., `authService.js`).

- Utilities: Use camelCase and describe the utility's purpose (e.g., `arrayHelpers.js`).

- Styles: Use the `.module.css` or `.module.scss` extension for CSS or SCSS files, respectively, when using CSS Modules (e.g., `Header.module.css`).

Barrel exports

Barrel exports are a design pattern in JavaScript and TypeScript projects that aggregate many exports from a module into a single convenient module. This pattern is useful for simplifying imports in other parts of an application.

A barrel export is essentially an `index.js` or `index.ts` file in a directory that re-exports things from other files, allowing for a more consolidated import. As an example, in the `src/features/Authentication/components/` folder, we can create an `index.js` file responsible for exporting all the components within the directory.

Exporting all components within the `Authentication/components/` directory

```
// .../Authentication/components/index.js
export { LoginForm } from './LoginForm';
export { SignUpForm } from './SignUpForm';
```

With this setup, instead of importing components like the following in other files:

Importing components directly from their files

```
// .../Authentication/Authentication.jsx
import { LoginForm } from './components/LoginForm';
import { SignUpForm } from './components/SignUpForm';
```

We can import them in a grouped manner:

Importing from the "barrel"

```
// .../Authentication/Authentication.jsx
import {
  LoginForm,
  SignUpForm,
} from "./components";
```

Barrel exports allow for cleaner, consolidated imports and can sometimes make moving files easier since we'd only have to update the barrel `index.js` file.

While useful, it's important to be mindful not to make barrel files too large or complex. Similarly, having barrel files for every parent directory can cause a hit on the performance of tooling and build processes within an application. Marvin Hagemeister, a member of the PreactJS team, has written a great article on this topic—Speeding up the JavaScript ecosystem - The barrel file debacle.

Following some of the practices we've mentioned above can help create a clear and maintainable folder and file structure that makes it easy to navigate and understand your large-scale React application. However, while the guidelines provided offer a solid starting point, **it's essential to adapt and refine your structure based on real-world usage and feedback**. Over time, you may discover that certain patterns emerge within your team or specific pain points that need addressing.

Other good practices

Folder and file organization have one of the bigger impacts on how code is organized within a large React/JavaScript application. Outside of this, there are other practices we can take to ensure that our application remains scalable, maintainable, and developer-friendly.

Modularize your code

Break down your components, functions, and services into small, reusable modules. This not only promotes DRY (Don't Repeat Yourself) principles but also makes code easier to test and refactor.

When it comes to React components, create small, focused ones where each component (ideally) should do one or a few things and do them well. In addition,

- **Encapsulate component state and logic**: Keep a component's state and related logic within the component, making it self-contained and easy to reason about.

- **Use functional components with hooks**: Prefer functional components over class components to improve readability and take advantage of React hooks.

We discuss modularization and componentization in more detail in the chapter **Modularity**.

Maintain a clear separation of concerns

Maintaining a clear separation of concerns means ensuring that each module, component, or function in the codebase has a distinct responsibility. Some strategies to help achieve this include:

- **Having a layered architecture**: Organize your code into layers such as presentation, logic, and data. For instance, in a React application, the presentation layer would be your components, the logic might be in your hooks or utility functions, and data could be managed by your state management tool or API integrations.

- **Avoid side effects** (where applicable): Functions or components should not always produce side effects (unintended or observable changes outside their scope). When they do, it should be explicit. Pure functions, which always produce the same output given the same input and produce no side effects, are easier to test and understand.

- **Decouple from external dependencies**: Instead of calling external services or APIs directly within components, use service layers or adapters. This makes it easier to change the external service in the future without affecting the rest of your code.

Separate data fetching logic from presentation logic

Keep data fetching logic separate from the presentation logic. To achieve this, we can create and use custom hooks or services focused on data retrieval and management. Alternatively, we can employ state management tools such as Redux, MobX, or even the React Context API. These tools help to keep our data-fetching logic distinct and organized rather than mixing it with the component logic of our application.

We delve into the topics of data fetching and state management in greater detail in the respective chapters titled **Data Fetching** and **State Management**.

Follow a CSS methodology

Adopting a consistent CSS methodology and approach can help in structuring your stylesheets in a scalable and maintainable way. This not only aids in avoiding conflicts and specificity issues but also makes it easier for developers to understand and contribute to the styling code. Some helpful guidelines for adhering to a certain CSS approach:

- Select a CSS approach that suits your team's preferences, such as BEM, SMACSS, or CSS-in-JS (StyleX).
- Scope your styles to specific components, avoiding global styles that may cause unintended side effects.
- Create reusable CSS classes or styled-components for common design patterns.

Implement unit and integration tests

Though testing code can help catch bugs early and ensure that an application behaves as expected, it can indirectly aid in code organization and enhance the overall design and maintainability of a codebase.

To write testable code, you'll naturally lean towards a modular design where components or functions have single responsibilities. Such a design makes it easier to write unit tests. In addition, overly complex

code becomes evident because it's harder to test. This can lead you to simplify or refactor your code to make it more testable.

Lastly, tests provide a safety net. When you're refactoring your code to improve its organization, having unit tests helps ensure that you haven't introduced new bugs or regressions in the process.

We discuss the topic of testing in more detail in the chapter titled **Testing**.

Enforce a consistent code style

Having a consistent coding style across your project makes it easier for you and other developers to read and understand the code. Tools like ESLint and Prettier can be used to enforce coding standards and style guidelines.

We discuss ESLint and Prettier in a bit more detail in the chapter titled **Tooling**.

Use TypeScript

TypeScript offers static type checking, which can catch errors at compile time, making your code more robust and maintainable. This, by nature, makes TypeScript a self-documenting tool because developers need to define types and interfaces for their data structures and function signatures.

In addition, TypeScript can play a big role in code organization by enforcing clear contracts, providing enhanced VSCode Intellisense support, and supporting easier refactoring efforts with the safety net of static type checking.

We discuss TypeScript in more detail in the chapter titled **TypeScript**.

Document your code

Good documentation is essential for any project. It helps new developers understand the codebase and provides a reference for all team members.

This includes README files, more extensive documentation for APIs or libraries, and even inline comments where applicable.

Use version control

In the context of code organization, version control systems, like Git, play a pivotal role in ensuring structure, traceability, and collaboration.

Version control maintains a chronological record of all changes made to the codebase. This log makes it easy to identify when and by whom a particular piece of code was introduced or modified. This traceability is crucial when trying to understand the evolution of code organization over time.

Branching allows developers to work on different features or bugs in isolation without affecting the main codebase. This ensures that the default branch remains clean and organized, with features being merged in only when they're complete and tested.

Lastly, commit messages and associated documentation (like README files) offer insights into why certain organizational decisions were made. Good commit messages can serve as a roadmap for the project's evolution and structural changes.

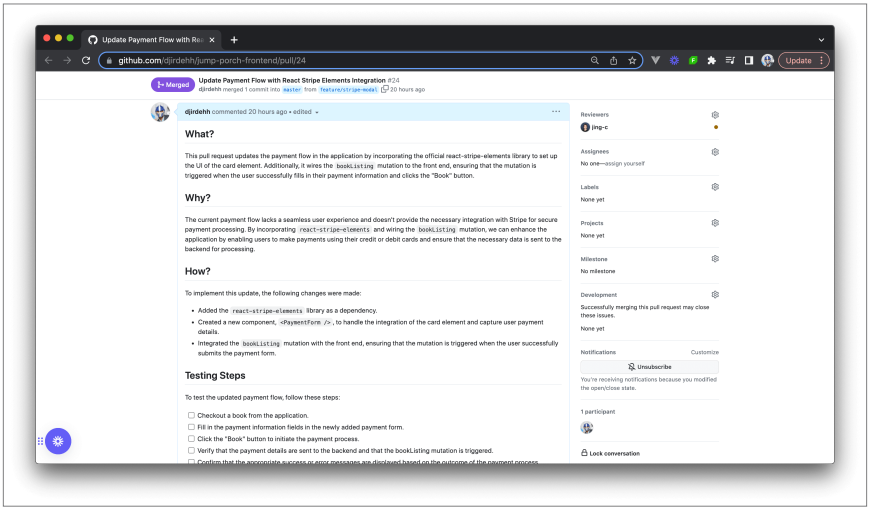


Figure 9-1. A detailed README for a pull request

Regularly refactor

Regular refactoring can greatly improve the structure and readability of your code. It helps maintain the codebase, making it easier to add new features and fix bugs as a codebase grows large. Regularly refactoring can involve:

- **Identifying and fixing code smells**. Look for areas of your codebase that exhibit code smells, such as overly complex components, duplicated code, or tightly coupled components.
- **Making incremental improvements**. Make small, incremental improvements to your codebase as you work rather than waiting for a major refactor.
- **Use code reviews**. Use code reviews as an opportunity to identify areas for improvement and refactoring.

Wrap up

Organizing code effectively in a web application, especially as it scales, is essential for long-term maintainability and efficient collaboration. An organized codebase is like a well-maintained library where each book (or piece of code) is placed in its designated spot, ensuring swift access and easy understanding.

While guidelines and some of the best practices we've discussed in this chapter are beneficial, they are not set in stone. The right organization strategy for your project depends on its unique requirements, your team's preferences, and the challenges you encounter along the way.

Communication within your team is key. Regularly discuss and re-evaluate your code organization approach, especially as the application grows and evolves. Taking the time to regularly refactor, using version control judiciously, documenting your decisions, and ensuring everyone is on the same page will pay dividends in the long run.

Personalization and A/B Testing

Modern web development isn't only about creating functional applications—it's about making these applications relevant and engaging to users. How do we know certain design decisions we make resonate with our audience? How can we tailor the user experience of an application for maximum engagement? This is where topics like **personalization** and **A/B testing** come in.



Personalization involves tailoring the user experience based on a user's preferences, behavior, and other data. A/B testing, sometimes commonly known as split testing or controlled experiments, involves testing two or more versions of a feature or interface to determine which one is more effective.

When A/B testing is combined with personalization, we can offer tailored experiences to users while measuring their effectiveness. As a simple example, for a shopping application, we can A/B test two different recommendation algorithms to see which one leads to better sales for a particular user segment (e.g., users based in North America).

Personalization, A/B testing, and managing controlled experiments are deep topics that require expertise in data analysis, engineering, and statistics. In this chapter, we'll discuss a summary of some of these concepts in a summarized fashion while also discussing some implementation steps that can be taken to facilitate personalization and A/B testing in a React application.

Personalization

Personalization refers to the practice of creating tailored experiences for individual users or groups based on their preferences, behavior, or other identifiable attributes.

The first step in implementing personalization is to identify the areas of the application where personalization can be useful. This could include personalizing the content shown to users based on their preferences, their device or location, or even based on their behavior.

By identifying personalization opportunities, we can determine which personalization techniques to use and how to implement them.

User data

Personalization often requires user data, such as user preferences, behavior, and location. To implement personalization in a React application, we can use techniques like user profiling, tracking user behavior, and collecting user feedback.

This user data can then be stored in a database where our React app can query for this information and use it to personalize the content and experience of an application based on the user viewing the app.

The Context API

The Context API in React provides a way to share values between components without having to explicitly pass a prop through every level of the tree. This can be helpful when wanting to make user data accessible across an application, allowing for easy personalization in various components based on a user's information.

For example, assume we want to tailor the color scheme of a certain component based on information we have about the user. We can create a context object in the root of our application to contain user information that we obtain from an external service or API.

Setting up the user context and provider

```
import React from "react";

// Create UserPreferences context
const UserContext = React.createContext({});

export const UserProvider = ({ children }) => {
  /*
   get user preferences information from
   external service/API
  */
  const userPreferences = getUserPreferences();

  return (
    <UserContext.Provider
      value={{ userPreferences }}
    >
      {children}
    </UserContext.Provider>
  );
};
```

In our main application component, we can wrap the components that need access to the user data with the `UserProvider` we've created above.

Wrapping child components with `UserProvider`

```
import { UserProvider } from './UserContext';

function App() {
  return (
    <UserProvider>
     {/* other components */}
    </UserProvider>
  );
}

export default App;
```

Now, any component within our application can access this user preferences data and use it for personalization. As a simple example, assume we wanted to display a certain background color for a component based on the user preferences information we have.

Personalizing an element's background color

```
function ChildComponent() {
  const { userPreferences } =
    useContext(UserContext);

  // Extracting color preference from user data
  const { backgroundColorPreference } =
    userPreferences;

  return (
    <div
      style={{
        backgroundColor:
          backgroundColorPreference,
      }}
    >
      Personalized content based on user
      preference goes here.
    </div>
  );
}
```

Though the example we've shown above is simple, leveraging the Context API to help personalize an application can be extended to handle

many different scenarios. We can dynamically render different components based on user data, render translations or localized content based on user location or locale preference, and perhaps even adjust UI elements or interactivity based on user accessibility requirements.

A/B Testing

A/B testing, also known as split testing or controlled experiments, is the method of comparing two or more versions of a web page, feature, or product against each other to determine which one performs better. Better performance can mean different things depending on the specific goals of the test. This can be increasing click-through rates, boosting sales, enhancing user engagement, reducing bounce rates, or achieving any other key performance indicator (KPI) you or your team deem crucial.

When attempting to implement an A/B test, many different key steps need to be kept in mind:

- **Segmenting users**: dividing an application's users into distinct groups, ensuring that each user will only see one version of the tested feature.

- **Serving different versions**: creating different variations of the feature or component in question. With conditional rendering, we can then serve the appropriate variation to each user segment.

- **Collecting data**: track user interactions, conversions, and other relevant metrics for each variation and store this information for analysis.

- **Analyze experiment results**: After gathering sufficient data or when the experiment is concluded, we can then compare the performance metrics of each variation to determine which version yields the best results.

Many of the above points require work beyond working within just a client/React application and can require a large effort to build and maintain. As a result, in many real-world scenarios, organizations utilize third-party tools like Optimizely, Statsig, and Launchdarkly to make this A/B testing process easier. These tools offer a pre-built

infrastructure, user-friendly dashboards that allow for easy test creation/monitoring/analysis, and analytics tools to analyze experiment results.

Let's go through an example of how an A/B test can be implemented in a React app with the help of the Statsig tool. Though we'll use the Statsig tool in our example, the concepts and methodologies we discuss are applicable to any A/B testing setup. With that said, assume we wanted to create an experiment with the following details.

Button Color Test:

- **Hypothesis**: Changing the color of a certain button to green will significantly increase the click-through rate when compared to the existing button color blue.
- **Objective**: Determine which button color leads to higher click-through rates.
- **Setup**: Create two variations of a button, one blue and one green. Serve one variation to 50% of all users and the other variation to the remaining 50% of users.
- **Data collection**: Track button clicks for both variations using Statsig's built-in event tracking.
- **Analysis**: Compare the click-through rate of both variations to determine which color is more effective.

In the Statsig dashboard, we can create a new experiment for the above and have two experiment groups:

- **Control:** 50% of users who'll be presented with the blue button.
- **Test**: 50% of users who'll instead be presented with the green button.

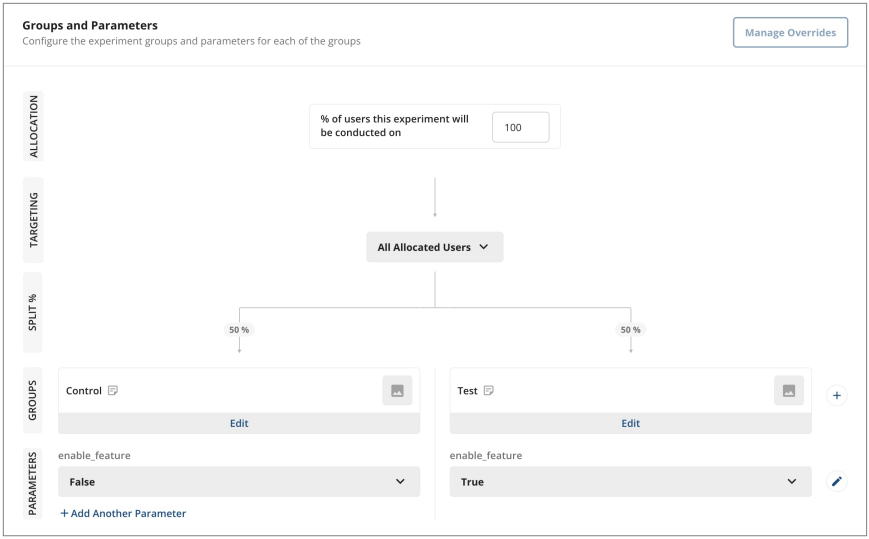


Figure 10-1: Setting up the `button_color` experiment

For each of the experiment groups, we've created an `enable_feature` parameter to dictate whether the new feature should be enabled. In the control group, `enable_feature` is set to a boolean value of `false`, while in the test group, `enable_feature` is set to `true`.

The next step of our experiment configuration is to specify a `clickthrough_rate` metric we want to track in this experiment. This metric will be calculated by taking the number of clicks on the button divided by the number of times the button is displayed to users.



Figure 10-2: Specifying `clickthrough_rate` as the primary metric to be tracked

Tools like Statsig allow us to either use certain metrics the tool provides out-of-the-box or create custom metrics that we want to measure in our experiments.

With this setup in place, we can work towards having our React application integrate with the experiment we've created in Statsig. Statsig provides a statsig-react SDK to help with this. At the root of our component tree, we can wrap our app in a StatsigProvider and initialize

the SDK with an active Client API Key we receive from our Statsig dashboard.

Initializing Stasig in our React app

```
import { StatsigProvider } from "statsig-react";

function App() {
  return (
    <StatsigProvider
      sdkKey="<STATSIG_CLIENT_SDK_KEY>"
      waitForInitialization={true}
    >
      <div className="App">
        {/* Rest of App ... */}
      </div>
    </StatsigProvider>
  );
}

export default App
```

In the relevant child component, we can then toggle the color of the button we want to test based on what experiment group the user falls in. To access the experiment configuration information, we can use the useExperiment hook available to us from the Statsig React SDK.

From the experiment configuration object, we'll access the value of the `enable_feature` parameter that has been assigned to the control and test groups.

Getting experiment configuration details

```
import { useExperiment } from "statsig-react";

function ButtonComponent() {
  // access experiment configuration
  const { config } = useExperiment(
    "button_color",
  );

  // access value of experiment parameter
  const showGreenButton = config.get(
    "enable_feature",
```

```
  );

  return (
    // ...
  )
}

export default ButtonComponent
```

The `enable_feature` parameter is given a value of `true` in the test group, the group in which we want to test how users behave when the color of the button is green. With this, we can use a simple conditional statement to decide which button color to render depending on which experiment group the user falls in:

Render different button colors based on experiment group

```
import { useExperiment } from "statsig-react";

function ButtonComponent() {
  // access experiment configuration
  const { config } = useExperiment(
    "button_color",
  );

  // access value of experiment parameter
  const showGreenButton = config.get(
    "enable_feature",
  );

  /*
    determine button color based on
    experiment group
  */
  const buttonColor = showGreenButton
    ? "green"
    : "blue";

  return (
    <button
      style={{ backgroundColor: buttonColor }}
    >
      Click Me
    </button>
```

```
  );
}

export default ButtonComponent;
```

At this moment in time, the `ButtonComponent` we've created above will render a button with either a blue or green color based on the user's assignment to the control or test group, respectively. The only thing remaining is to track how users are responding to different variants of the experiment.

The primary metric of our experiment is `clickthrough_rate`, which is calculated by taking the number of clicks on the button divided by the number of times the button is displayed to the users. The number of times the button is displayed to users will be tracked and handled by Statsig's event tracking system. However, we'll still need to make sure that every interaction with the button is captured, and we can do this with Statsig's logEvent() API.

When the button element is clicked in our `ButtonComponent` example, we'll log an event labeled `button_clicked`.

Log button clicks

```
import {
  Statsig,
  useExperiment,
} from "statsig-react";

function ButtonComponent() {
  const { config } = useExperiment(
    "button_color",
  );

  const showGreenButton = config.get(
    "enable_feature",
  );

  const buttonColor = showGreenButton
    ? "green"
    : "blue";

  const onButtonClick = () => {
```

```
    // log button click
    Statsig.logEvent(
      "button_click",
      buttonColor,
    );
  };

  return (
    <button
      style={{ backgroundColor: buttonColor }}
      onClick={onButtonClick}
    >
      Click Me
    </button>
  );
}

export default ButtonComponent;
```

When the button is now clicked by a user, the `onButtonClick()` function triggers the `Statsig.logEvent()` call, which will send the click event to the Statsig servers for tracking.

Once the experiment yields statistically significant results, and assuming our custom `clickthrough_rate` metric is configured correctly, we'll be able to analyze the experiment results right in the Statsig dashboard.

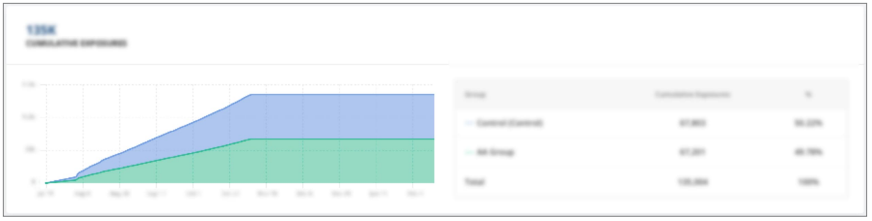


Figure 10-3: Analyzing experiment results

If the test group (i.e., group where the button color is green) has a higher click-through rate compared to the control group (i.e., group where the button color is blue), this would indicate that the green button variant performed better in encouraging users to click. With this information, we can then decide on the next steps, which might include a full-scale

rollout of the green button or further testing with different shades or styles.

Analyzing experiment results consists of many important topics such as determining statistical significance, understanding the impact on key performance indicators, evaluating secondary metrics, considering external factors, and interpreting user behavior patterns. By using third-party tools, teams and organizations can streamline the experimentation process and visualize results more effectively, which can help make more informed decisions from experiment outcomes.

Feature Flags

In a large-scale application, when we're ready to launch a new feature to users, there can be many instances where we may not want to launch the feature to 100% of all users right out of the box. Some of these reasons include:

- **Mitigating risk.** Rolling out a feature gradually helps in identifying any potential bugs or issues with a limited subset of users before they affect the entire user base. This phased approach minimizes the risk associated with introducing new features.

- **Stress testing infrastructure.** Some new features might have a significant impact on back-end infrastructure, especially if they involve data-intensive operations. Rolling out to a subset of users allows a team to monitor server loads and optimize accordingly before rolling out to all users.

- **Collecting user feedback**. Releasing a new feature to a smaller audience first can help gather feedback before the feature is ready for a full rollout. This user feedback can be crucial to understanding if the feature meets user needs or if there are any improvements required.

- **Testing the market**. If the new feature is a significant deviation from what users are accustomed to, it might be valuable to test its acceptance in the market. Rolling out the feature regionally or to specific demographics can provide insights into how different user segments perceive the change.

- **Ensuring a rollback plan**. If anything goes wrong during a phased rollout, having a smaller subset of users affected makes it easier to roll back the feature or implement fixes without causing widespread disruption.

Feature flags (also commonly known as feature gates or beta flags) offer a powerful way to manage the release and control of new features within an application. They act as a switch that can turn on or off specific functionalities, enabling teams to have more granular control over what users experience in real time.

"In every organization I've been a part of, I've always rolled out big UI changes carefully with the help of feature flags. I would start small, like just 10% of users, then bump it up–25%, 50%, and finally all the way to 100%. The feature flagging system was often something custom built and maintained by the company, which gave me and the team great flexibility and customization options.

*While the staged rollout is underway, I would keep a close eye on latency and error tracking dashboards. If any issues arise, I'll rollback the feature to 0%, investigate the cause, and address the problem before attempting another rollout. This approach allows me and the team to proceed with caution and precision as we launch, ensuring a reliable experience for users." **[Hassan Djirdeh, Software Engineer @ Doordash]***

While it's possible to build a custom feature flagging system, many companies and organizations prefer using third-party tools like Statsig, Growthbook, and Optimizely to manage their flags. In addition to allowing teams to run A/B testing and experiments seamlessly, these tools also provide extensive support for creating and managing feature flags.

Continuing with the example of using Statsig as our feature flag and experimentation tool, we can use the Statsig dashboard to create a feature flag for an application. Assume we wanted to create a feature flag to control the rollout of a new button color across our app.

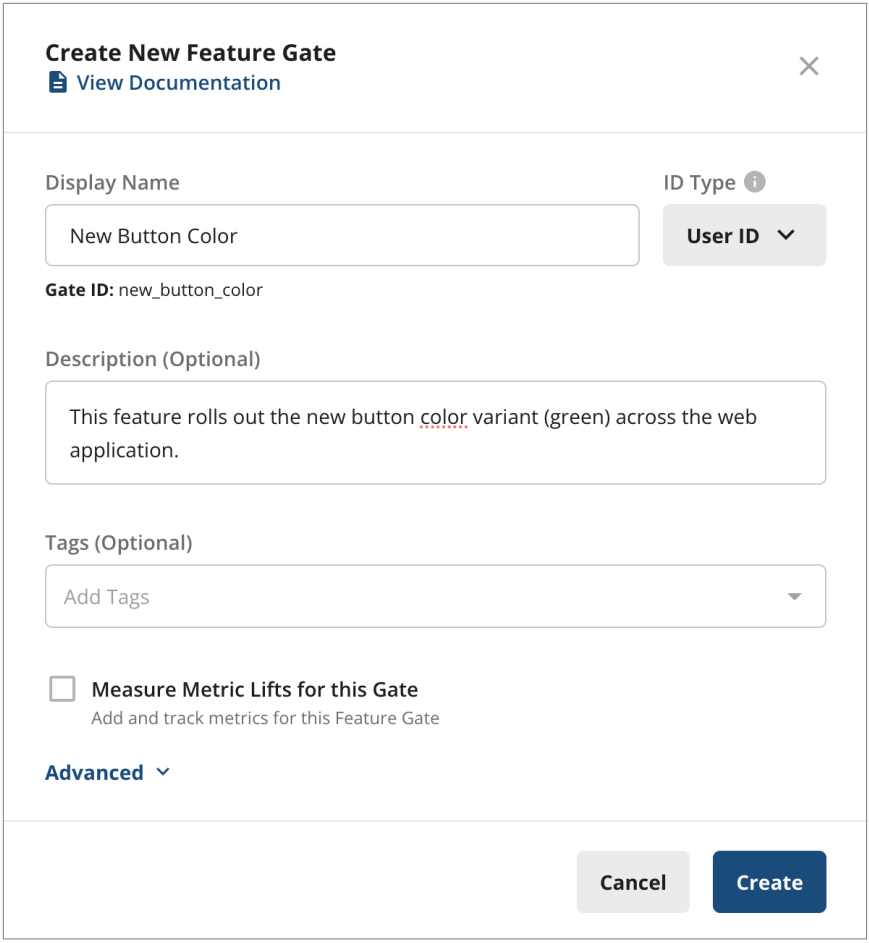


Figure 10-4: Creating a new feature flag

With the feature flag created, we can then define a rule for how we'll want our feature flag to be rolled out. For example, we could rollout for certain browsers, operating systems, or a certain subset of users that use our app (e.g., all employees).

For the new button color feature, assume we wanted to set a rule that only 20% of all our web users will be presented with this new feature.

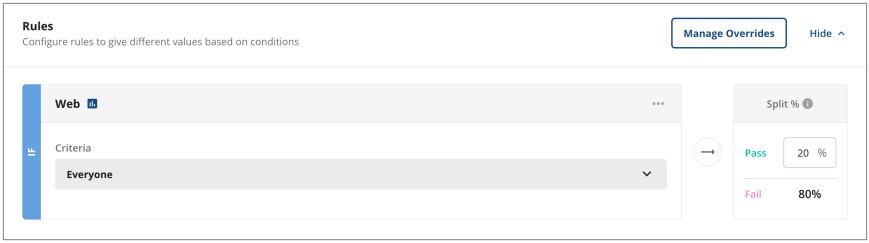


Figure 10-5: Setting a rollout to 20% of all web users

With the feature flag now configured in our Statsig dashboard, we can move to our React app and attempt to implement the new feature behind the feature flag.

In our React app, integrating Statsig's feature flags is straightforward and shares similarities with the A/B testing setup we went through above. We can utilize the useGate Hook provided by the Statsig React SDK to evaluate whether the feature should be enabled for a specific user.

Display a green button color when feature flag is enabled

```
import { useGate } from "statsig-react";

function ButtonComponent() {
  // Evaluate the feature flag
  const { value: isFeatureEnabled } = useGate(
    "new_button_color",
  );

  // Decide which button color to use
  const buttonColor = isFeatureEnabled
    ? "green"
    : "blue";

  return (
    <button
      style={{ backgroundColor: buttonColor }}
    >
      Click Me
    </button>
  );
}

export default ButtonComponent;
```

With the above code in place, users who fall into the group where the new feature is enabled will see the green button when they load the page, while the rest of the users will see the default blue button. This dynamic rendering allows for real-time changes to user experiences based on the conditions set in the feature flag.

In practice, if no issues arise and/or feedback on the green button is positive, the team can gradually increase the rollout percentage from 20% to larger segments of users **without deploying any new code**. Conversely, if issues arise or feedback is negative, the feature can be turned off instantly, reverting all users to the blue button experience.

Feature flags not only provide technical benefits but also offer business advantages. They enable product managers, marketing teams, and customer support to coordinate feature releases, run targeted promotions, and manage customer feedback more effectively.

Wrap up

In this chapter, we've come to understand how to create tailored user experiences through personalization, experimentation via A/B testing, and conducting strategic rollouts of features using feature flags. When employed thoughtfully, these capabilities help enable teams and organizations to create user-centric products that evolve and adapt in response to real-world feedback and changing user needs.

Scalable Web Architecture

Throughout the book, we've discussed topics to help address the building and management of large-scale JavaScript/React applications. Among other things, this included discussing the use of a design system to ensure UI consistency and accelerate the development process, leveraging sophisticated data-fetching libraries to optimize data retrieval, and thinking about ways to manage and handle client-side state effectively.



All these points focus on the "large" part of building and managing large-scale apps but don't necessarily cover the concept of **scalability**—a system's capacity to handle varying workloads by adding or removing resources as needed. In this brief chapter, we'll spend time discussing the

concept of scalability and explore some strategies for building scalable architectures and infrastructure.

Scalability

Scalability, in a nutshell, can be summed up as **the capability of a web architecture to handle growth gracefully**. This means that as the number of users, transactions, requests, or data increases, the system can maintain or improve its performance, provide uninterrupted service, and continue to be cost-effective. Scalability is about building an infrastructure that can expand in a linear fashion, where adding additional resources improves performance proportionately to the resources added.

A scalable web architecture comprises various components that work together to ensure an application can handle increased workloads. In the following section, we'll discuss some of these components briefly.

Load balancers

Load balancers are essential components in scalable web architectures. They distribute incoming traffic across multiple servers to balance the load and prevent individual servers from being overwhelmed. This distribution helps maintain optimal performance, even during periods of high traffic.

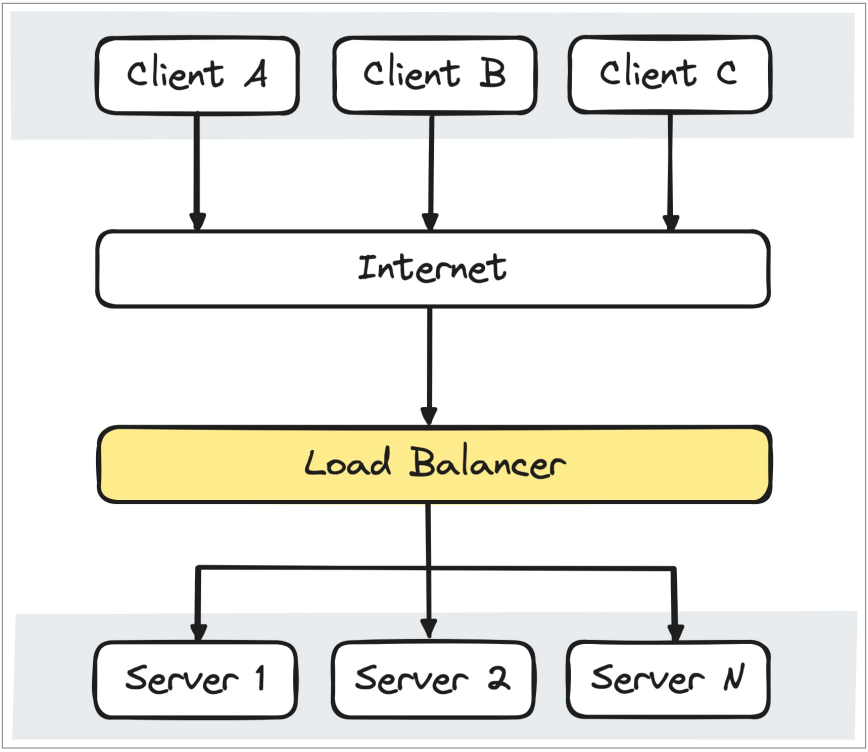


Figure 11-1. Diagram illustrating a load-balancing system

The load balancer is placed between the client and the server cluster that it manages and acts as the entry point for all incoming traffic to a website or application. Configuring a load balancer can be complex and involves several key considerations to ensure that traffic is distributed effectively:

- **Algorithm selection**: A load balancing algorithm (round robin, least connections, IP hash, etc.) needs to be implemented that dictates how the load balancer distributes traffic among the servers.

- **Health checks**: Regular health checks need to be conducted to ensure that servers are operational. If a server fails a health check, it is temporarily removed from the pool until it is deemed healthy again.

- **Auto-scaling**: A load balancer should ideally work seamlessly with auto-scaling systems to add or remove resources based on

current demand, ensuring that the architecture can scale in and out efficiently.

When using cloud services like AWS, Google Cloud, or Azure for deployment, the concepts and practices of implementing a load balancer and its associated features are both simplified and enhanced by the tools and services provided by these platforms. Using these cloud services, we get out-of-the-box features to support managed health checks, integrated scaling, security, automated deployments, and more.

Caching

Caching is the process of temporarily storing frequently accessed data in readily accessible storage locations, known as caches, to speed up data retrieval. These caches act as intermediate stores and enable users or applications to access data more quickly compared to its original storage location, often a database or a back-end server. Caching can be implemented in a few different areas:

- **Browser caching**: Store website resources on the user's local computer when a site is visited, reducing loading times on subsequent visits.

- **CDN (Content Delivery Network) caching**: Utilize geographically distributed servers to provide fast delivery of content (webpages, images, videos, etc.) by caching content in multiple locations around the world.

- **Application/data caching**: Store commonly queried data in in-memory caches like Redis or Memcached. This allows for rapid access to certain frequently accessed application data, taking a load off databases and back-end systems.

Caching can be a vital part of a scalable architecture as it can significantly reduce the load on back-end systems and databases by serving certain cached content, which is much quicker to access. This results in faster response times for the end user and less processing power required to handle each request, which is important during traffic spikes.

When it comes to how and when content can be cached, several strategies can be used depending on the goal. For example, static content such as images, CSS, and JavaScript files that don't change often can perhaps be cached for longer periods, while dynamic content might require a more complex strategy based on how often the content changes and the tolerable delay before the cache is updated.

In the context of cloud services, many providers offer managed caching services (such as Amazon ElastiCache, Azure Cache, and Google Cloud Memorystore) that take care of the distribution, scaling, and management of caches set up with Redis or Memcached.

Content Delivery Networks (CDNs)

Content Delivery Networks (CDNs) are a network of servers strategically placed in different locations around the world, designed to serve static content (like HTML pages, images, JavaScript, and CSS files) **from the server closest to the user**.

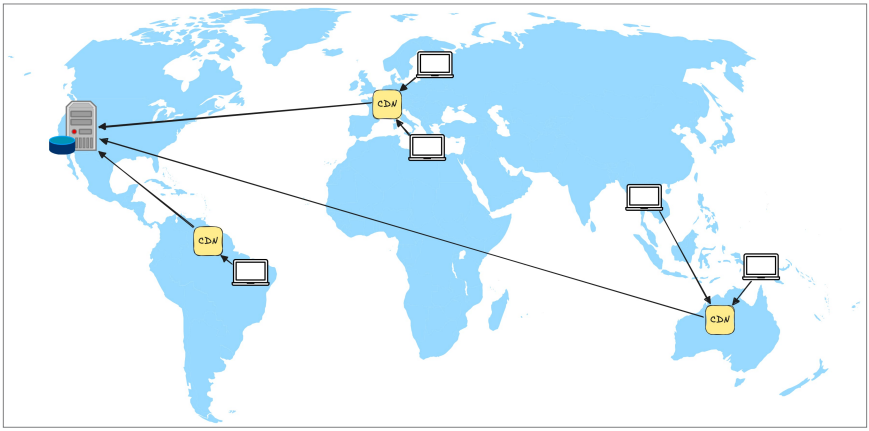


Figure 11-2. Map illustrating how CDNs distribute content closer to users

CDNs play a crucial role in a scalable web architecture by bringing content geographically closer to users, thus reducing latency and speeding up the delivery of content.

Cloud providers often include integrated CDN services that make it easy for businesses to deploy a content delivery network as part of their web

architecture. For example, Amazon Web Services (AWS) has Amazon CloudFront, Microsoft Azure provides Azure CDN, and Google Cloud offers Cloud CDN. These services work by caching content at "edge locations" close to where users are located.

Horizontal scaling

Predicting the exact number of users and determining the infrastructure needed for scaling can be challenging. To handle this, organizations often employ a combination of **horizontal** and **vertical scaling** strategies.

Horizontal scaling, also known as scaling out, involves adding more units to an application's cloud architecture, such as additional servers in a cluster or new instances in a cloud environment. This method is often favored for web-based architectures because it allows for flexibility and resilience. As demand increases, new servers can be added to the pool to distribute the workload further. In the event of a server failure, others can take over, providing high availability.

Horizontal scaling is often suitable when:

- We want to avoid downtime due to a single point of failure.
- Our application requires frequent upgrades.
- We want to avoid vendor lock-in and explore multiple services.
- We want to improve system resilience by using multiple services.

Vertical scaling

Vertical scaling, also known as scaling up, involves maximizing the resources of a single unit to handle increasing load. This can include adding processing power and memory to the physical machine running the server or optimizing algorithms and application code in software terms.

Vertical scaling is typically easier and faster than horizontal scaling as it doesn't require the complexity of setting up new servers. However, it has limits; a server can only be upgraded to a certain point, and such upgrades can involve downtime.

Vertical scaling is often suitable when:

- We need a simple architecture with reduced operational costs.
- We want a system that can scale with low power consumption.
- We need an easily installed and scalable system with lower licensing costs.
- We want to maintain application compatibility.

Modern cloud services offer solutions that simplify vertical and horizontal scaling strategies. These services provide auto-scaling, which can automatically adjust the number of computational resources according to the server load, making horizontal scaling more dynamic. Similarly, they offer instances that can be resized, facilitating vertical scaling.

Microservices

Microservices is an architectural approach that decomposes an application into smaller, independently deployable services that can be scaled and managed separately. Each service runs its own process and communicates with lightweight mechanisms between one another, often through an HTTP resource API or through RPC calls.

Microservices architecture is in contrast to monolithic architecture, where all components of the application are tightly integrated and must be scaled together, which can be cumbersome and inefficient. Microservices enable scalability by allowing each service to be scaled independently. If one part of the application experiences heavy demand, only the related services need to be scaled up, which is more cost-effective and less complex than scaling the entire application.

With the advantages microservices bring, they also introduce complexity in terms of service integration and management, and they can require a robust infrastructure for service discovery, load balancing, and failure recovery. Tools like Kubernetes, Docker, and cloud-based services are often used to manage this complexity by providing orchestration, containerization, and service discovery capabilities.

Characteristics of a scalable application

In the above section, we've come to understand different components that help facilitate a scalable web architecture. A truly scalable system

integrates these components (and others!) in a cohesive and efficient manner to ensure that it can respond to increasing demand without performance degradation or excessive costs.

In a nutshell, a scalable application should excel in the following areas:

1. **Performance**: The app must operate well under stress with low latency, as the speed of a website affects usage, user satisfaction, search engine rankings, and ultimately, revenue and retention.

2. **Availability and reliability**: Scalable apps should rarely go down under stress and must reliably produce and store data upon request.

3. **Manageability**: The ease of diagnosing and understanding problems, making updates or modifications, and operating the system without failure or exceptions is essential for a scalable cloud architecture.

4. **Cost**: Scalable applications should not be prohibitively expensive to build, maintain, or scale. Planning for scalability during development allows the app to expand as demand increases without causing undue expenses.

A typical scalable web architecture consists of four primary layers: web servers, database servers, load balancers, and shared file servers. Each layer can be scaled independently, with the database layer being the most challenging to scale. One approach to efficient database scaling is using master-slave replication, where master nodes can read and write data, and slave nodes can only read data. Load balancers distribute load across master nodes to ensure optimal performance.

Some other best practices include database optimization and asynchronous processing. Optimizing database performance through indexing, query optimization, and employing database sharding or partitioning can help distribute data across multiple servers, improving performance and scalability. The use of asynchronous processing and message queues to handle time-consuming tasks in the background can help prevent bottlenecks and improve application performance.

Where do Kubernetes and Docker fit in?

We briefly mentioned Kubernetes and Docker earlier in this chapter, but it's worth diving into these platforms with a little more detail. This is because these platforms play crucial roles in creating and managing scalable web architectures, primarily by facilitating **containerization** and **orchestration**.

Docker

Docker is an open-source platform that simplifies the process of creating, deploying, and running applications in **containers**.

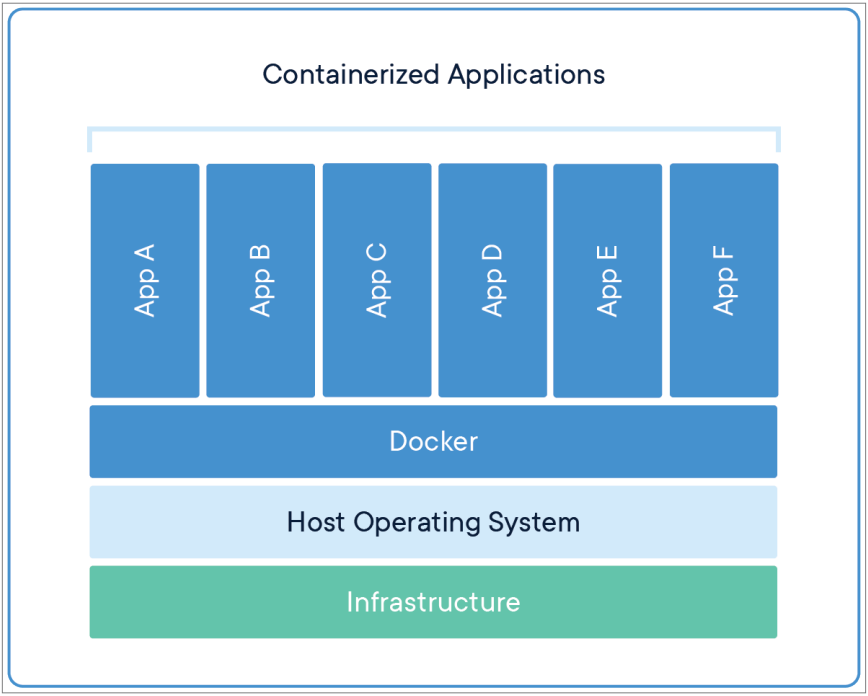


Figure 11-3. Diagram of containerized application architecture using Docker. From the Docker site resources documentation.

Containers package applications and their dependencies together, ensuring that they run consistently in any environment. This approach improves the portability, efficiency, and manageability of applications, making it easier to build and scale complex web architectures.

The advantages that Docker provides include:

1. **Portability**: Docker containers can run consistently across different environments, reducing the "it works on my machine" problem.

2. **Isolation**: Containers encapsulate applications and their dependencies, minimizing conflicts and improving security.

3. **Resource efficiency**: Containers share the host OS kernel and consume fewer resources than virtual machines, allowing for better resource utilization.

4. **Version control and component reuse**: Docker images can be versioned and shared easily, enabling code reuse and simplified application management.

5. **Ecosystem and community**: Docker has a vast ecosystem with numerous tools and integrations, as well as a large, active community.

Docker also comes with a few challenges:

1. **Learning curve**: Docker requires learning new concepts and commands, which may be challenging for some developers.

2. **Limited Windows support**: Although Docker supports Windows containers, the feature set is not as comprehensive as that for Linux containers.

3. **Security concerns**: Running containers with root privileges or using outdated images can expose applications to security risks.

Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. Kubernetes is designed to work with Docker and other containerization technologies.

Kubernetes can enhance a scalable web architecture by facilitating the following:

1. **Scalability**: Kubernetes simplifies the management and scaling of containerized applications, making it easier to handle increased demand.

2. **High availability**: Kubernetes can automatically detect and replace failed containers, ensuring high uptime and resilience.

3. **Load balancing and service discovery**: Kubernetes provides built-in load balancing and service discovery for containerized applications.

4. **Rolling updates and rollbacks**: Kubernetes allows for seamless application updates and rollbacks with minimal downtime.

5. **Extensibility**: Kubernetes can be extended with custom resources and third-party plugins to suit specific needs.

Kubernetes also comes with a few potential drawbacks:

1. **Complexity**: Kubernetes has a steep learning curve and can be complex to set up and configure, particularly for those new to the concept of containerization.

2. **Resource overhead**: Kubernetes clusters require additional resources for control plane components, which may lead to increased infrastructure costs.

3. **Limited support for stateful applications**: Although Kubernetes has improved support for stateful applications with StatefulSets, managing stateful applications can still be challenging compared to stateless ones.

In summary, Docker and Kubernetes offer numerous benefits for building scalable web architectures, but they also come with some complexities and potential drawbacks. Developers and teams should carefully consider their specific needs and requirements when deciding whether to adopt these technologies.

Where do companies like Vercel and Netlify fit in?

Vercel and Netlify are platforms built on top of AWS (Amazon Web Services)/GCP (Google Cloud Platform) that specialize in providing hosting and deployment solutions for front-end web applications. They fit into the scalable web architecture picture by **significantly simplifying the deployment and scaling process for developers, particularly for static sites and serverless functions**.

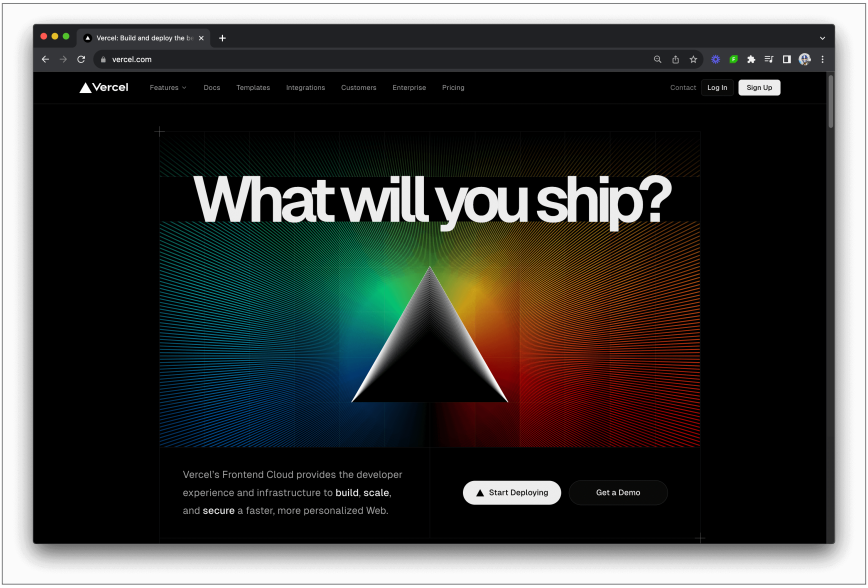


Figure 11-4. Homepage of vercel.com

Tools like Vercel and Netfliy contribute to building and maintaining a scalable web architecture through a variety of means:

1. **Simplified deployment**: Vercel and Netlify offer a streamlined deployment process, allowing developers to easily push their code to Git repositories and have their applications automatically built and deployed.

2. **Serverless functions**: Both platforms support serverless functions, which allows developers to write back-end code without managing the underlying infrastructure. Serverless functions automatically

scale with the number of requests, providing a scalable solution for handling back-end tasks.

3. **Global CDN**: Vercel and Netlify distribute applications across a global content delivery network (CDN), ensuring fast load times and better performance for users worldwide. This also helps to distribute the load, improving the scalability of the hosted applications.

4. **Automatic scaling**: Both platforms automatically scale applications based on demand, ensuring that apps can handle increased traffic without any manual intervention.

5. **Continuous integration and deployment**: Vercel and Netlify offer built-in continuous integration and deployment (CI/CD) pipelines, making it easy for developers to push updates to their applications.

6. **Custom domains and HTTPS**: Both platforms provide custom domain configuration and automatic HTTPS certificate management, simplifying the process of securing applications.

With the above capabilities, tools like Vercel and Netlify enable developers to launch and scale applications more efficiently. The underlying AWS or GCP infrastructure ensures robustness and reliability, while Vercel and Netlify provide a developer-friendly layer that simplifies interaction with these cloud services. This often makes Vercel and Netlify very well-suited for front-end applications, static sites, and serverless functions.

Despite the advantages Vercel and Netlify provide, applications can sometimes be sufficiently complex to require more control over the underlying infrastructure. In these cases, using containerization and orchestration platforms like Docker and Kubernetes, along with cloud providers like AWS, Google Cloud, or Microsoft Azure, may be more appropriate.

Wrap Up

In this chapter, we aimed to emphasize the importance of building a system that can grow and adapt to increased demand. Scalability ensures

that as user bases expand and traffic intensifies, our web applications continue to perform efficiently without disruption.

While we discussed some key components of scalable architectures, it's important to recognize that what we've discussed represents only a portion of an overarching scalable web architecture. Other critical aspects, such as automated testing, continuous integration (CI), continuous deployment (CD), etc., are instrumental in maintaining the health and performance of applications as they scale.

To read more on this topic, we recommend the Google Cloud patterns for scalable and reliable applications guide and Simform's great scalable web applications on AWS write-up.

Testing

In software development, testing is the process of evaluating a system or its components to determine whether they meet the specified requirements. The purpose of testing is to ensure that the application works as expected and meets all functional and non-functional requirements. **In other words, testing is how we make sure that the software does what it's supposed to do**.



When it comes to large-scale React JavaScript applications, testing can be critical since larger applications are complex and can be challenging to debug. Testing can help catch bugs early in the development process, making it easier to fix them before they cause significant issues.

Additionally, when testing is part of the continuous integration pipeline (i.e., the process of automatically building, testing, and deploying code changes), it helps ensure that any code changes are thoroughly tested before they are merged into the default branch or released to production. This reduces the risk of bugs and other issues making their way into production environments, as well as saves time and effort in manual testing and bug fixing.

"In my years at Netflix, I've learned a lot about testing. One of those is it's not just about the code; it's about the culture as well. You have to build testing into the idea of engineering. The understanding is that you're going to make mistakes, and one way to prevent those mistakes is to write tests.

The other part of that, besides the cultural aspect, is the idea that you need to have a cohesive tool set. When I first joined the company, we had unit tests that were all over the place. We had Jest, we had Mocha, and I think we had some Enzyme tests, too.

*Now, we've kind of unified on a standard set of ways of writing tests, which would help people. There was some aspect of wanting to allow people to write tests the way they're comfortable with and familiar with, but it turned out that in the long run, it's better just to have one standard for writing tests and get everybody aligned on doing that. And then make it part of the culture. Every PR check, you say, 'Oh, did you write tests for that? Why aren't there tests for that?'. That's what I found the most success in testing. So it comes down to those two aspects. One is the cultural side, and the other is a cohesive set of tools." **[Jem Young, Engineering Manager @ Netflix]***

To ensure the reliability and functionality of a React application at scale, we can utilize various types of testing, which include **unit testing**, **end-to-end testing**, **integration testing**, and **snapshot testing**. In this chapter, we'll spend some time discussing each of these types of tests, their purpose, and how we can go about creating them in a React application.

Unit tests

Unit tests focus on examining individual components or functions within an application, isolated from the rest of the codebase. These

tests verify that each component or function behaves as expected and meets its functional requirements.

Jest and React Testing Library are the tools often recommended for running and writing unit tests:

- **Jest** is a popular JavaScript testing framework that makes it easy to write and run tests for React applications. It provides a variety of useful features such as mocking, code coverage, and parallel test execution.

- **React Testing Library** is a lightweight library designed specifically for testing React components. It encourages writing tests that focus on the user's experience rather than implementation details.

To illustrate how unit tests can be written for a React application, let's consider a simple example. Assume we have a `Counter` component that increments or decrements a number based on user interaction with buttons.

A `Counter` component

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
    </div>
  );
```

```
};

export default Counter;
```

If we want to test this component in isolation, we can consider writing tests that assert:

- The count value is correctly displayed in the span element and has an initial value of 0.
- The count increments by 1 when the "+" button is clicked.
- The count decrements by 1 when the "-" button is clicked.

With Jest, we're able to create the structure for the above tests within `describe()` and `it()` blocks in a separate `Counter.test.tsx` file.

The `Counter.test.tsx` file

```
import React from 'react';

describe('Counter component', () => {
  it('initial count is 0', () => {
    // ...
  });

  it('increases count on "+" button click', () => {
    // ...
  });

  it('decreases count on "-" button click', () => {
    // ...
  });
});
```

React Testing Library provides a set of utility functions that allow us to interact with our React components as if we were a user. These functions simulate user events like clicking, typing, and scrolling and can be used to assert that the component behaves correctly in response to these events.

For our first test, we can use the queryByText() utility method to find the span element that displays the count value and assert that it has an initial

value of 0. To make the assertion, we can rely on Jest's expect() helper to assert that the found element is not null (i.e., the element is present).

Testing initial count is 0

```
import React from 'react';
import { render } from '@testing-library/react';
import Counter from './Counter'

describe('Counter component', () => {
  it('renders with initial count of 0', () => {
    const { getByText } = render(<Counter />);
    expect(getByText('0')).not.toBeNull();
  });

  // ...
});
```

For our other tests, we can use React Testing Library's fireEvent() method to simulate the user behavior of clicking the "+" and "-" buttons and assert how the `Counter` component behaves in each of these conditions.

Testing when incrementing and decrementing count

```
import React from 'react';
import {
  render,
  fireEvent
} from '@testing-library/react';
import Counter from './Counter'

describe('Counter component', () => {
  it('renders with initial count of 0', () => {
    const { getByText } = render(<Counter />);
    expect(getByText('0')).not.toBeNull();
  });

  it('increases count on "+" button click', () => {
    const { getByText } = render(<Counter />);
    const incrementButton = getByText('+');

    fireEvent.click(incrementButton);
```

```
    expect(getByText('1')).not.toBeNull();
  });

  it('decreases count on "-" button click', () => {
    const { getByText } = render(<Counter />);
    const decrementButton = getByText('-');

    fireEvent.click(decrementButton);

    expect(getByText('-1')).not.toBeNull();
  });
});
```

If the tests all pass, we can be confident that our component is working as intended.

In a large-scale React application, we may encounter situations where we need to test more complex scenarios like asynchronous behavior (e.g., data fetching from APIs or handling user input with a delay) or testing components that rely on context or Redux stores. However, the basic format of our tests would often remain the same as what we've done in the `Counter` example—we'll still need to render the component and interact with it using utility functions provided by React Testing Library. Ideally, we'll always do so in a testing pattern known as **Arrange, Act, Assert**.

Arrange, Act, Assert

The Arrange, Act, Assert pattern is a widely adopted approach for structuring unit tests in a clear and concise manner. This pattern breaks down the test process into three distinct phases:

1. **Arrange**: In this phase, we set up the initial state of the component or system under test. This may involve rendering the component with specific props, creating mock objects, or initializing any required dependencies.

2. **Act**: During this phase, we perform actions or trigger events that simulate user interactions or system operations. These actions can include firing events like clicks or keypresses, calling functions, or updating the state of our components.

3. **Assert**: In the final phase, we verify that the component or function behaves as expected after performing the actions in the previous step. We can use assertions to check whether specific conditions hold true—such as checking if an element is present in the DOM, comparing updated state values against expected results, or validating that certain functions have been called with appropriate arguments.

Referencing the Arrange, Act, Assert pattern to our `Counter` component tests would look like this:

A reference to the Arrange, Act, Assert pattern in our `Counter` component tests

```
import React from 'react';
import {
  render,
  fireEvent
} from '@testing-library/react';
import Counter from './Counter'

describe('Counter component', () => {
  it('renders with initial count of 0', () => {
    // Arrange
    const { getByText } = render(<Counter />);

    /*
      Act - No action needed since we're testing
      initial render
    */

    // Assert
    expect(getByText('0')).not.toBeNull();
  });

  it('increases count on "+" button click', () => {
    // Arrange
    const { getByText } = render(<Counter />);
    const incrementButton = getByText('+');

    // Act
    fireEvent.click(incrementButton);
```

```
    // Assert
    expect(getByText('1')).not.toBeNull();
  });

  it('decreases count on "-" button click', () => {
    // Arrange
    const { getByText } = render(<Counter />);
    const decrementButton = getByText('-');

    // Act
    fireEvent.click(decrementButton);

    // Assert
    expect(getByText('-1')).not.toBeNull();
  });
});
```

End-to-end tests

Unlike unit tests, **end-to-end tests evaluate the complete functionality of an application by simulating real-world user interactions across multiple components and services**. These tests aim to ensure that the entire system, including its front-end, back-end, and any integrations with external systems, works as expected from a user's perspective.

Cypress is a popular end-to-end testing tool for web applications that offers a robust set of features for writing and running tests. It provides a straightforward API to simulate user interactions, supports real-time reloading, and offers time-travel debugging capabilities.

An updated counter example

To demonstrate how we can create end-to-end tests using Cypress for a React application, let's use an updated counter component example. In this version, we'll have a React application with two separate routes: an index route ("/") and a "/counter" route.

The main `App` component

```
import React from "react";
import {
  BrowserRouter as Router,
```

```
  Route,
  Switch,
} from "react-router-dom";
import HomePage from "./HomePage";
import CounterList from "./CounterList";

function App() {
  return (
    <Router>
      <Switch>
        <Route
          exact
          path="/"
          component={HomePage}
        />
        <Route
          path="/counter"
          component={CounterList}
        />
      </Switch>
    </Router>
  );
}

export default App;
```

The "/" route will display a `HomePage` component that has a welcome message and a link that, when clicked, redirects the user to the "/counter" route.

The `HomePage` component

```
import React from "react";
import { Link } from "react-router-dom";

const HomePage = () => (
  <div>
    <h1>Welcome to the Counter app</h1>
    <Link to="/counter">Go to Counter</Link>
  </div>
);

export default HomePage;
```

The "/counter" route will contain a parent component (`CounterList`) that fetches a list of items from an API and renders each item as a child component (`Counter`). Each `Counter` component will have the ability to increment or decrement its count property, which gets persisted to the server and synced with the client.

The `CounterList` component

```
import React, {
  useState,
  useEffect,
} from "react";
import Counter from "./Counter";

const CounterList = () => {
  const [items, setItems] = useState([]);

  useEffect(() => {
    fetch("/api/items")
      .then((response) => response.json())
      .then((data) => setItems(data));
  }, []);

  return (
    <div>
      <h1>Counter List</h1>
      {items.map((item) => (
        <Counter key={item.id} item={item} />
      ))}
    </div>
  );
};

export default CounterList;
```

The `Counter` component

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

const Counter = ({ item }) => {
  const [count, setCount] = useState(item.count);

  const updateCount = (change) => {
    fetch(`/api/items/${item.id}`, {
```

```
      method: "PUT",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        count: count + change,
      }),
    })
      .then((response) => response.json())
      .then((updatedItem) => {
        setCount(updatedItem.count);
      });
  };

  return (
    <div>
      <h3>{item.name}</h3>
      <div>
        Count:{" "}
        <span className="count">{count}</span>
      </div>
      <button
        className="increment"
        onClick={() => updateCount(1)}
      >
        +
      </button>
      <button
        className="decrement"
        onClick={() => updateCount(-1)}
      >
        -
      </button>
    </div>
  );
};
```

To set up end-to-end tests for this scenario, we can do the following:

1. Create Cypress test files for both the index route and "/counter" route.

2. Write tests for the index route to make sure it redirects users to the "/counter" route upon clicking a link or button.

3. Write tests for the “/counter” route that simulate user interactions such as fetching data, incrementing/decrementing count values, persisting changes to the server, and syncing updates with clients.

Notice how, in our end-to-end testing scenario, **we’re concerned with testing the application as a whole (i.e., from end to end)** as opposed to testing individual functions or components in isolation. The end-to-end tests we create will focus on the integration between components and the API service, as well as the overall user experience.

Testing the "/" route

We’ll first write a test for the index route ("/") that checks whether clicking the link redirects users to the “/counter” route. Like Jest, Cypress allows us to organize our tests in describe() and it() blocks.

Testing the "/" route

```
describe('/', () => {
  beforeEach(() => {
    cy.visit('/');
  });

  it('redirects to /counter on link click', () => {
    cy.get('a').click();

    cy.url().should('include', '/counter');
  });
});
```

`beforeEach()` is a Hook that runs before each test in the describe block. cy.visit('/') is a Cypress command that tells the browser to navigate to the index route ("/") of the web application. This setup ensures that every test in the suite starts from the index route.

In the test function, we use the cy.get() function to select the first anchor tag (`<a>`) it finds on the page, which is presumed to be the “Go to Counter” link. click() is a command that simulates a click event on the selected element.

We retrieve the current URL of the browser with cy.url() and then assert that the URL includes the string `'/counter'`. This checks that the

application has appropriately navigated to the “/counter” route after clicking the link.

Testing the "/counter" route

To test the “/counter” route, we’ll test the fetching of data from an API, incrementing and decrementing count values, persisting changes to the server, and syncing updates with the client. We’ll set up our test file to have the “/counter” route visited before every test is run.

Setting up tests for the "/counter" route

```
describe('/counter', () => {
  beforeEach(() => {
    cy.visit('/counter');
  });

  // ...
});
```

The components displayed in the “/counter” route interact with an API to fetch data from the server or update it when the user clicks the appropriate buttons. When running our tests in Cypress, we have the option to have our tests actually hit the server/API or to have these requests stubbed or mocked. Each of these behaviors has its advantages and disadvantages:

- **Real API Requests**: By using real API requests in our tests, we can ensure that our tests accurately simulate the actual behavior of our application in a live environment. However, this approach can make our tests slower and more susceptible to flakiness due to network issues or external dependencies.

- **Stubbed/mocked API Responses**: With stubbed or mocked API responses, our tests can run faster and be more stable since they don’t depend on external factors like network latency or server availability. However, this approach may not fully represent the behavior of our application when interacting with a real API, and we need to ensure that our stubs or mocks accurately mimic the actual API responses.

The Network Requests guide of the Cypress documentation recommends using real API requests sparingly and only when testing critical paths of an application (e.g., login, sign up, billing, etc.). Otherwise, stubbing API responses should be used for the vast majority of tests.

In the next section, we'll discuss how mocking and stubbing API requests can be beneficial, but for now, we'll assume that actual requests are being made to our API in our tests.

Stubbing the GET request for "/counter"

```
describe('/counter', () => {
  beforeEach(() => {
    cy.intercept('GET', '/api/items', [
      { id: 1, name: 'Item 1', count: 0 },
      { id: 2, name: 'Item 2', count: 0 },
    ]);
    cy.visit('/counter');
  });

  // ...
});
```

In the example above, we're intercepting any GET requests made to the `/api/items` endpoint with cy.intercept() and have it return a predefined array of items as the response. This way, we control the data displayed in our counter components during testing.

With our `beforeEach()` Hook established, we can begin to write some tests for the "/counter" route. We'll test that:

- The counter list fetches and displays data correctly.
- With the list of items fetched, we're able to increment and decrement the count value of each item when the right buttons are clicked.

We'll first write the test to check whether the fetched item counts are displayed correctly on the page. We'll use the `cy.get()` utility to select all elements with class "count" (which should correspond to each counter's displayed count value), then assert that there are two such elements present and their text content matches the expected values.

Testing data fetching and display in the "/counter" route

```
describe('/counter', () => {
  beforeEach(() => {
    cy.intercept('GET', '/api/items', [
      { id: 1, name: 'Item 1', count: 0 },
      { id: 2, name: 'Item 2', count: 0 },
    ]);
    cy.visit('/counter');
  });

  it('displays counters fetched from API', () => {
    cy.get('.count').should(($counts) => {
      expect($counts).to.have.length(2);
      expect($counts.eq(0)).to.contain.text('0');
      expect($counts.eq(1)).to.contain.text('0');
    });
  });

  // ...
});
```

In our next test, we'll simulate clicking the increment button of the first counter element by using `cy.get()` to select all elements with class "increment" and then calling `first()` to target the first element in the collection.

Testing increment functionality in the "/counter" route

```
describe("/counter", () => {
  beforeEach(() => {
    cy.intercept('GET', '/api/items', [
      { id: 1, name: 'Item 1', count: 0 },
      { id: 2, name: 'Item 2', count: 0 },
    ]);
    cy.visit("/counter");
  });

  it("displays counters fetched from API", () => {
    cy.get(".count").should(($counts) => {
      expect($counts).to.have.length(2);
      expect($counts.eq(0)).to.contain.text("0");
      expect($counts.eq(1)).to.contain.text("0");
    });
```

```
  });

  it("increases count on increment click", () => {
    const updatedItem = {
      id: 1,
      name: "Item 1",
      count: 2,
    };

    cy.get(".increment").first().click().click();

    cy.get(".count")
      .first()
      .should("contain.text", updatedItem.count);
  });

  // ...
});
```

In the test above, we stub a PUT request to `/api/items/1` and assert that the request payload contains an updated `count` property with an incremented value. We then simulate clicking the first increment button twice using `first().click().click()` and then check if the displayed count value of the first counter has been incremented as expected.

We can conclude the tests for our "/counter" route by writing a similar test for the decrement functionality.

Testing decrement functionality in the "/counter" route

```
describe("/counter", () => {
  beforeEach(() => {
    cy.visit("/counter");
  });

  it("displays counters fetched from API", () => {
    cy.get(".count").should(($counts) => {
      expect($counts).to.have.length(2);
      expect($counts.eq(0)).to.contain.text("0");
      expect($counts.eq(1)).to.contain.text("0");
    });
  });
```

```
  it("increases count on increment click", () => {
    const updatedItem = {
      id: 1,
      name: "Item 1",
      count: 2,
    };

    cy.get(".increment").first().click().click();

    cy.get(".count")
      .first()
      .should("contain.text", updatedItem.count);
  });

  it('decreases count on decrement click', () => {
    const updatedItem = {
      id: 2,
      name: 'Item 2',
      count: -2,
    };

    cy.get('.decrement').first().click().click();

    cy.get('.count')
      .first()
      .should("contain.text", updatedItem.count);
  });
});
```

Unlike the way we've written our unit tests earlier, our end-to-end testing approach requires a more integrated perspective. We're not concerned with testing individual functions or components but are more focused on validating the entire application's flow and interaction between various parts.

Integration tests

Integration tests generally focus on testing the interaction between multiple units or components in an application. While lacking a clear-cut definition, integration tests occupy a space between unit tests and end-to-end tests.

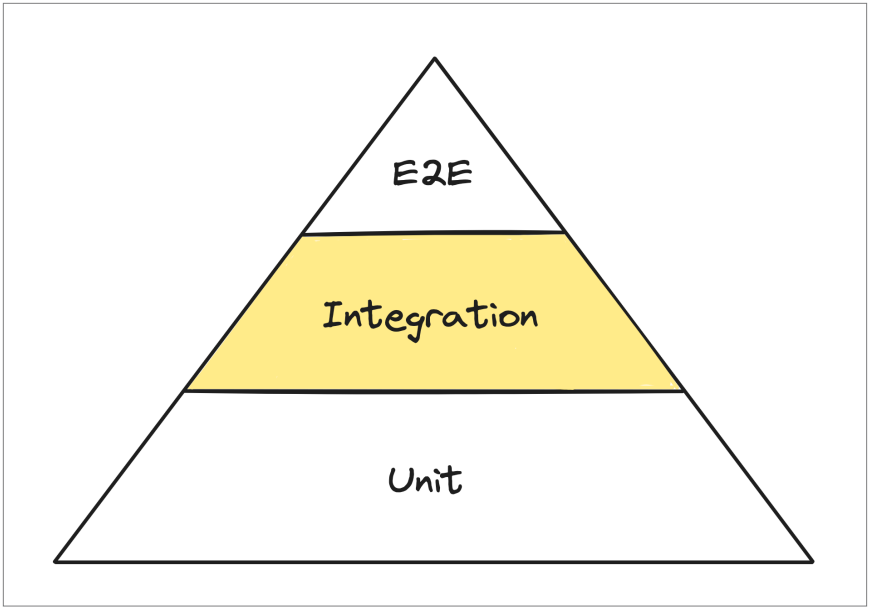


Figure 12-1. Simplified version of the testing pyramid from the Google Testing Blog

Let's consider the updated counter example once again, this time exploring how we could write an integration test that checks the functionality between the `Counter` component and the API service it relies on. In this case, we'll test whether the component successfully updates its markup when receiving new data from the API after a user clicks on the increment or decrement buttons. Here's a pseudo-code implementation of writing this test with Jest and React Testing Library.

Integration testing of the `Counter` component and an API service

```
import React from "react";
import {
  render,
  fireEvent,
  waitFor,
} from "@testing-library/react";
import { Counter } from "./Counter";

// Mock API service
jest.mock("./apiService", () => ({
  updateCount: jest.fn(),
}));
```

```
describe("Counter", () => {
  test("UI updates on item count++", async () => {
    const mockData = {
      id: 1,
      name: "Item 1",
      count: 2,
    };
    const { queryByText } = render(
      <Counter
        item={{
          id: 1,
          name: "Item 1",
          count: 0,
        }}
      />,
    );

    // Click increment button
    fireEvent.click(queryByText("+"));

    // Wait for API response
    await waitFor(() =>
      expect(
        require("./apiService").updateCount,
      ).toHaveBeenCalledWith(1, mockData.count),
    );

    // Check for UI update with new data
    expect(
      queryByText("Count: 2"),
    ).not.toBeNull();
  });
});
```

In the above code example, we're assuming our `Counter` component relies on a separate API service to interact with the server API it relies on. In this case, we're using Jest's mock() function to mock the implementation of an existing `updateCount()` API service method, which allows us to simulate a response when our component calls it. In the rest of the test, we're simulating a user clicking the increment button, waiting for the mocked API response, and then asserting that our component updated its UI with new data.

Notice in the above integration test example that we're testing a unit of functionality together by testing how the `Counter` component behaves when interacting with an API. Furthermore, we mock the behavior of the API instead of interacting with the actual API. This approach allows us to isolate the integration between the UI and the API without involving external dependencies, thus making our test more reliable and focused.

What tools we use in our tests doesn't dictate the type of tests we're writing. However, not mocking API requests in our tests, leveraging actual test environments, and seeding data in our back-end specific for testing are all features that resemble more of an actual end-to-end testing environment.

On the flip side, testing the interaction between components, mocking API requests, and focusing on specific parts of the application flow are characteristics more commonly associated with integration testing. These tests aim to provide a balance between the thoroughness of end-to-end tests and the efficiency of unit tests, ensuring that individual components work well together and function as expected within the context of the overall application.

Between unit, integration, and end-to-end tests, which of these tests should we be spending more time writing in a large-scale React application? We'll discuss this point in more detail in the **"How should we test our app?"** section after we briefly discuss one other type of test, snapshot tests.

Snapshot tests

Snapshot tests are a type of testing methodology that focuses on **capturing the UI output of a component** at a specific point in time and then **comparing it with future outputs** to ensure that no unintended changes have occurred. This approach is particularly useful for detecting potential regressions in the UI, especially when making updates or refactoring code.

Snapshot testing can be written with Jest by allowing us to generate snapshots of our components' rendered output and store them as reference files. We'll go through an example of snapshot testing a simple `Link` component that renders an anchor element, taking a URL

(`page`) and any child elements (`children`) as props to generate a clickable link.

A simple `Link` component

```
export default function Link({
  page,
  children,
}) {
  return <a href={page}>{children}</a>;
}
```

Here's an example of writing a snapshot test for the `Link` component with the help of an accompanying package, react-test-renderer.

Snapshot testing the `Link` component

```
import renderer from "react-test-renderer";
import Link from "../Link";

it("renders correctly", () => {
  const tree = renderer
    .create(
      <Link page="http://www.facebook.com">
        Facebook
      </Link>,
    )
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

When a snapshot test is first run, Jest will generate a snapshot file containing the serialized output of the `Link` component. This file is then saved alongside the test and serves as a reference for future tests.

Initial snapshot of the `Link` component

```
exports[`renders correctly 1`] = `
<a
  href="http://www.facebook.com"
>
  Facebook
</a>
```

```
`;
```

Assume we wanted to make a change to what the component outputs by having the component render a different link under the test.

Updating the test to render a different link

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
  const tree = renderer
    .create(
      <Link page="http://www.instagram.com">
        Instagram
      </Link>
    )
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

If we were to run our test with this change, Jest would point out the differences between the new output and the initial snapshot, alerting us to any unintended changes in the component's behavior.

Snapshot test result after updating the link

```
FAIL  src/Link.test.js
  ● renders correctly

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot ...

    - Expected
    + Received

      <a
    -   href="http://www.facebook.com"
    +   href="http://www.instagram.com"
      >
    -   Facebook
    +   Instagram
      </a>
```

```
7 |
8 | it("renders correctly", () => {
```

In a test failure like the above, we can see that Jest has identified a mismatch between the updated output and the stored snapshot. If this change was intentional, we can update the snapshot by simply running:

Updating Jest snapshots

```
jest --updateSnapshot
```

With the snapshots updated, Jest will update the snapshot reference file to reflect the latest change.

Updated snapshot of the `Link` component

```
exports[`renders correctly 1`] = `
<a
  href="http://www.instagram.com"
>
  Instagram
</a>
`;
```

With the snapshot reference updated, the test will now pass.

All snapshot tests follow a sequence where, initially, a test is run; if a pre-existing snapshot is present, it is compared with a newly generated snapshot to determine if the test passes or fails. If it fails, the differences are identified, indicating a discrepancy between expected and actual outcomes, which requires updating either the snapshot or the tested code. If there is no pre-existing snapshot, a new one is created and saved, leading to an automatically passing test.

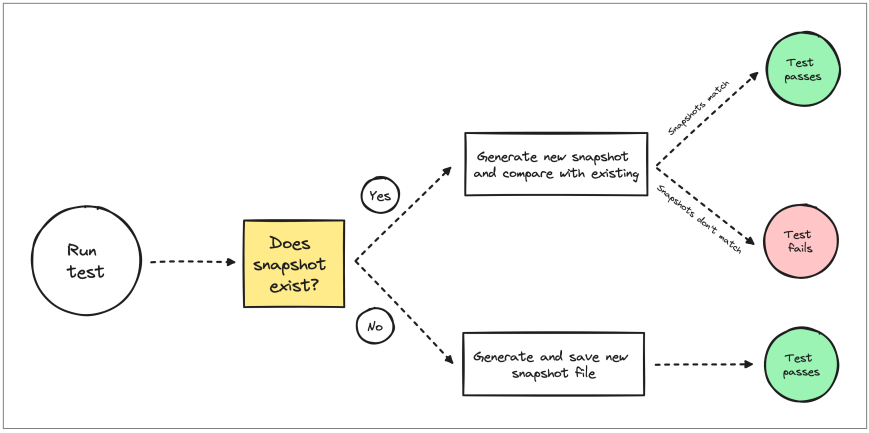


Figure 12-2. Snapshot testing process

How should we test our app?

Thus far, we have a good understanding of the different types of tests commonly written when testing React applications. With that said, which of these kind of tests should we focus on and/or write? Which should we ignore? How many tests should we write? Though there are no right answers to these questions, in this section, we'll spend a bit of time discussing some helpful guidelines that have worked best for us.

When should we write snapshot tests?

Snapshot tests are a bit of an outlier when compared to the other types of tests we've discussed (unit, integration, and end-to-end). This is because snapshot tests are best suited for specific use cases **where UI consistency is crucial, such as design systems or components with complex and intricate visual details**. When writing snapshot tests, it's best to:

- **Write snapshot tests sparingly**. While snapshot tests are great for verifying UI consistency, they shouldn't be the primary testing method. Rely on unit tests for testing individual functions and components, integration tests for verifying interactions between different parts of your application, and end-to-end tests for checking the overall user experience and functionality.

- **Keep snapshots small**. Focus on testing individual components instead of entire pages to simplify identifying and addressing discrepancies. Smaller snapshots are easier to understand and maintain, making it less likely for unintended changes to go unnoticed.

- **Document test cases**: When appropriate, document the test cases covered by each snapshot test. This helps other developers understand the purpose of the test and what output/UI the snapshot test is verifying.

Should we always aim for 100% code coverage?

Code coverage is a metric that indicates the percentage of a codebase covered by tests. Jest provides built-in support for tracking code coverage, and for more detailed reporting and tracking, code coverage tools like Istanbul can be used.



Figure 12-3. An example of Jest code coverage from the Jest homepage

While having comprehensive test coverage brings confidence in the stability of your application, **aiming for 100% test coverage is not always necessary and can sometimes be counterproductive**. It would

often lead to writing unnecessary tests or focusing on less critical parts of the application, consuming valuable time and resources.

In practice, it's often more valuable to aim for a specific coverage threshold that makes sense for you and your team rather than striving for 100% coverage. For example, you might decide that 60% or 80% test coverage is sufficient to give you confidence in the quality of your codebase.

Additionally, consider using tools like Codecov or Coveralls to monitor code coverage over time and set up alerts when test coverage falls below a certain threshold. This way, you can maintain awareness of any gaps in your tests and address them proactively as needed.

Should we write all our code with a test-driven development mindset?

Test-driven development (TDD) is an approach to development that involves writing tests for your code *before* you write the code itself. This can provide several benefits in the development process, such as:

1. **Improved code quality**: Writing tests before writing the actual code forces us to think about the expected behavior and outcomes of our code from the very beginning. This can often lead to well-designed and more robust code implementations.

2. **Easier debugging and maintenance**: When a test fails, it becomes easier to identify and fix the issue since you're already aware of the expected outcome. This can speed up debugging efforts and make maintaining your application more manageable in the long run.

3. **Faster development**: By focusing on one specific feature or component at a time, TDD encourages developers to break large tasks into smaller, more manageable chunks. This can help increase productivity and reduce the likelihood of technical debt accumulating over time.

Despite these advantages, adopting TDD for all code that is to be written is not a strict requirement for developing high-quality React applications.

You may prefer other testing methodologies or find that TDD doesn't align well with their existing workflows or project requirements.

Unit tests vs. Integration tests vs. End-to-end tests

When it comes to determining which of these tests should be emphasized more than the other, **there's no one right answer**. Different testing strategies exist that recommend and adapt the testing approach based on various factors, such as project size, complexity, team expertise, and available resources.

The *Testing Pyramid*, first introduced by Mike Cohn in his book "Succeeding with Agile," and further discussed in Google's popular article "Just Say No to More End-to-End Tests," emphasizes the importance of having a higher number of unit tests, followed by a smaller number of integration tests, and an even smaller number of end-to-end tests.

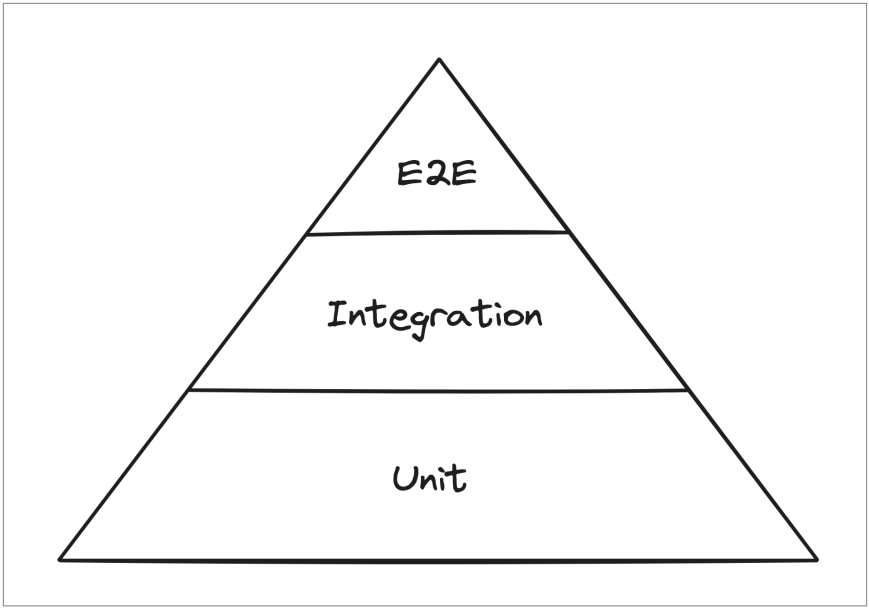


Figure 12-4. Simplified version of the testing pyramid from the Google Testing Blog

The *Testing Pyramid* approach aims to maximize test coverage while minimizing the time and resources spent on testing.

Another popular testing strategy that has worked for many is summarized beautifully in Guillermo Rauch's tweet "Write tests. Not too many. Mostly integration.".



Figure 12-5. Write tests. Not too many. Mostly integration.

The above strategy has also been popularly coined as *Testing Trophy* by Kent C. Dodds in his fantastic article similarly titled Write tests. Not too many. Mostly integration. or the *Test Diamond* adaptation by Ramona Schwering in her article Pyramid or Crab? Find a testing strategy that fits.

This testing strategy advocates for **focusing more on integration tests**, followed by unit tests and end-to-end tests. However, Kent C. Dodds further highlights the use of static typing (such as TypeScript) as playing a significant role in preventing type-related errors. These static types make the base of the *Testing Trophy* strategy.

By prioritizing integration tests, we can strike a balance between test confidence and development efficiency. Integration tests can provide a more comprehensive understanding of how components interact with each other in real-world scenarios without incurring the overhead associated with end-to-end testing.

One main takeaway from the above different strategies is to have end-to-end tests be written the least. End-to-end tests tend to be more time-consuming to create and maintain, and they often have a higher execution time compared to unit and integration tests. As a result, end-to-

end tests are often encouraged to be written only on core application flows such as user authentication, payment processing, and other critical pathways.

Whether you and your team decide to focus more on unit tests or integration tests would ultimately depend on the specific needs and context of your project, as well as the team's preference for depth and breadth of testing.

Tooling

The appropriate selection of tooling can make a big difference in the efficiency and effectiveness of your development workflow. Tools such as ESLint are instrumental in enforcing coding standards and preemptively identifying errors. Testing suites like Jest offer a robust framework to ensure that applications are both scalable and robust. Build systems, including Vite, Webpack, Turbopack, Parcel, and Rollup, play a crucial role in bundling code and optimizing performance. Additionally, tools like Git are essential for managing your codebase, tracking changes, and promoting collaboration among team members.



In previous sections of this book, we have touched upon some of these tools. However, in this chapter, we'll delve a little bit deeper, focusing on how these tools can play an important role in building and maintaining large-scale React applications.

Version control (Git)

Version control, with Git being the most prominent example, is a fundamental aspect of modern software development, especially for large-scale software applications. It enables collaborative development, allowing multiple developers to work on the same project without conflicts. Each contribution can be tracked, merged, and managed effectively, which is vital in team environments.

Additionally, Git supports branching and merging strategies that are crucial for managing features, fixes, and releases. Developers can work on different aspects of a project simultaneously in separate branches and later merge these changes back into the main codebase in a controlled and systematic manner.

Below, we'll discuss a few best practices to consider when setting up version control in a large-scale software application.

Consistent branching strategy

Implement a clear and consistent branching strategy, like feature branching, to manage the development of new features, fixes, and releases. This strategy should be well-documented and understood by all team members. As a simple example, this would involve:

- **Main Branch**: A branch that holds the most current, stable version of the application.
- **Feature Branches**: Separate branches created for each new feature, fix, or enhancement. These branches are derived from the 'main' branch and are named in a way that clearly identifies their purpose (e.g., 'feature-new-login,' 'fix-header-bug').

By implementing and adhering to a well-documented and consistent branching strategy, teams can ensure that the development process is

organized, efficient, and minimizes conflicts, which is particularly important in large-scale software development environments.

Review process

Before a feature branch is merged back into the main branch (or development branch), it should go through a clear code review process. This involves creating a pull request and having it reviewed by one or more team members, which helps identify issues and ensures code quality *before* new code is merged.

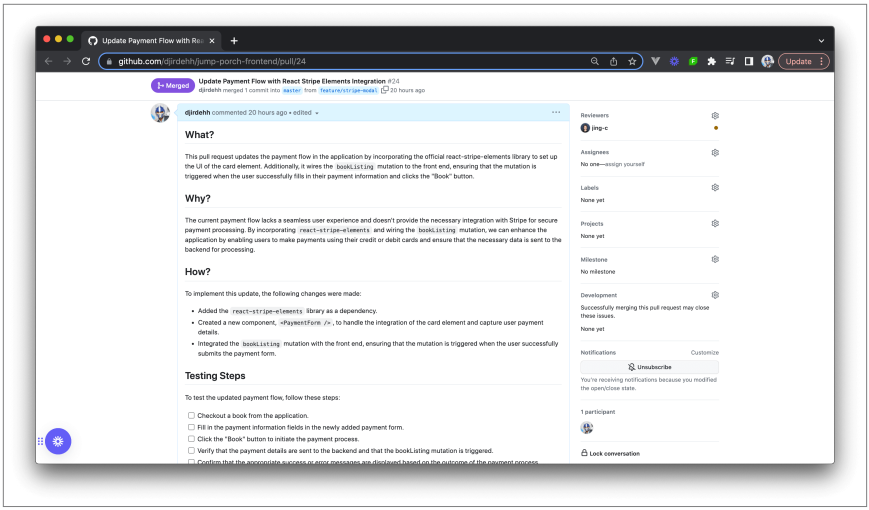


Figure 13-1. A detailed Pull Request description

Security and Compliance Checks

Many organizations have specific coding standards and practices that need to be followed, especially in regulated industries. For instance, in healthcare applications within the United States, developers must comply with HIPAA (Health Insurance Portability and Accountability Act) standards to ensure the protection of sensitive patient data.

To align with these diverse standards, implementing security and compliance checks within the Git workflow can be essential. This can include automated scanning for vulnerabilities in the code or dependencies, ensuring that the software remains secure against potential

threats. Continuing with the example of a healthcare application where compliance with HIPAA is mandatory, integrating a tool like SonarQube into the Git workflow could help automatically scan the codebase for security vulnerabilities, such as unencrypted patient data transmissions or insecure storage of sensitive information, which are direct violations of HIPAA standards.

Lastly, version control systems are also integral to implementing Continuous Integration and Continuous Deployment (CI/CD) pipelines, which brings us to the next section.

Continuous Integration

Continuous Integration, commonly abbreviated as CI, is the practice of automatically merging code changes from multiple developers into a single software project immediately after the changes are made. While the setup of CI can vary, as each organization adapts it to meet its unique workflow and requirements, we'll explore what a typical CI process usually involves.

In a standard Continuous Integration (CI) process, whenever a developer commits changes to the codebase, such as attempting to merge a new commit into the 'main' branch, these changes undergo automatic testing and integration with the existing code. This automation is enabled by a CI server, which keeps track of the version control repository for any new commits. Once a new commit is detected, the CI server initiates a series of automated tests (i.e., checks) on the code. This suite of checks often includes unit tests, integration tests, and other forms of automated checks, all designed to verify that the newly introduced code does not disrupt any existing functionality.

Should any of the relevant automated checks fail during a CI process, the system will halt the integration process. This precautionary measure is taken to prevent the potentially problematic code from being merged into the main codebase.

If these checks are successfully passed, the CI process advances to subsequent stages, ultimately leading to the merging of the code changes into the 'main' branch of the codebase.

Several CI servers are widely used in the software development industry today. Each offers unique features and integrations, catering to different requirements and preferences:

- GitHub Actions: Built into GitHub, GitHub Actions enables automation of workflows, including CI, directly within GitHub repositories. It's becoming increasingly popular due to its seamless integration into existing Github-hosted projects and ease of use.

- GitLab CI: Integrated into the GitLab platform, GitLab CI/CD is a powerful tool for the automation of the entire software development lifecycle. It's particularly convenient for projects already hosted on GitLab.

- Travis CI: Known for its ease of use, Travis CI is a hosted CI service used primarily in projects hosted on GitHub.

- CircleCI: This CI server offers robust performance and is known for its fast build times. CircleCI supports both cloud-based and on-premise deployment.

- Jenkins: An open-source automation server, Jenkins is highly popular due to its vast plugin ecosystem, which allows it to integrate with almost any tool in the CI/CD toolchain.

- Any many others.

Establishing a robust CI process for large-scale React applications is highly encouraged for many of the reasons mentioned above. For React web applications, the checks that can be made in a CI pipeline typically involve the following:

- Linting, formatting, and code style checks (e.g., with ESLint and Prettier)
- Unit, Integration, and E2E tests (e.g., with Jest and Cypress)
- TypeScript typings
- Accessibility testing (e.g., with tools like Axe)

Bundlers

In modern React applications, the vast majority of code that we write is done within an environment specifically established to make our coding journey easier. This can be seen with how we leverage React itself to create interactive user interfaces, styled-components to apply styles, TypeScript to ensure type safety, and so forth.

Browsers don't recognize any of these aforementioned tools and technologies. If we attempted to run any of this code directly in our browser, it would fail to execute properly. This is because browsers only support HTML, CSS, and JavaScript, necessitating a need to convert our development code into browser-compatible formats. This is primarily where bundlers come in.

Bundlers like Vite, Webpack, and Turbopack are designed to bridge the gap between the rich, complex codebases we develop and the more limited languages that browsers can interpret directly. They do this by taking code written in languages or syntaxes that browsers cannot understand (e.g., JSX, TypeScript, SASS, etc.) and compiling them to plain HTML, CSS, and JavaScript—code executable in a web browser environment.

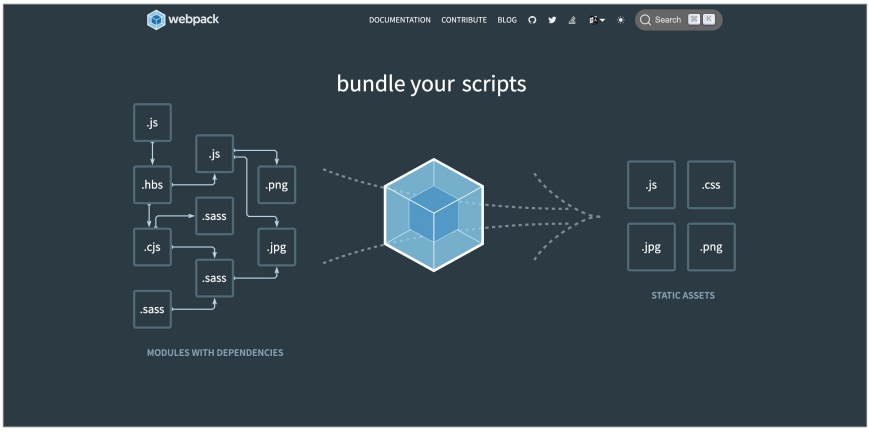


Figure 13-2. Illustration of Webpack's module bundling process from https://webpack.js.org/

Though code compilation and transformation is a core feature of these bundlers, these tools also provide additional capabilities that are essential in modern web development:

- Vite expands beyond being just a bundler; it also serves as a development server. It leverages Rollup for production builds and utilizes native ES modules during development for faster updates and a more streamlined development experience.

- Beyond HTML, JavaScript, and CSS, bundlers efficiently manage other assets, such as images and fonts.

- Bundlers play a key role in optimizing the performance of web applications by minifying code, optimizing image sizes, and facilitating techniques like lazy-loading and code-splitting to reduce the overall size of the application, leading to faster loading times.

- Tree shaking is an important feature provided by bundlers, where they analyze the dependency tree and remove any unused code from the final bundle. This results in a smaller, more efficient bundle, improving load times and overall application performance.

When working within large-scale React applications, bundlers facilitate efficient dependency management and support for scalable architectures, making them a necessary component of the modern web development workflow.

Linting

Linting is the process of enforcing coding standards and identifying errors/ inconsistencies during development and is essential for maintaining code quality. For linting TypeScript (and JavaScript) code, ESLint is the widely preferred library. It offers configurability, is simple to implement, and includes a collection of default rules. The Getting Started with ESLint documentation highlights the steps needed to set up ESLint within a project.

Below are some good practices to keep in mind when establishing linting in a large-scale React application:

- Utilize (or reference) established ESLint rule sets like Airbnb's ESLint configuration, which is widely adopted and covers a comprehensive set of rules that enforce good coding practices.

- Tailor your ESLint setup by adding custom rules or modifying existing ones. This allows you to enforce specific coding standards or practices that are unique to your project or organization.

- Develop and maintain a coding style guide for your engineering team and organization. This guide should be reflected in your ESLint configuration to ensure consistency across your codebase. The style guide would also help new team members understand the coding standards and maintain them.

- Use additional tools like Prettier and Husky to enforce code formatting, spell-checking, and linting before committing or pushing code.

    - Prettier is a code formatting tool that can make your code more readable with customizable rules.

    - Husky enforces the execution of lints and tests before committing or pushing code.

- Ensure that linting checks are a part of your CI pipeline. Set up your CI tools to run ESLint on every pull request or code commit. If linting fails, it should prevent the merge of code, ensuring that only code adhering to your standards gets integrated into your main branch.

- Periodically review and update your linting rules to adapt to new coding standards, best practices, and changes in the language or framework you are using. Keeping your rules up-to-date ensures that your linting process remains relevant and effective.

In large-scale React applications, establishing a robust linting process with tools like ESLint is not just about maintaining code quality; it's about creating a sustainable and efficient workflow. It helps ensure that as an application grows and the team expands, the code remains consistent, understandable, and adheres to the intended best practices.

Logging and performance monitoring

Web performance monitoring and logging are critical for large-scale web applications. These tools help developers and system administrators identify and diagnose performance issues, errors, and other problems that can impact the user experience.

Logging

Logging, primarily error logging, is crucial for identifying and diagnosing issues that can impact user experience. With the help of logging tools, developers can track and diagnose errors that occur in production, such as unhandled exceptions, syntax errors, and runtime errors. This information can help developers identify bugs and fix them quickly, thereby improving user experience and reducing downtime.

Here are some approaches to logging you may find useful:

- **Use a centralized logging solution**. A centralized logging solution helps to collect and store logs from various parts of your application in one place. This makes it easier to search for and analyze logs, which is essential for debugging and monitoring your application. Some popular logging solutions used in large-scale organizations include Splunk, Datadog, Sentry, and the ELK Stack (Elasticsearch, Logstash, Kibana).

- **Be selective with what you log**. Logging too much information can have a negative impact on your ability to search and analyze logs. Be selective about what you log, and consider filtering out unnecessary information.

- **Use structured logging**. Structured logging involves logging data in a structured format, such as JSON, rather than plain text. This makes it easier to search for and analyze logs and can help you identify patterns and trends in your application's behavior.

- **Include context in your logs**. Including context in your logs, such as the user ID, request ID, and application version, can help you to identify the source of a problem more quickly. Keep in mind privacy and security considerations here.

- **Monitor your logs**. Monitoring your logs in real time can help you to identify and respond to issues quickly before they have a significant impact on your application's performance.

Performance monitoring

Performance monitoring is the ability to continuously track and evaluate the efficiency and speed of a system or application. This process involves collecting, analyzing, and reporting data related to an application's performance, such as response times, server load, and resource utilization.

A primary benefit of web performance monitoring is the ability to identify and diagnose slow pages and long load times. With the help of performance monitoring tools, developers can identify bottlenecks and optimize the code, server, and database to improve overall performance. In addition to real-time logging and error logging, Datadog, New Relic, and Splunk are some tools that offer capabilities for system performance tracking and monitoring.

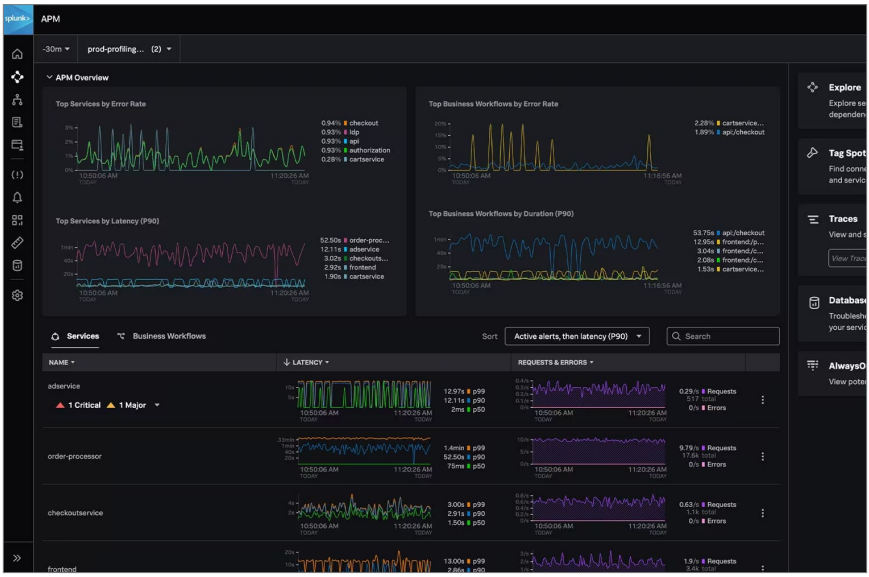


Figure 13-3. A Splunk performance monitoring dashboard tracking different services

It's also important to note that performance monitoring and error logging can help identify issues with third-party services and APIs. For example, if a third-party service is experiencing downtime or is slow to respond, it can impact the performance of your application. Performance monitoring and error logging can help identify these issues and provide insights into how to resolve them.

Finally, performance monitoring and error logging are crucial for maintaining the health and stability of large-scale web applications. With the help of these tools, developers and system administrators can proactively identify and diagnose issues before they impact the user experience. This can reduce downtime, improve stability, and ensure that the application meets the needs of its users.

Wrap up

The tooling ecosystem for React applications is vast and integral to the success of these projects. From ESLint ensuring code quality and TypeScript to enhance code reliability and maintainability to Jest guaranteeing well-tested applications, each tool plays a distinct role.

Additionally, version control with Git, continuous integration with platforms such as GitHub Actions or Jenkins, and bundling/development environments like Webpack or Vite create a strong infrastructure that streamlines development, enhances team collaboration, and guarantees the release of dependable software in React projects.

Technical Migrations

When building large-scale web applications, technical migrations become an inevitable part of the development process. As an application grows and technologies evolve, we often need to migrate existing code to newer, more efficient technologies or methodologies.



A technical migration refers to the process of moving an existing system, application, or dataset from one technology stack, platform, or version to another. They can be driven by a need to improve performance, maintainability, and security or to keep up with evolving industry standards.

Migrations can involve various changes and upgrades, such as:

- Upgrading or replacing dependencies, libraries, or frameworks.
- Moving to a new programming language or platform.
- Migrating to a new version of a database or data storage system.
- Adopting new tools or methodologies for testing, monitoring, or debugging.
- Refactoring existing code to improve performance, maintainability, or scalability.
- Migrating to a new deployment infrastructure, such as a cloud-based system.
- Updating the application's architecture to support new features or business requirements.
- Implementing new security measures to protect against threats or vulnerabilities.
- Etc.

Technical migrations can sometimes be a complex and time-consuming process as they can require careful planning, testing, and execution to ensure a smooth transition. They may involve the need to work with multiple teams or stakeholders, such as developers, QA engineers, project managers, and business analysts.

In this chapter, we'll discuss different types of migrations as well as some good practices on how to plan and approach technical migrations within a large web application.

Different migration strategies

Maintaining and upgrading large, legacy JavaScript web applications can be a daunting task, especially when they are built using older libraries or frameworks and exhibit spaghetti code or poor documentation. In the following section, we'll explore various strategies to approach the maintenance and migration of such applications, discussing the pros and cons of each method.

Good Migration

A Good Migration involves completely rewriting the application code and ensuring all its components are fully migrated to the new framework

or technology. This approach allows developers to start fresh, take advantage of modern tools and techniques, and create a more efficient application.

However, a complete rewrite can be time-consuming and labor-intensive and cause disruptions in the availability of the application. This method is more suited for small projects, less critical applications, or those that don't change frequently.

Fast Migration

A Fast Migration offers an alternative to the full rewrite approach since it involves dividing an application into parts or pieces and migrating each part incrementally. This approach allows developers to release migrated components as they are completed, providing faster feedback and reducing the time the application is unavailable.

Despite its advantages, Fast Migrations can lead to conflicts in tools, libraries, dependencies, and frameworks. Additionally, supporting different versions of the same tool can cause problems, particularly when global policies (like CSS cascade logic) are in place.

Strangler Application

The Strangler Application approach, named after the "Strangler Fig" pattern, involves identifying the edges of an existing web application and adding new functionalities using a newer framework until the old system is "strangled." This method reduces potential risks associated with migrating an app by gradually replacing old components with new ones in a controlled and systematic manner. The process is akin to how a strangler fig gradually envelops and outcompetes an existing tree.

To follow this approach, developers must divide the app into different pieces of related imperative code and wrap them in new components (e.g., React components). While this method doesn't add any additional behavior, it enables the creation of components that render existing content.

Hybrid approach

In some cases, a combination of the above strategies might be the most suitable approach. For example, developers could use the Good Migration approach for specific critical components while employing the Fast Migration approach for less critical ones. This hybrid approach enables a more flexible migration process tailored to the specific needs and constraints of the application.

When working on a large, legacy JavaScript web application, it's crucial to carefully plan the maintenance and migration process. Developers should assess the specific requirements and constraints of the application and choose the most appropriate strategy or combination of strategies to ensure a smooth and efficient migration process. By doing so, they can minimize disruption, reduce potential risks, and optimize the performance and usability of the application in the long term.

For an example deep-dive on how to migrate from an older JavaScript tool to a much newer framework, Smashing Magazine has a great write-up on this with How To Migrate From jQuery To Next.js.

Migration strategy

A good migration strategy, ideally, should encompass a comprehensive and methodical approach, which can oftentimes include the following key elements:

Understanding the scope of the migration

Before embarking on a technical migration, it's important to understand the scope of the migration. This involves identifying the components or features that need to be migrated, the technologies involved, and the impact on the application's architecture, performance, and user experience.

Developing a migration plan

Once we've identified the scope of the migration, we can work towards developing a migration plan. This involves defining the steps involved in

the migration, the timeline, and the resources required. It's important to develop a plan that is realistic and achievable, taking into account the complexity of the migration and any potential risks or challenges.

A migration plan should often cover migration steps for both code as well as database/schema migrations, including dependencies between them.

Implementing incremental migrations

To minimize disruption to the application and reduce the risk of errors or bugs, it's often best to implement incremental migrations. This involves migrating small pieces of the application at a time and testing and shipping each piece thoroughly before moving on to the next.

Monitoring and optimizing performance

During the migration process, it's important to monitor and optimize the performance of the application. This may involve using tools like Chrome DevTools to identify and optimize performance bottlenecks and implementing techniques like lazy-loading and code-splitting to improve performance for newly migrated code and components.

Testing thoroughly

Testing is a critical aspect of any technical migration. It's important to test each component thoroughly before moving on to the next. Integration testing should also be performed to ensure that combined components function correctly together.

Communicating with stakeholders

During the migration process, it's important to communicate with stakeholders, including end-users, developers, and managers. This involves providing regular updates on the progress of the migration, addressing any concerns or questions that arise, and soliciting feedback to ensure that the migration meets the needs of the business and the users.

By following these principles and strategies, we can approach technical migrations for large-scale applications in a way that minimizes disruption, reduces risk, and ensures that the migration meets the needs of the business and the users.

Codemods

When migrating large-scale applications, manually updating every bit of code can be a time-consuming and error-prone process. Codemods (short for code modifications) are scripts that automate the process of changing code in JavaScript applications and can significantly streamline this process.

Codemods can automate certain aspects of the migration process by scanning the codebase and making the necessary changes automatically. They can be used to update syntax, change imports, update configuration files, and more.

Example: Convert all relative imports to absolute imports

Assume we have a large JavaScript codebase with many files that use relative imports:

Importing XYZ in a relative manner

```
import XYZ from '../../../../components/XYZ'
```

As a project scales, relative paths can sometimes become cumbersome and less readable. During the migration process, assume we want to convert all relative imports within the `src/` directory to absolute imports, assuming a proper module resolution configuration.

Importing XYZ in an absolute manner

```
import XYZ from 'src/components/XYZ'
```

To achieve this, we can first write a codemod script to help achieve the following:

- **Detection**: Identify all files with relative import statements.

- **Path Analysis**: Analyze the current file path and the relative import path to calculate the absolute path.
- **Replacement**: Replace the relative path with the calculated absolute path.

Github user Phenax has written a script for the above in a public Github Gist—absolute-import-codemod-transform.js.

Phenax's absolute-import-codemod-transform.js script

```
const path = require("path");

const SOURCE = "src";
const SOURCE_PATH = path.resolve(SOURCE) + "/";

const getAbsolutePath = (
  importPath,
  filePath,
) => {
  return path
    .resolve(path.dirname(filePath), importPath)
    .replace(SOURCE_PATH, "");
};

const replaceImportPath = (j, node, filePath) =>
  j.importDeclaration(
    node.value.specifiers,
    j.literal(
      getAbsolutePath(
        node.value.source.value,
        filePath,
      ),
    ),
  );

export default function (file, api) {
  const j = api.jscodeshift;
  const filePath = file.path;

  return j(file.source)
    .find(j.ImportDeclaration)
    .replaceWith((node) =>
      replaceImportPath(j, node, filePath),
    )
```

```
    .toSource();
}
```

With the codemod script prepared, we can then leverage the jscodeshift toolkit, prepared by the Facebook team, to run the codemod over multiple JavaScript or TypeScript files.

jscodeshift CLI command to execute the codemod

```
jscodeshift -t ./absolute-import-codemod-\
transform.js src/**/*.js
```

When done correctly, the above command will automatically update all matched files, converting relative imports to absolute imports without us having to manually make that change in every file!

Example: Convert ES6 classes to functional components

The task of converting ES6 classes to functional components in React could be another excellent use case for codemods, especially when dealing with a large codebase. This conversion can streamline the code and embrace more modern React practices since class-based components are no longer the recommended approach to how components should be defined.

Writing a codemod to convert *all* class-based components to functional components can be difficult due to how different class and functional components are in their structure and behavior. To make things simpler, we can create a codemod script to convert ES6 class components that only have a `render()` method and safe properties (like statics and props) and do not use refs.

Example of ES6 class component that will be converted

```
import React, { Component } from 'react';

class SimpleComponent extends Component {
  static defaultProps = {
    greeting: 'Hello',
  };

  render() {
```

```
    const { greeting, name } = this.props;
    return (
      <div>
        {greeting}, {name}!
      </div>
    );
  }
}

SimpleComponent.propTypes = {
  name: PropTypes.string.isRequired,
};

export default SimpleComponent;
```

To achieve the above, we can write a codemod script to help achieve the following:

- **Detection**: Identify all ES6 class components in the codebase.
- **Analysis**: Examine the structure of the class component to determine if it meets the criteria for conversion. This involves checking for the presence of a `render()` method, static properties like `defaultProps` and `propTypes`, and ensuring that the class does not use complex features like refs, state, or lifecycle methods beyond `render()`.
- **Transformation**: Convert the identified class components into functional components.

Here is a pseudocode implementation of this codemod:

Codemod to convert certain ES6 classes to functional components

```
const { parse } = require("recast");
const { default: j } = require("jscodeshift");

const canConvertToFunctionalComponent = (
  classDeclaration,
) => {
  /*
    Check if the class has only a render method,
    statics, and props and does not use refs,
```

```
    state, or other lifecycle methods.

    Return true if it's a candidate for
    conversion, false otherwise.
  */
};

const convertClassToFunction = (
  classDeclaration,
) => {
  /*
    Convert the class component to a functional
    component based on the criteria defined in
    `canConvertToFunctionalComponent`.
  */
};

export default function (file, api) {
  const j = api.jscodeshift;
  const root = j(file.source);

  return root
    .find(j.ClassDeclaration)
    .filter((path) =>
      canConvertToFunctionalComponent(path.node),
    )
    .replaceWith((path) =>
      convertClassToFunction(path.node),
    )
    .toSource();
}
```

With the codemod prepared, we can then run the codemod with the jscodeshift toolkit to convert the appropriate class-based components to functional components.

Functional component equivalent of `SimpleComponent` example shared above

```
import React from "react";
import PropTypes from "prop-types";

function SimpleComponent({
  greeting = "Hello",
```

```
  name,
}) {
  return (
    <div>
      {greeting}, {name}!
    </div>
  );
}

export default SimpleComponent;
```

Codemods are especially useful when migrating between major versions of frameworks or libraries. These updates often involve significant changes to syntax, API changes, and other breaking changes. Codemods can automate many of these changes, making the migration process faster, more consistent, and less error-prone.

In addition to making code migrations easier, codemods can also be used for other tasks, such as refactoring code or applying best practices across a codebase.

The Role of Generative AI

Generative AI refers to a type of artificial intelligence that is capable of generating new content, ranging from text and images to code and music. It operates by learning from a vast dataset of existing examples and then uses that learned information to generate new, original material that is similar in style or content to the input it was trained on. Tools like ChatGPT, Gemini, Github Copilot, and Google's Project IDX are examples of generative AI tools that have gained significant popularity in the last couple of years.

Generative AI can play a role in helping with code migrations by providing capabilities like code generation, code completion, and code translation.

Code Generation

Generative AI tools can be used to generate new code based on natural language prompts, making it possible to quickly create new code in the target language during a migration process. This can sometimes save

developers time and effort as they don't need to manually rewrite the entire codebase. An example of such a tool is OpenAI's ChatGPT.

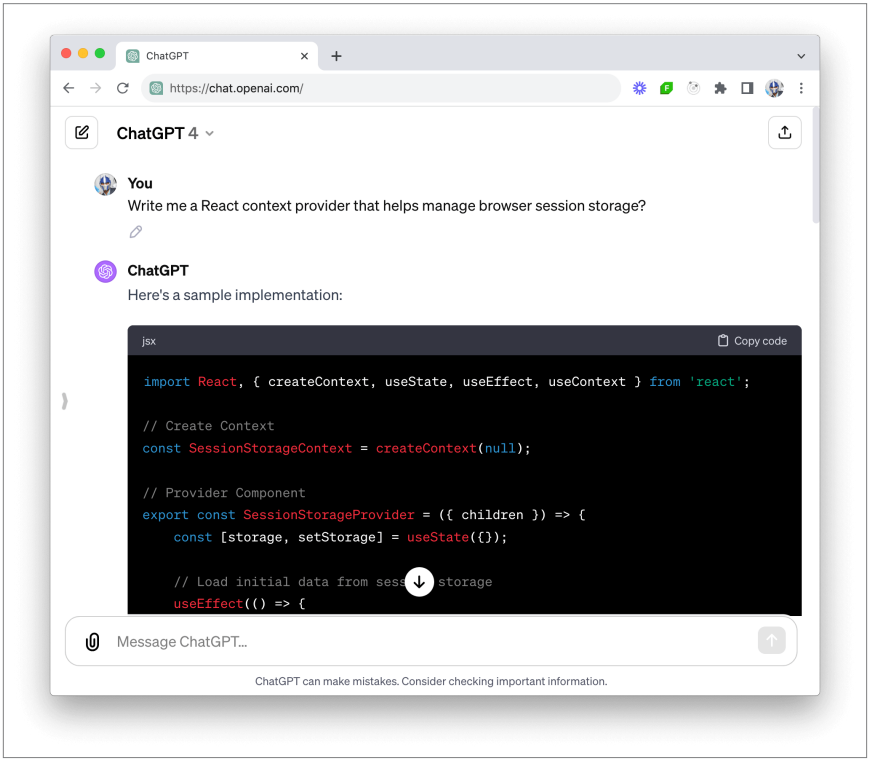


Figure 14-1. Prompting ChatGPT to create a React context provider

Code Completion

Generative AI tools can also complete partially written code, which can be useful during code migrations when developers need to fill in gaps or adapt existing code to the new language or framework. Tools like GitHub Copilot use OpenAI's Codex model to offer code suggestions right from a developer's editor.



Figure 14-2. Github Copilot autocompleting a React functional component

Code Translation

Generative AI models can even be employed to translate code from one programming language to another, which can be helpful during code migrations from one language to another. These models can understand the context of the source code and linguistically construct the equivalent code in the target language. Here's an example of asking Github Copilot to translate a `binarySearch` function from JavaScript to Python.

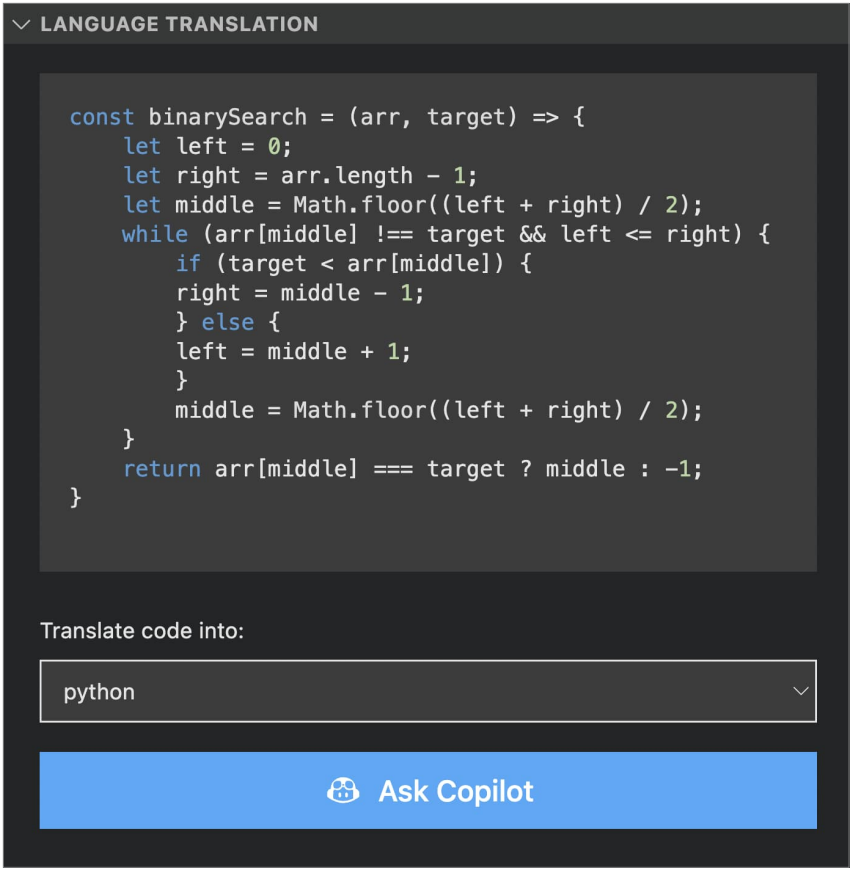


Figure 14-3. Language translation with Github Copilot. From Rizèl Scarlett's article—Use Copilot to Write and Translate a Binary Search Algorithm

Despite generative AI being a new technology, it offers capabilities to reduce the manual effort and time spent on code migrations. With that said, the technology does come with some challenges and limitations:

- Generated code may not always be accurate or optimal, requiring manual review and adjustments.
- Generative AI models can require large amounts of high-quality data if there is a need to train and refine complex algorithms.
- The technology is new and still evolving, and its effectiveness in various migration scenarios may vary.

While generative AI is a powerful tool in code migrations, **it should be used as an aid rather than a replacement for human expertise**. While these tools can significantly accelerate parts of the migration process, generated code and code changes should never be accepted without thorough review and validation by human developers.

TypeScript

TypeScript was introduced in 2012 by Microsoft as a **statically typed** language to help make JavaScript code more secure. For those unfamiliar with TypeScript, its main feature is the ability to specify types for variables, parameters, and functions in our code. It does so by being a strongly typed superset of JavaScript where types are checked during *compile-time* (i.e., when code is being *compiled*) as opposed to *run-time* (i.e., when code is being run).



In 2022, TypeScript was recognized as the fastest-growing programming language, with its usage nearly tripling from 2017 to 2022, according to the JetBrains State of Developer Ecosystem 2022. Before we explore the

benefits TypeScript offers in a React application, let's first discuss how it enhances the "type-safety" of a JavaScript codebase.

Type safety

TypeScript's standout feature is its static type checking, which helps catch type-related errors during **compilation** rather than at runtime. To understand this, it's best to first understand how JavaScript is a dynamically typed language.

In JavaScript, we're able to assign a certain data type to a variable and, later on, assign *another* data type to that same variable. For example, we can create a variable called `number` that is initialized with a number value of `5`.

Initializing a value of 5

```
let number = 5;
```

Later on in our code, we're able to reassign the value of the `number` variable to a string of `"5"`.

Reassigning the value of the variable to `"5"`

```
let number = 5;

// Some code here...

number = "5";
```

This is an example of how JavaScript is a *weakly typed* language—it doesn't enforce a fixed data type for its variables. In contrast to strongly typed languages, where a variable's type is declared and doesn't change, JavaScript permits variables to change their type over their lifetime. This flexibility can be both a benefit and a drawback.

The benefit is that developers can write code faster and with more ease. We don't have to explicitly specify the type of a variable when we declare it, and operations that mix types (like adding a `string` and a `number`) won't immediately cause errors. This dynamic typing can

speed up the initial development process and make JavaScript more beginner-friendly.

On the other hand, the drawback is that it can lead to unintended bugs. When variables change types unexpectedly or when functions that expect a certain type receive another, the result can be unpredictable. Errors might not appear until runtime, making them harder to catch during the development process. This can lead to scenarios where a function behaves differently than expected, and debugging becomes more challenging.

This drawback is one of the main reasons why TypeScript, a superset of JavaScript, has gained significant popularity. By introducing static typing, it brings the best of both worlds, allowing developers to write JavaScript code but with optional type annotations.

Revisiting the above example in TypeScript, when we initialize the `number` variable, we can state that the static type of the `number` variable is to be of type `number`.

Assigning a data type of `number` in TypeScript

```
let number: number = 5;
```

If we try to reassign the `number` variable to a value of a different type (e.g., a string of `"5"`), the TypeScript compiler will emit an error along the lines of `Type 'string' is not assignable to type 'number'`.

Attempting to reassign a variable to a different type in TypeScript

```
let number: number = 5;

// Some code here...

/*
  Type 'string' is not assignable to type
  'number'.ts(2322)
*/
number = "5";
```

TypeScript is smart enough to recognize that the `number` variable is of the `number` type even if we were to not explicitly specify its static type.

TypeScript will recognize an issue with the code we've written above *when the code is being compiled* to JavaScript (i.e., during compile-time). This provides the benefit of safety and predictability since we're able to catch potential type-related errors during development compile-time rather than runtime.

In addition to introducing type safety, TypeScript also often provides:

- **A better developer experience**: Editors like VSCode offer improved IntelliSense , auto-completion, and refactoring capabilities when working with TypeScript code.

- **Easier maintenance**: As projects grow, it becomes harder to make changes without inadvertently breaking something. TypeScript's static typing makes it easier to navigate and refactor the codebase while avoiding the introduction of regressive bugs and issues.

- **Better collaboration**: Due to the self-documenting nature of types and type definitions, TypeScript oftentimes makes it easier to understand the code written by others.

Build Tools & TypeScript

It's important to keep in mind TypeScript is only a **development tool**. TypeScript is not recognized by browsers and always needs to be compiled to valid JavaScript. When working within large React applications, this compilation process is often done within build tools like Vite or Webpack. These build tools further leverage tools like esbuild or ts-loader to transpile TypeScript code into plain JavaScript.

If you're about to begin a new React TypeScript application, we recommend using a framework like Vite or NextJS that comes with built-in TypeScript support. With these tools, you won't have to configure the TypeScript compiler or Webpack yourself. Just create a TypeScript configuration file (`tsconfig.json`) file, and these frameworks will take care of the rest.

Configuration & Linting

tsconfig.json

The tsconfig.json file is a JSON file that is created and kept at the root of a TypeScript app and indicates the parent directory is a TypeScript project. With this file, we can customize our TypeScript configuration and guide our TypeScript compiler with the options required to compile the project.

An example `tsconfig.json` file for a React application

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "CommonJS",
    "jsx": "react",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*.ts", "src/**/*.tsx"],
  "exclude": ["node_modules"]
}
```

Let's quickly break down the options used in the above example configuration:

- `target`: The target ECMAScript version, such as ES6.
- `module`: The module system, for instance, CommonJS.
- `jsx`: The handling method for JSX files.
- `outDir`: The directory where compiled JavaScript files will be placed.
- `rootDir`: The root directory for input files.
- `strict`: A flag enabling all strict type-checking options (see below for details).
- `esModuleInterop`: A setting that allows emission of ES6-style imports/exports in the generated code.

- `skipLibCheck`: A flag to skip type-checking of declaration files.
- `forceConsistentCasingInFileNames`: A check to ensure consistent casing in file names.
- `include`: An array of filenames or patterns to include in the program.
- `exclude`: An array of filenames or patterns to exclude when resolving `include`.

The `strict` option, when set to `true`, enables a wide range of type-checking behaviors that make the type system more rigorous. If we only want to enable a subset of these strict checks, we can disable the `strict` option and manually set the individual options we're interested in.

Enabling individual type-checking options manually

```
{
  "compilerOptions": {
    // ...
    "strict": false,
    "noImplicitAny": true,
    "strictNullChecks": true,
    // ...
  },
  // ...
}
```

`noImplicitAny` and `strictNullChecks` are examples of such individual type-checking options that can be enabled separately to enforce specific aspects of the type system.

`noImplicitAny` ensures any variable or parameter without a type (that can't be inferred by TypeScript) gets flagged as an error unless it's explicitly typed. This helps catch untyped variables that could introduce runtime errors.

`strictNullChecks` enforces stricter `null` and `undefined` checks. This means we can't assign `null` or `undefined` to a variable unless we explicitly declare its type as such. This helps in avoiding common null reference errors.

These examples and the above configuration only surface a small number of options we can dictate and control in our compiler. To see a full list of each available option and what it does, be sure to reference the TSConfig Reference section of the TypeScript documentation.

ESLint

Linting is the process of enforcing coding standards and identifying errors/ inconsistencies during development and is essential for maintaining code quality. For linting TypeScript (and JavaScript) code, ESLint is the widely preferred library. It offers configurability, is simple to implement, and includes a collection of default rules.

To set up and configure ESLint for a TypeScript project, we'll need to install necessary dependencies like:

- eslint: the core ESLint library
- @typescript-eslint/parser: a parser enabling ESLint to analyze TypeScript code)
- @typescript-eslint/eslint-plugin: a plugin that, when used together with `@typescript-eslint/parser`, incorporates numerous ESLint rules specific to TypeScript.

Installing ESLint dependencies

```
yarn add -D eslint \
@typescript-eslint/parser \
@typescript-eslint/eslint-plugin
```

Once the necessary packages are installed, we can create an `.eslintrc` file at the root of our React TypeScript project to configure our application's linting rules.

An example `.eslintrc` file for a React/TypeScript application

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": "latest",
    "sourceType": "module"
```

```
  },
  "extends": [
    "plugin:@typescript-eslint/recommended"
  ],
  "rules": {
    "indent": "off",
    "@typescript-eslint/indent": "off"
  }
}
```

In the above example configuration:

- `parser`: Specifies @typescript-eslint/parser to be used by ESLint.
- `parserOptions`: Allows us to specify the language options we want ESLint to support.
- `extends`: Extends the linting rules specified in the @typescript-eslint/recommended plugin.
- `rules`: Allows us to specify the specific ESLint rules we want in our application.

For a deeper dive into linting with ESLint, be sure to check out the chapter on **Tooling**.

React + TypeScript

Combining React and TypeScript can provide a robust environment for developing large-scale applications. In this section, we'll explore various patterns for how TypeScript can make writing React code more efficient, readable, and maintainable.

Typing Props in Function Components

When defining function components in React, it is often recommended to define the properties (props) the component receives using TypeScript interfaces or types.

Defining props in a React component

```
import React from 'react';

interface Props {
  name: string;
  age: number;
  address?: string;
}

const Person = ({ name, age, address }: Props) => {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      {address && <p>Address: {address}</p>}
    </div>
  );
};

export default Person;
```

In the above example, we've specified the `Person` component, which, when rendered, requires the `name` and `age` props to be defined and of type `string` and `number`, respectively. The component also accepts an optional `address` prop of type `string`.

When attempting to render the component with incorrect types, TypeScript will raise a compile-time error.

Rendering a component with incorrect types

```
import React from 'react';
import Person from './Person';

const App = () => {
  return (
    /*
      TypeScript Error:
      Type 'number' is not assignable to type 'string'.
      Type 'string' is not assignable to type 'number'.
    */
    <Person name={42} age={"twenty"} />
  );
```

```
};

export default App;
```

Typing Children

children is a special prop that is used to pass JSX elements from a parent component to a child component. By default, the `children` prop can be of any type. However, it can sometimes be useful to restrict the `children` prop to only accept certain types of elements.

`React.ReactNode` is a highly generic type for the `children` prop, accommodating any valid React child element, which includes primitive values like strings and numbers, React elements, and arrays or fragments containing these types.

Component that accepts children as React.ReactNode

```
import React from "react";

interface Props {
  children: React.ReactNode;
}

const Person = ({ children }: Props) => {
  return <div>{children}</div>;
};

export default Person;
```

Rendering a component with an array of elements as children

```
import React from "react";
import Person from "./Person";

const App = () => {
  return (
    <Person>
      {[
        <p key="1">Name: James</p>,
        <p key="2">Age: 30</p>,
      ]}
```

```
    </Person>
  );
};

export default App;
```

`React.ReactElement` is a more specific type for the `children` prop, representing an object of a React element that is typically created via JSX. It is useful when we want to restrict the `children` to single React elements rather than text or arrays.

Component that accepts children as `React.ReactElement`

```
import React from 'react';

interface Props {
  children: React.ReactElement;
}

const Person = ({ children }: Props) => {
  return <div>{children}</div>;
};

export default Person;
```

Rendering component with a single element as children

```
import React from 'react';
import Person from './Person';

const App = () => {
  return (
    <Person>
      <p>Name: James</p>
    </Person>
  );
};

export default App;
```

We can even be more specific on the shape of the children a component can accept by providing different types for each child:

```
interface Props {
  children: [
    React.ReactNode,
    number,
    React.ReactElement
  ];
}
```

React Hooks

React Hooks are functions that let us use state and other React features in functional components. When using the core Hooks from the React library, there are certain patterns and practices that we can learn to write type-safe React code.

useState

The `useState` Hook is the fundamental Hook used to manage the local state in a functional component. TypeScript can infer the type of the state from the initial value passed to `useState`, but we can also explicitly define the type of the state.

In the following example, we explicitly define the type of `count` as `number`.

Explicitly defining the type of state

```
const [count, setCount] = useState<number>(0);
```

If we were to attempt to set `count` to a value that isn't a number, TypeScript would flag this as an error. This type-checking ensures that the state remains consistent with the expected data type, preventing bugs and enhancing code quality.

Updating the number `count` property to a string

```
import React, { useState } from 'react';

const Counter: React.FC = () => {
  // Explicitly defining type of count as number
  const [count, setCount] = useState<number>(0);
```

```
  const updateCount = () => {
    /*
      TypeScript Error:
      Argument of type 'string' is not
      assignable to parameter of type
      'number | ((prevState: number) => number)'.
    */
    setCount("5");
  };

  return (
    <div>
      <h2>Counter: {count}</h2>
      <button onClick={updateCount}>
        Increment
      </button>
    </div>
  );
};

export default Counter;
```

useReducer

The `useReducer` Hook is used for managing more complex state logic in a functional component. It is similar to `useState` but is oftentimes more suited when handling complex state transitions, implementing optimizations, and managing state that requires a more structured approach.

We discuss the benefits of using the `useReducer` Hook in more detail in the chapter titled **State Management**.

With TypeScript, we can define types for the *state* and *action* of the `reducer` function and then use them in the `useReducer` Hook.

Defining types for state and action

```
import { useReducer } from 'react';

type State = { count: number };
type Action = {
  type: 'increment' | 'decrement'
```

```
};

const initialState: State = { count: 0 };

function reducer(
  state: State,
  action: Action
): State {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(
    reducer,
    initialState
  );

  // ...
};
```

If we were to use the `dispatch` function incorrectly with the `useReducer` Hook in the above TypeScript environment, TypeScript would flag an error. This is useful in ensuring that actions dispatched are consistent with the defined action types.

For example, consider the defined `Action` type above, which only allows `'increment'` or `'decrement'` as valid action types. If we attempt to dispatch an action that doesn't match this type, TypeScript will produce an error.

Dispatching an action from a component with an invalid type

```
// ...

const Counter = () => {
  const [state, dispatch] =
    useReducer(reducer, initialState);
```

```
  const wrongUpdate = () => {
    /*
      TypeScript Error:
      Argument of type '{ type: 'multiply' }'
      is not assignable  to parameter of type
      'Action'. Object literal may only specify
      known properties, and 'type' does not
      exist in type 'Action'.
    */
    dispatch({ type: 'multiply' });
  };

  // ...
};
```

This type-checking ensures that the `reducer` function only receives and processes the actions it is designed to handle, thereby maintaining the integrity and predictability of the state management.

useContext

The `useContext` Hook is used to create global state that can be accessed anywhere in the application. With TypeScript, we're able to define the type of the context state we create. Here's an example of using `useContext` with TypeScript to handle a theme context within a React application:

Creating a Themable React Component with Context API

```
import React, {
  createContext,
  useContext
} from 'react';

type Theme = 'light' | 'dark';
const ThemeContext =
  createContext < Theme > "light";

const ThemedComponent = () => {
  const theme = useContext(ThemeContext);

  const style = {
```

```
    background:
      theme === "light" ? "#fff" : "#333",
  };

  return <div style={style}>...</div>;
};

export default ThemedComponent;
```

Above, a `ThemeContext` is defined to accept only two possible theme values: 'light' or 'dark'. The `ThemedComponent` utilizes this context to dynamically set its background style based on the current theme.

Having the `ThemeContext` accept only two possible theme values ensures that any consumer of this context will have to adhere to these specified theme types.

TypeScript can also be applied in a similar manner with other core React Hooks such as `useEffect`, `useMemo`, and `useCallback` to guarantee the correct typing of their inputs and outputs. Beyond the core Hooks, TypeScript is valuable for defining types for custom Hooks and ensuring their correct usage across various components and contexts.

Custom Hooks

Custom Hooks are functions that we can create to extract and reuse stateful logic across multiple components. When we create a custom Hook, we can also define the types of its input and output.

Here's an example of a custom `useToggle` Hook, that allows components to manage a boolean state easily. The Hook takes an initial state as input and returns a tuple with the current state and a function to toggle it.

A custom `useToggle` Hook written in JavaScript

```
import { useState, useCallback } from 'react';

const useToggle = (initialState = false) => {
  const [isOn, setIsOn] = useState(initialState);

  const toggle = useCallback(() => {
```

```
    setIsOn(prevIsOn => !prevIsOn);
  }, []);

  return [isOn, toggle];
};

export default useToggle;
```

With TypeScript, we're able to explicitly define the types for the input and the output of the Hook. This ensures that the Hook is used correctly and allows for better type inference in the components that use it.

Here's the same `useToggle` Hook, written in TypeScript, that explicitly returns a tuple (an array with a fixed number of elements of known types).

The same `useToggle` Hook written in TypeScript

```
import { useState, useCallback } from 'react';

type UseToggle = {
  isOn: boolean;
  toggle: () => void;
};

const useToggle = (
  initialState: boolean
): [boolean, () => void] => {
  const [isOn, setIsOn] = useState(initialState);

  const toggle = useCallback(() => {
    setIsOn(prevIsOn => !prevIsOn);
  }, []);

  return [isOn, toggle] as const;
};

export default useToggle;
```

When the Hook is used in components, TypeScript ensures that the initial state provided to the Hook and the types of values it returns are used correctly.

Incorrect usage of the `useToggle` Hook

```
import React from 'react';
import useToggle from './useToggle';

const Component: React.FC = () => {
  /*
    TypeScript Error:
    Argument of type 'string' is not
    assignable to parameter of type 'boolean'.
  */
  const [isOn, setToggle] = useToggle("wrong type");
  return (
    // ...
  );
};

export default Component;
```

Correct usage of the `useToggle` Hook

```
import React from 'react';
import useToggle from './useToggle';

const Component: React.FC = () => {
  /*
    isOn: boolean
    toggle: () => void
  */
  const [isOn, toggle] = useToggle(false);

  return (
    // ...
  );
};

export default Component;
```

Enums

Enums in TypeScript are a way of organizing a collection of related values and can be very helpful in improving readability and maintainability in large React applications. They allow the grouping of

values that belong to a similar category making the code more intuitive and less prone to errors that can occur with the use of disparate, unstructured values. This facilitates easier refactoring and updating of values, as changes only need to be made in one centralized location.

Enums can group together numeric or string values.

Numeric enum

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}

// Usage
// move will have the value of 1
const move = Direction.Up;
```

String enum

```
enum Status {
  IDLE = 'IDLE',
  LOADING = 'LOADING',
  SUCCESS = 'SUCCESS',
  ERROR = 'ERROR'
}

// Usage
// currentStatus will be 'LOADING'
const currentStatus = Status.LOADING;
```

In modern TypeScript, we're also able to use the `as const` assertion instead of the `enum` keyword to create an enum.

Const enum with `as const` assertion:

```
const status = {
  IDLE: 'IDLE',
  LOADING: 'LOADING',
  SUCCESS: 'SUCCESS',
  ERROR: 'ERROR',
} as const;
```

```
// Usage
// currentStatus will be 'LOADING'
const currentStatus = Status.LOADING;
```

In the above example, using `as const` makes each property of the `status` object a literal type. This helps create a read-only object with fixed values, similar to an enum and is particularly useful when we want to use the actual values directly and ensure they are not changed.

Enums can also be helpful in rendering different UI components or elements based on certain states or conditions in React applications. For instance, in the provided `LoadingIndicator` component, the `status` const enum is used to manage various UI states (loading, success, and error) in a concise and type-safe manner.

Conditionally rendering UI with an enum

```
import React from 'react';

const status = {
  IDLE: 'IDLE',
  LOADING: 'LOADING',
  SUCCESS: 'SUCCESS',
  ERROR: 'ERROR',
} as const;

type Status = typeof status[keyof typeof status];

interface Props {
  status: Status;
}

const LoadingIndicator: React.FC<Props> = ({
  status,
}) => {
  switch (status) {
    case status.IDLE:
      return null;
    case status.LOADING:
      return <div>Loading...</div>;
    case status.SUCCESS:
      return <div>Data loaded successfully</div>;
    case status.ERROR:
```

```
      return <div>Error loading data</div>;
    default:
      return null;
  }
};

export default LoadingIndicator;
```

Generic Components

Generics, in TypeScript, are a feature that allows us to create reusable functions that can work over a variety of types rather than a single one. Generics can also be leveraged for React components to have them receive data (i.e., props) of different shapes. To best understand Generics, we'll go through an example of attempting to use a single component to handle accepting multiple data types.

Imagine a scenario where we have a `DataTable` component designed to show data within a table view. To start, we can have the `DataTable` be used to render user information within a table as follows:

A `DataTable` component

```
import React from 'react';

type UserData = {
  id: number;
  name: string;
  email: string;
};

interface Props {
  data: Array<UserData>;
  columns: Array<keyof UserData>;
}

const DataTable = ({
  data,
  columns
}: Props) => {
  return (
    <table>
      <thead>
```

```
          <tr>
            {columns.map((col, index) => (
              <th key={index}>{col}</th>
            ))}
          </tr>
        </thead>
        <tbody>
          {data.map(
            (item: UserData, index: number) => (
            <tr key={index}>
              {columns.map((col, idx) => (
                <td key={idx}>{item[col]}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    );
}
```

Rendering `DataTable` with user information

```
<DataTable
  data={[
    {
      id: 1,
      name: "John Doe",
      email: "john.doe@example.com",
    },
  ]}
  columns={["id", "name", "email"]}
/>
```

The `DataTable` component above takes a prop value named `data` of type `Array<UserData>`, where `UserData` represents an object containing user information. It also expects a value for the prop labeled `columns`, which is to be an array that lists which keys (or columns) of the `UserData` objects we want to display. Using `keyof UserData` ensures that only the keys from `UserData` can be used.

What if we wanted to reuse the same `DataTable` component for different shapes of data? For example, assume we wanted to render a data table of product information. One approach we could take to achieve

this is to use a union type to represent the multiple possible data shapes that our `DataTable` component can accept.

Adjusting `DataTable` to accept different shapes of data with union types

```
import React from 'react';

type UserData = {
  id: number;
  name: string;
  email: string;
};

type ProductData = {
  productId: number;
  productName: string;
  price: number;
};

interface Props {
  data: Array<UserData> | Array<ProductData>;
  columns:
    Array<keyof UserData> |
    Array<keyof ProductData>;
};

const DataTable = ({ data, columns }: Props) => {
  // implementation...
};
```

By employing union types, we've created flexibility in our `DataTable` component by allowing it to cater to different data shapes. However, this wouldn't scale well if we needed to continue to add more data types because as more data shapes are added or existing ones change, the maintenance cost escalates.

Another approach to having our `DataTable` component be adaptable to a large number of data shapes is to simply use the any type to state that our component can accept data of *any shape*.

Adjusting `DataTable` to accept data of any shape

```
import React from 'react';

interface Props {
  data: any;
  columns: string[];
}

const DataTable = ({ data, columns }: Props) => {
  // implementation...
}
```

While using the `any` type might seem like an easy solution to make the component adaptable, it comes at the cost of losing the benefits TypeScript offers. By using `any`, we're opting out of type safety checks, which can potentially introduce runtime errors.

This is where TypeScript Generics come in. They are specifically designed for these scenarios by providing us the flexibility of handling multiple data shapes **while preserving type safety**.

We can design the `DataTable` component as a generic one, accepting a `data` prop of type `Array<T>`, where `T` can be any object.

Having `DataTable` accept a type variable

```
import React from 'react';

const DataTable = <T extends object>(
  { data, columns }: Props<T>
) => {
  return (
    // ...
  );
};
```

In the above code snippet, the `<T extends object>` syntax declares a generic type `T`, which extends an object. This means `T` can be any object type.

The generic type variable, `T`, is passed to the `Props` interface, which is also generic, meaning it accepts a type variable. `T` is then used to specify

the types for the `data` and `columns` arrays: `data` as an array of any type `T`, and `columns` as an array of `T`'s keys.

Passing the type variable to the `Props` generic interface

```
import React from 'react';

interface Props<T> {
  data: Array<T>;
  columns: Array<keyof T>;
}

const DataTable = <T extends object>(
  { data, columns }: Props<T>
) => {
  return (
    // ...
  );
};
```

In its entirety, the `DataTable` component will now look like the following:

The generic `DataTable` component

```
import React from 'react';

interface Props<T> {
  data: Array<T>;
  columns: Array<keyof T>;
}

const DataTable = <T extends object>(
  { data, columns }: Props<T>
) => {
  return (
    <table>
      <thead>
        <tr>
          {columns.map((col, index) => (
            <th key={index}>{col}</th>
          ))}
        </tr>
      </thead>
```

```
      <tbody>
        {data.map((item: T, index: number) => (
          <tr key={index}>
            {columns.map((col, idx) => (
              <td key={idx}>
                {item[col as keyof T]}
              </td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  );
};
```

With these changes, a parent component can use the `DataTable` component to render a table of user information.

Rendering `DataTable` with user data

```
import React from "react";
import DataTable from "./DataTable";

const App = () => {
  const userData = [
    { id: 1, name: "John", age: 30 },
    { id: 2, name: "Jane", age: 25 },
  ];

  return <DataTable data={userData} />;
};

export default App;
```

Alternatively, we could have the `DataTable` component render a series of product information elsewhere in our app.

Rendering `DataTable` with product data

```
import React from "react";
import DataTable from "./DataTable";

const App = () => {
  const productData = [
```

```
    {
      productId: "p1",
      productName: "Laptop",
      price: 1000,
    },
    {
      productId: "p2",
      productName: "Phone",
      price: 500,
    },
  ];

  return <DataTable data={productData} />;
};

export default App;
```

In both examples above, TypeScript will be able to infer what the shape of data is being passed in without us having to specify any types explicitly. This is a powerful feature of TypeScript as it allows the compiler to check that the data we are passing to the `DataTable` component has a consistent shape without us having to specify types for our data explicitly.

Since the `DataTable` also accepts a generic parameter, we can opt out of type inference and explicitly define the type of the `data` prop if we want to.

Explicitly defining the shape of data

```
import React from 'react';
import DataTable from './DataTable';

type UserData = {
  id: number;
  name: string;
  email: string;
};

const App = () => {
  const userData = [
    { id: 1, name: 'John', age: 30 },
    { id: 2, name: 'Jane', age: 25 }
  ];
```

```
  return (
    <DataTable<UserData> data={userData} />
  );
};

export default App;
```

The above example is a slightly more rigid approach, as we've explicitly specified a type to describe the shape of data used in the `DataTable` component. This can sometimes be useful when we want to enforce a strict contract for the component's usage, ensuring that consumers of the component are aware of the exact data shape expected.

For additional reading on useful React and TypeScript component patterns, the React TypeScript Cheatsheets is a great open-source resource that offers some additional examples and tips.

Declaration Files

To harness the full potential of TypeScript, the third-party libraries we use should also be equipped with dynamic types. Unfortunately, many third-party libraries we might want to use are oftentimes only written in native JavaScript (e.g., lodash) or with an extension of JavaScript. This is where TypeScript declaration files come in.

Declaration files are files that declare the types of modules or libraries that are not written in TypeScript. These files have the `.d.ts` extension and contain type information about a library's functions, classes, and variables but no implementation code.

When importing and using a library that does not have types associated with it, we can either create custom declaration files or import the appropriate declaration files from the DefinitelyTyped repository.

DefinitelyTyped

DefinitelyTyped is a repository containing high-quality TypeScript declaration files for a wide range of popular JavaScript libraries. The

repository is a community-driven initiative, and anyone can contribute by adding or improving the type definitions.

When using an external library that does not include its own type definitions, DefinitelyTyped should be our first port of call. For instance, if we're using the `lodash` library, we can install its type definitions by executing:

Installing `@types/lodash` as a development dependency

```
yarn add @types/lodash -D
```

This will install the lodash type definitions, and TypeScript will automatically recognize and utilize them in our project.

Custom Declaration Files

If DefinitelyTyped does not host type definitions for a particular library, then it would be necessary for us to create custom declaration files.

Creating a custom declaration file involves generating a new `.d.ts` file and manually writing the type definitions for the library. Here's a basic example of a custom declaration file for a hypothetical library:

Example of a custom declaration file

```
// index.d.ts
declare module 'hypothetical-library' {
  export function add(
    a: number,
    b: number
  ): number;

  export function subtract(
    a: number,
    b: number
  ): number;
}
```

In this example, we declare a module `hypothetical-library` with two functions, `add` and `subtract`, each taking two numbers as parameters and returning a number.

Once the custom declaration file is created, we might need to adjust our TypeScript configuration file to ensure TypeScript includes it during compilation. This can be done by adding the path of the declaration file or its folder to the include option in the `tsconfig.json` file.

Configuring TypeScript to Include Declaration Files

```
{
  // ...
  "include": [
    "./path/to/your/declarations/*.d.ts",
    // other paths
  ],
  // ...
}
```

Auto-generating Types from an API

In large-scale React applications, we often find ourselves having our React app interact with an API to fetch data or perform actions. These APIs can be RESTful, GraphQL, gRPC-based, or even something else.

For TypeScript to provide meaningful value when interacting with these APIs, we should have TypeScript types that mirror the structure of the data that the API returns or accepts. Let's consider a scenario where we have a React application that interacts with a RESTful API. The API has an endpoint that returns a user object with specific attributes when an `id` is provided. First, we can create a TypeScript interface for a user, reflecting the structure of data expected from the API:

`User` interface

```
interface User {
  id: number;
  name: string;
};
```

In our React component, we can retrieve user data from the API by ID. While fetching this data, we'll assign it the type `User` to ensure the retrieved data matches this type.

Fetching user

```
import React, {
  useState,
  useEffect
} from 'react';
import { User } from './types';

const UserDetail: React.FC<{ userId: number }> =
  ({ userId }) => {

  // setting user to be of type "User"
  const [user, setUser] =
    useState<User | null>(null);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        const response = await fetch(
          `https://example.com/api/users/${userId}`
        );
        const userData = await response.json();

        /*
           cast the userData from the API
           to the "User" type
        */
        setUser(userData as User);
      }
    };

    fetchUser();
  }, [userId]);

  // Render the user details...
};
```

Manually creating the `User` interface as we've done above works well in scenarios where our React client app interacts with a few endpoints from the server. However, for large APIs with many endpoints, complex data structures, and changing schemas, creating these types from scratch will be time-consuming and error-prone. Manual maintenance of these types can also lead to discrepancies between the API and the application, resulting in runtime errors and increased debugging time. This is where auto-generating TypeScript types from an API becomes extremely valuable.

By auto-generating types, we can ensure that our application's types *are always in sync* with the API. This reduces manual work and decreases the likelihood of bugs caused by incorrect or outdated types. Fortunately, there are several tools available that can help auto-generate TypeScript types from an API.

GraphQL

When working with GraphQL, the schema, which is the structure of the data, is already well-defined. This is because GraphQL is already a strongly typed language, and the schema provides a contract that describes the shape of the data, the types involved, and the relationships between them.

To auto-generate TypeScript types from a GraphQL API, one popular tool is GraphQL Code Generator. Consider a very simple GraphQL schema like the one below:

An example GraphQL Schema

```
type User {
  id: ID!
  name: String!
}

type Query {
  getUser(id: ID!): User
}

type Mutation {
  createUser(name: String!): User!
}
```

After setting up and running the GraphQL Code Generator command, the tool will generate TypeScript types for the GraphQL objects (`User`), queries (`getUser`), and mutations (`createUser`) for the schema above.

Auto-generated TypeScript types from a GraphQL Schema

```
export type Scalars = {
  ID: { input: string; output: string; }
```

```
  String: { input: string; output: string; }
};

export type User = {
  __typename?: 'User';
  id: Scalars['ID'];
  name: Scalars['String'];
};

export type Query = {
  __typename?: 'Query';
  getUser?: Maybe<User>;
};

export type Mutation = {
  __typename?: 'Mutation';
  createUser?: Maybe<User>;
};

export type QueryGetUserArgs = {
  id: Scalars['ID'];
};

export type MutationCreateUserArgs = {
  name: Scalars['String'];
};
```

We can then import and use these auto-generated types in our TypeScript code to ensure type safety when interacting with the GraphQL API.

REST

For RESTful APIs, the process can be a bit more challenging because REST does not have a strongly typed schema like GraphQL. However, if the API is described using a specification like OpenAPI, we can use that to generate TypeScript types.

OpenAPI is a standard way to describe RESTful APIs using a JSON or YAML format. This description includes the endpoints, request and response types, and other relevant information about the API.

An example OpenAPI 3.1.0 specification written in YAML

```yaml
openapi: 3.1.0
info:
  title: User API
  version: 1.0.0
paths:
  /user:
    get:
      operationId: getUser
      parameters:
        - name: id
          in: query
          schema:
            type: string
      responses:
        '200':
          description: Successful response
    post:
      operationId: createUser
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                name:
                  type: string
      responses:
        '201':
          description: User created
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
```

One popular tool to generate TypeScript types from an OpenAPI specification is openapi-generator. This tool can generate client libraries,

server stubs, API documentation, and more for various programming languages, including TypeScript.

Auto-generated TypeScript types from a RESTful API

```
export interface User {
  id: string;
  name: string;
}

export interface GetUserParameters {
  id: string;
}

export interface CreateUserBody {
  name: string;
}
```

These types can then be used in our TypeScript code to ensure type safety when making API requests and handling responses.

gRPC

gRPC is a high-performance, open-source, and universal remote procedure call (RPC) framework initially developed by Google. It leverages HTTP/2 for transport and Protocol Buffers (protobufs) as the interface description language. Protobufs are a language-agnostic binary serialization format developed by Google.

gRPC Web is a JavaScript implementation of gRPC for browser clients and was developed internally at Google as part of the front-end stacks for Google's Web applications and cloud services. As a result, React applications can leverage gRPC Web to communicate with gRPC services directly from the browser, enabling full-duplex communication between the server and the client.

Consider a simple gRPC service defined using Protocol Buffers like the following:

An example gRPC service definition

```
syntax = "proto3";

package user;

service UserService {
  rpc GetUser (GetUserRequest) returns (User);
  rpc CreateUser (CreateUserRequest) returns (User);
}

message User {
  string id = 1;
  string name = 2;
}

message GetUserRequest {
  string id = 1;
}

message CreateUserRequest {
  string name = 1;
}
```

There are several tools available to generate TypeScript types from protobufs. One such tool is ts-proto—an idiomatic protobuf generator for TypeScript. After running the `ts-proto` tool on a `.proto` file, it will generate TypeScript types and client and server code for our gRPC service. For example, the generated code for the above `.proto` file could look like this:

Auto-generated TypeScript types from a gRPC service definition

```
// ...

export interface User {
  id: string;
  name: string;
}

export interface GetUserRequest {
  id: string;
}
```

```
export interface CreateUserRequest {
  name: string;
}

// ...

/*
  Similar encode and decode functions
  for GetUserRequest and CreateUserRequest
*/
```

We can then import and use these auto-generated types and methods in our TypeScript code to interact with the gRPC service.

Auto-generating TypeScript types from an API, whether it's RESTful, GraphQL, or gRPC-based, is a valuable and important practice that can save time, reduce bugs, and help maintain a robust and type-safe codebase.

Migrating an existing React app to TypeScript

Depending on the size of an application, migrating an existing React JavaScript app to TypeScript can be an overwhelming task. It involves understanding and resolving type errors that TypeScript will identify, which may not have been apparent in the original JavaScript code. Additionally, large applications often have complex states and props structures, making it challenging to accurately type all components and functions. Complexity increases if the app integrates with external libraries or APIs that might not have TypeScript support. All these aspects can make the migration process time-consuming and technically demanding.

With that said, some steps can be taken to simplify the migration of a large React JavaScript app to TypeScript, starting with establishing a migration plan before any work begins.

Plan your migration

Before you start migrating, **have a plan in place**. Decide which parts of your app you will migrate first. For example, it might be easier to start with smaller, less complex components before moving on to more complex and interconnected ones. Additionally, communicate with your team and ensure everyone is aware of the migration plan, as it may affect the work of others.

Migrate incrementally

Migrating incrementally is key to managing the complexity of the transition. **You don't have to migrate your entire application at once**. Instead, you can migrate one component or module at a time. This approach helps in keeping the application functional throughout the migration process.

Leverage JSDoc

JSDoc is a markup language used to annotate JavaScript code. When adopting TypeScript into an existing React app, one approach is to gradually introduce it into your codebase using JSDoc. This involves adding JSDoc comments to your existing JavaScript code to provide type annotations. Over time, you can convert your code to TypeScript as needed, using the JSDoc comments as a starting point for defining types.

Refactor code if necessary

Sometimes, your existing JavaScript code might not be structured in a way that is easy to migrate to TypeScript. In such cases, it might be helpful to refactor your code where appropriate during the migration. For example, you might need to break down large functions into smaller ones or convert callback-based code to use Promises or async/await.

Use IDEs that have good TypeScript support

Utilize IDEs that have good support for TypeScript, such as Visual Studio Code. These tools can provide useful features like

autocompletion, type checking, and error highlighting, which can be incredibly helpful during the migration process.

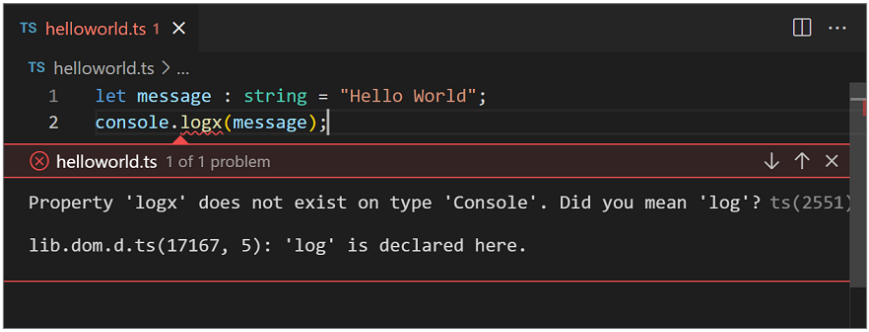


Figure 15-1. VSCode displaying a TypeScript error

Configure TypeScript compiler options in a gradual manner

Start with a loose TypeScript configuration by setting the `strict` option to `false` in your `tsconfig.json` file or by individually turning off the majority of strict checks. This will make the compiler more forgiving, which can be helpful when you are just starting the migration. As you progress and get more comfortable with TypeScript, you can gradually enable more strict compiler options.

Install DefinitelyTyped packages

Many popular JavaScript libraries have TypeScript type definitions available in the DefinitelyTyped repository. Before you start migrating, check if the libraries you are using have type definitions available, and install the corresponding `@types` packages.

Use `any` as a last resort

While migrating, you might be tempted to use the `any` type to bypass TypeScript's type checking. While this can be helpful to get your code to compile in the short term, it defeats the purpose of using TypeScript, which is to have a robust type system that catches errors at compile time. Try to define proper types for your variables, even if it takes a bit more time, and default to using `any` only as a last resort.

Leverage TypeScript's inference

TypeScript is very good at inferring types. In many cases, you don't need to explicitly define the types of your variables, as TypeScript can infer them from the context. This can save you a lot of time and effort during the migration.

Update your build and test tools

Make sure that your build and test tools are configured to handle TypeScript. You might need to install additional plugins or configure your existing tools to work with TypeScript.

Update Your CI (Continuous Integration) pipeline

Continuous Integration (CI) is the practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration is often verified by an automated build and automated tests.

When migrating to TypeScript, it is essential to update your CI pipeline to include the TypeScript compilation step. This ensures that any TypeScript errors are caught early in the development process and not just on a developer's local machine. Ideally, the build pipeline needs to not file TypeScript errors/warnings for existing JavaScript files that have yet to be migrated to TypeScript while you may still be in the migration process.

The above are only some practices to keep in mind when migrating an existing React/JavaScript application over to using TypeScript. Depending on the application you work on, you may encounter different challenges that are not covered in the above. Remember, migrating a large React application to TypeScript is a process, and it's okay to take it one step at a time.

Additional reading

- A great piece on how Stripe migrated millions of lines of code from Flow (a less commonly used JavaScript static type

checker) to TypeScript for their largest JavaScript codebase (Stripe Dashboard): Migrating millions of lines of code to TypeScript.

- A useful cheat sheet on migrating to TypeScript: Migrating (to TypeScript) Cheatsheet.

Conclusions

As we draw this book to a close, let's take a moment to look back at what we've covered in the ever-changing world of building large JavaScript React applications. Web development is always moving, and building apps that are big, easy to keep up, and run smoothly isn't a walk in the park and can require a lot of thinking and planning.



Throughout the book, we've dived into a whole bunch of different concepts, each super important for building large-scale applications. Amongst other things, we've discussed concepts such as fetching data, managing an app's client-side state, making our code work worldwide, and the importance of testing and using the right tools.

Oftentimes, when embarking on the journey of building a large-scale React application, there can be a lot to consider. Let's walk through some points to consider:

Getting started

Think about what your app aims to achieve, and also give some thought to the initial aspects of how you might approach writing your first lines of code:

- Who are the intended users of the app going to be?
- What are the app's must-have features?
- What problems is the app solving for these users?
- How will the user interface and experience cater to your target audience?
- What is the scope of the initial release vs. long-term vision for the app?

Answering these questions is more than just preliminary planning; **it can set the foundation for everything that follows**. This early stage of conceptualization is where you lay the groundwork for your application's design, functionality, and user experience.

Organizing code

When it comes to organizing your code, think about a structure that not only makes your code easy to read, maintain, and scale but also fits well with your team's style. This might mean breaking down your app into modular components, setting up a clear and logical directory layout, and sticking to a set of naming rules everyone agrees on.

Equally important is establishing and following a coding style guide—a guide that outlines best practices, coding conventions, and patterns that your team agrees upon.

Type-safety

Would you and your team be up for using TypeScript in all development code? Using TypeScript does mean some extra setup, but think about the long-term advantages of having type-safe code.

Fetching data

Think about how your React app will retrieve its data. Are you connecting to one or several APIs? What kind of APIs are these—RESTful, GraphQL, or something else?

The complexity of your data-fetching utilities will hinge on the factors, so setting up a solid foundation here is vital. This means choosing the right libraries and patterns from the get-go, considering aspects like caching strategies, error handling, and state management related to data fetching.

Managing client-side state

How will you handle your app's client-side state? Can you depend on your data-fetching library's caching, or do you need something more powerful like Redux?

Additionally, consider the power of native React Hooks like useReducer and useContext. These Hooks can offer simpler yet effective ways to manage state, especially in scenarios where you might not need the full-scale solution that something like Redux provides.

Design systems and component libraries

When working closely with your product and design teams, it's crucial to align on the visual and functional consistency of your application. Do you have an internal component library or design system? If not, consider exploring third-party libraries or CSS frameworks, as they could provide a valuable head start in streamlining your development (and design) process.

Internationalization/Accessibility

Who's your audience? If your app is global, should you be thinking about internationalization right from the start?

Performance

How will you measure and keep track of your app's performance? Are you using tools like Lighthouse, and how will you monitor these metrics over time?

Testing

What's your game plan for testing? Are you focusing more on integration tests, with a mix of unit and end-to-end tests? What testing tools will you use for this?

Understanding user behavior

How will you get insights into user behavior and preferences? Are you planning to track user settings and perform personalization or A/B tests?

The above can be key in figuring out which features your users love and which ones need more work.

Serving the app

Finally, how will you serve your code to users? What practices does your team follow for merging and reviewing code? It can be crucial to nail down your deployment strategy early, as this can make a big difference in the smooth running of your development process.

Remember, all the above are just starting points. The real magic happens when you mix these with your unique challenges and team dynamics. There's no one-size-fits-all solution in React web development, especially when it comes to working on big projects. Keep these questions in mind, stay open to new ideas, and don't be afraid to pivot your strategies as you learn more.

Additional reading

- Designing very large (JavaScript) applications — Industrial Empathy & Designing Even Larger (JavaScript) Applications — Malte Ubl

- Examples of Large Production-Grade Open Source React Apps — Max Rozen
- Twitter Lite and Progressive Web Apps At Scale — Paul Armstrong
- Lessons learned: how I'd rebuild a social web app like Twitter today — Paul Armstrong
- Organizing large React applications — Jack Franklin

And that's a wrap! We hope this book lights up your path to building amazing, large-scale React applications. We'll be adding new content to the book over time so if you're interested in following along with these updates, do ensure you've joined our Substack newsletter at https://largeapps.substack.com/!

Remember, the journey is always as important as the destination. Happy coding!

- Addy & Hassan

About the Authors



Addy Osmani is an engineering leader working on Google Chrome. He leads up Chrome's Developer Experience organization, helping reduce the friction for developers to build great user experiences.



Hassan Djirdeh is a software engineer and has helped build large production applications at scale at organizations like Doordash (where he’s currently employed), Instacart, and Shopify.