

O'REILLY®

Fluent React

Build Fast, Performant, and Intuitive
Web Applications



Early
Release

RAW &
UNEDITED

Tejas Kumar

Fluent React

Build Fast, Performant, and Intuitive Web Applications

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Tejas Kumar



Beijing • Boston • Farnham • Sebastopol • Tokyo

Fluent React

by Tejas Kumar

Copyright 2023 Tejas Kumar. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisition Editor: Amanda Quinn
- Development Editor: Shira Evans
- Production Editor: Beth Kelly
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- March 2024: First Edition

Revision History for the Early Release

- 2022-10-25: First Release
- 2022-12-12: Second Release
- 2023-02-21: Third Release
- 2023-04-05: Fourth Release
- 2023-06-27: Fifth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098138714> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Fluent React*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other

technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-13865-3

[FILL IN]

Chapter 1. The Entry-level Stuff

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Let’s start with a disclaimer: React was made to be used by all. In fact, you could go through life never having read this book and continue to use React without problems! This book dives much deeper into React for those of us that are curious about its internals, advanced patterns, and best practices. Let’s go on a deep dive together starting at the top: the higher-level, entry-level topics. We’ll start with the basics of React, and then dive deeper and deeper into the details of how React works.

In this chapter, we'll talk about why React exists, how it works, and what problems it solves. We'll cover its initial inspiration and design, and follow it from its humble beginnings at Facebook to the prevalent solution that it is today. This chapter is a bit of a meta chapter, but it's important to understand the context of React before we dive into the details. Let's get started!

Why Is React a Thing?

The answer in one word is: **updates**. In the early days of the web, we had a lot of static pages. We'd fill out forms, hit submit, and load an entirely new page. This was fine for a while, but then we started to want more. We wanted to be able to see things update instantly without having to wait for a new page to be rendered and loaded. We wanted the web and its pages to feel *snappier* and more "instant". The problem was that these instant updates were pretty hard to do *at scale* for a number of reasons:

1. **Performance:** imperatively making updates to web pages often caused performance bottlenecks because we were prone to perform work that triggered browsers to reflow and repaint the page.
2. **Reliability:** keeping track of state and making sure that the state was consistent was hard to do, because we had to keep track of state in multiple places and make sure that the state was consis-

tent across all of those places. This was especially hard to do when we had multiple people working on the same codebase.

3. **Security:** we had to be sure to sanitize all HTML and JavaScript that we were injecting into the page to prevent exploits like cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks carefully.

A World Without User Interface Libraries

These were some of the large problems for companies that were building web apps at the time. They had to figure out how to make their apps feel snappy and instant, but also scale to millions of users and work reliably in a safe way. For example, let's consider a button click: when a user clicks a button, we want to update the user interface to reflect that the button has been clicked. We'd need to consider at least 4 different "states" the user interface can be in:

1. Pre-click: the button is in its default state, and has not been clicked.
2. Clicked, but pending: the button has been clicked, but the action that the button is supposed to perform has not yet completed.
3. Clicked, and succeeded: the button has been clicked, and the action that the button is supposed to perform has completed.
4. Clicked, and failed: the button has been clicked, but the action that the button is supposed to perform has failed.

Once we have these states, we need to figure out how to update the user interface to reflect these states. Often times, updating the user interface would require the following steps:

1. Find the button in the host environment (often the browser) using some type of primitive like `document.querySelector` or `document.getElementById`
2. Attach event listeners to the button to listen for click events
3. Perform any state updates in response to events
4. When the button leaves the page, remove the event listeners and clean up any state

This is a simple example, but it's a good one to start with. Let's say we have a button that says "Like" and when a user clicks it, we want to update the button to say "Liked". How do we do this? To start with, we'd have an HTML element:

```
<button type="button">Like</button>
```

We'd need some way to reference this button with JavaScript, so we'd give it an `id` attribute:

```
<button type="button" id="like-button">Like</button>
```

Great! Now that there's an `id`, JavaScript can work with it to make it interactive. We can get a reference to the button using `document.getElementById`, and then we'll add an event listener to the button to listen for click events:

```
const likeButton = document.getElementById("like-  
likeButton.addEventListener("click", () => {  
  // do something  
});
```

Now that we have an event listener, we can do something when the button is clicked. Let's say we want to update the button to say "Liked" when it's clicked. We can do this by updating the button's text content:

```
const likeButton = document.getElementById("like-  
likeButton.addEventListener("click", () => {  
  likeButton.textContent = "Liked";  
});
```

Great! Now we have a button that says "Like" and when it's clicked, it says "Liked". The problem here is that we can't "unlike" things. Let's

fix that and update the button to say “Like” again. We’d need to add some state to the button to keep track of whether or not it’s been clicked. We can do this by adding a `data-liked` attribute to the button:

```
<button type="button" id="like-button" data-liked
```

Now that we have this attribute, we can use it to keep track of whether or not the button has been clicked. We can update the button’s text content based on the value of this attribute:

```
const likeButton = document.getElementById("like-  
likeButton.addEventListener("click", () => {  
  const liked = likeButton.getAttribute("data-li  
  likeButton.setAttribute("data-liked", !liked);  
  likeButton.textContent = liked ? "Like" : "Like  
});
```

Wait, but we’re just changing the `textContent` of the button! We’re not actually saving the liked state to a database. Normally to do this, we’d communicate over the network like so:

```
const likeButton = document.getElementById("like-  
likeButton.addEventListener("click", () => {  
  const liked = likeButton.getAttribute("data-li
```

```

// communicate over the network
fetch("/like", {
  method: "POST",
  body: JSON.stringify({ liked: !liked }),
}).then(() => {
  likeButton.setAttribute("data-liked", !liked);
  likeButton.textContent = liked ? "Like" : "Like";
});
});

```

Now we're communicating over the network, but what if the network request fails? We'd need to update the button's text content to reflect that the network request failed. We can do this by adding a `data-failed` attribute to the button:

```

<button type="button" id="like-button" data-liked="false" data-failed="false">
  Like
</button>

```

Now we can update the button's text content based on the value of this attribute:

```

const likeButton = document.getElementById("like-button");

```

```

likeButton.addEventListener("click", () => {
  const liked = likeButton.getAttribute("data-liked")

  // communicate over the network
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: !liked }),
  })
    .then(() => {
      likeButton.setAttribute("data-liked", !liked)
      likeButton.textContent = liked ? "Like" : "Dislike"
    })
    .catch(() => {
      likeButton.setAttribute("data-failed", true)
      likeButton.textContent = "Failed";
    });
});

```

There's one more case to handle: the process where we're currently "liking" a thing. That is, the pending state. To model this in code, we'd set yet another attribute on the button for pending state like so:

```

<button
  type="button"
  id="like-button"
  data-pending="false"
  data-liked="false"

```

```
    data-failed="false"
  >
    Like
  </button>
```

Now, we can disable the button if a network request is in process so that multiple clicks don't queue up network requests and lead to odd race conditions and server overload.

```
const likeButton = document.getElementById("like-
likeButton.addEventListener("click", () => {
  const liked = likeButton.getAttribute("data-li
  const isPending = likeButton.getAttribute("data

  if (isPending) {
    return; // do nothing
  }

  likeButton.setAttribute("data-pending", "true")
  likeButton.setAttribute("disabled", "disabled")

  // communicate over the network
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: !liked }),
  })
  .then(() => {
```

```
        likeButton.setAttribute("data-liked", !liked);
        likeButton.textContent = liked ? "Like" : "Dislike";
    })
    .catch(() => {
        likeButton.setAttribute("data-failed", "true");
        likeButton.textContent = "Failed";
    })
    .finally(() => {
        likeButton.setAttribute("data-pending", "false");
        likeButton.setAttribute("disabled", null);
    });
});
```

Okay, now our button is kind of robust and can handle multiple states—but a few questions still remain:

1. Is `data-pending` really necessary? Can't we just check if the button is disabled? Probably not because a disabled button could be disabled for other reasons, like the user not being logged in or otherwise not having permission to click the button.
2. Would it make more sense to have a `data-state` attribute instead of `data-liked` and `data-failed`? Probably, but then we'd need to add a bunch of logic to handle the different states.
3. How do we test this button in isolation? Can we?
4. Why do we have the button initially written in HTML, and then later work with it in JavaScript? Wouldn't it be better if we could just

create the button in JavaScript and then

`document.appendChild` it? This would make it easier to test and would make the code more self-contained, but then we'd have to keep track of its parent if its parent isn't `document`. In fact, we might have to keep track of *all* the parents on the page.

React helps us solve some of these problems but not all of them: for example, the question of how to break up state into separate flags (`isPending`, `hasFailed`, etc.) or a single state variable (like `state`) is a question that React doesn't answer for us. It's a question that we have to answer for ourselves. But React does help us solve the problem of scale: creating a lot of buttons that need to be interactive and updating the user interface in response to events in a minimal and efficient way, and doing this in a testible, reproducible, and reliable way.

This is a very simple example, but it's a good one to start with. So far, we've seen how we can use JavaScript to make a button interactive, but this is a very manual process if we want to do it *well*: we have to find the button in the browser, add an event listener, update the button's text content, and account for myriad edge cases. This is a lot of work, and it's not very scalable. What if we had a lot of buttons on the page? What if we had a lot of buttons that needed to be interactive? What if we had a lot of buttons that needed to be interactive, and we needed to update the user interface in response to events?

This pain of creating reliable and scalable user interfaces was shared by many web companies at the time. It was at this point on the web that we saw the rise of multiple JavaScript-based solutions that aimed to solve this: Backbone, KnockoutJS, and jQuery. Let's look at these solutions in turn and see how they solved this problem. This will help us understand how React is different from these solutions, and maybe even superior to them.

Backbone

Backbone was one of the first of these solutions, and it was a very simple solution: it was a library that provided a way to create “models” and “views”. Models were conceptually sources of data, and views were conceptually user interfaces that consumed and rendered that data. Backbone exported comfortable APIs to work with these models and views, and then it provided a way to connect the models and views together. Though this was a somewhat simple solution, it was a solution that was very powerful and flexible for its time. It was also a solution that was scalable to use and allowed developers to test their code in isolation.

Here's our button example from before, but this time using Backbone:

```
const LikeButton = Backbone.View.extend({  
  tagName: "button",  
  attributes: {
```

```

attributes: {
  type: "button",
},
events: {
  click: "onClick",
},
initialize() {
  this.model.on("change", this.render, this);
},
render() {
  this.$el.text(this.model.get("liked") ? "Like" : "Dislike");
  return this;
},
onClick() {
  fetch("/like", {
    method: "POST",
    body: JSON.stringify({ liked: !this.model.get("liked") })
  })
    .then(() => {
      this.model.set("liked", !this.model.get("liked"));
    })
    .catch(() => {
      this.model.set("failed", true);
    })
    .finally(() => {
      this.model.set("pending", false);
    });
},
});

```

```
//  
  
const likeButton = new LikeButton({  
  model: new Backbone.Model({  
    liked: false,  
  }),  
});  
  
document.body.appendChild(likeButton.render().el);
```

Notice how `LikeButton` is a subclass of `Backbone.View` and how it has a `render` method that returns `this`. We'd go on to see a similar `render` method in React, but let's not get ahead of ourselves. Backbone exposed a chainable API that allowed developers to colocate logic as properties on objects. Comparing this to our previous example, we can see that Backbone has made it far more comfortable to create a button that is interactive and that updates the user interface in response to events. It also does this in a more structured way by grouping logic together. Some might also note that Backbone has made it more approachable to test this button in isolation because we can create a `LikeButton` instance and then call its `render` method to test it.

We'd test this component like so:

```
test("LikeButton", () => {
```

```
const likeButton = new LikeButton({
  model: new Backbone.Model({
    liked: false,
  }),
});
expect(likeButton.render().el.textContent).toBe(
  'Like'
);
```

We can even test the button's behavior after its state changes, as in the case of a click event like so:

```
test("LikeButton", () => {
  const likeButton = new LikeButton({
    model: new Backbone.Model({
      liked: false,
    }),
  });
  expect(likeButton.render().el.textContent).toBe('Like');
  likeButton.onClick();
  expect(likeButton.render().el.textContent).toBe('Liked');
});
```

For this reason, Backbone was a very popular solution at the time. The alternative was to write a lot of imperative code that was hard to test and hard to reason about, with no guarantees that the code

would work as expected in a reliable way. Therefore, Backbone was a very welcome solution.

KnockoutJS

Let's compare this approach with another popular solution at the time: KnockoutJS. KnockoutJS was a library that provided a way to create “observables” and “bindings”. Observables were conceptually sources of data, and bindings were conceptually user interfaces that consumed and rendered that data: observables were like models, and bindings were like views. KnockoutJS exported APIs to work with these observables and bindings. Let's look at how we'd implement this button in KnockoutJS. This will help us understand “why React” a little better. Here's the KnockoutJS version of our button:

```
function createViewModel({ liked }) {  
  const isPending = ko.observable(false);  
  const hasFailed = ko.observable(false);  
  const onClick = () => {  
    isPending(true);  
    fetch("/like", {  
      method: "POST",  
      body: JSON.stringify({ liked: !liked() }),  
    })  
      .then(() => {  
        liked(!liked());  
      })  
      .catch(() => {  
        hasFailed(true);  
      })  
  };  
}
```

```
    })
    .catch(() => {
        hasFailed(true);
    })
    .finally(() => {
        isPending(false);
    });
};
return {
    isPending,
    hasFailed,
    onClick,
    liked,
};
}

ko.applyBindings(createViewModel({ liked: ko.observable(false)}), document.getElementById('app'))
```

A “view model” is a JavaScript object that contains keys and values that we bind to various elements in our page using the `data-bind` attribute. There are no “components” or “templates” in KnockoutJS, just a view model and a way to bind it to the browser.

Our function `createViewModel` is how we’d create a view model with Knockout. We then use `ko.applyBindings` to connect the view model to the host environment (the browser). The

`ko.applyBindings` function takes a view model and then finds all the elements in the browser that have a `data-bind` attribute, which Knockout uses to bind them to the view model.

A button in our browser would be bound to this view model's properties like so:

```
<button
  type="button"
  data-bind="click: onClick, text: liked ? 'Liked' : 'Like'
></button>
```

We *bind* the HTML element to the “view model” we created using our `createViewModel` function, and the site becomes interactive. As you can imagine, explicitly subscribing to changes in observables and then updating the user interface in response to these changes is a lot of work. KnockoutJS was a great library for its time, but it was also a library that required a lot of boilerplate code to get things done.

Moreover, view models often grew to be very large and complex, which led to increasing uncertainty around refactors and optimizations to code. Eventually, we'd end up with verbose monolithic view models that were hard to test and hard to reason about. Still, KnockoutJS was a very popular solution and it was a great library for its

time. It was also relatively easy to test in isolation, which was a big plus.

For posterity, here's how we'd test this button in KnockoutJS:

```
test("LikeButton", () => {
  const viewModel = createViewModel({ liked: ko.observable(false) });
  expect(viewModel.liked()).toBe(false);
  viewModel.onClick();
  expect(viewModel.liked()).toBe(true);
});
```

jQuery

Okay—we're slowly understanding why React was needed, introduced, and loved. Let's take one final detour into jQuery to get a full picture of the state of the art at the time and the value that React brought to the table. Here's our button example from before, but this time using jQuery:

```
<button type="button" id="like-button">Like</button>
```

```
$("#like-button").on("click", function () {
  this.prop("disabled", true);
  fetch("/like")
```

```

    method: "POST",
    body: JSON.stringify({ liked: this.text() ===
  })
  .then(() => {
    this.text(this.text() === "Like" ? "Liked"
  })
  .catch(() => {
    this.text("Failed");
  })
  .finally(() => {
    this.prop("disabled", false);
  });
});

```

From this example, we observe a common pattern that previous libraries for building user interfaces implemented: they bound data to the user interface and used this data binding to update the user interface in place. While Knockout and Backbone bound models to views in one way or another, jQuery was far more active in directly manipulating the user interface itself.

jQuery ran in a heavily “side-effectful” way, constantly interacting with and altering state outside of its own control. This was a pattern that was common at the time, and it was a pattern that was difficult to reason about and test because the world around the code was constant-

ly changing. At some point, we'd have to stop and ask ourselves: "what is the state of the browser right now?"—a question that became increasingly difficult to answer as our codebases grew.

This button with jQuery is hard to test because it's just an event handler. If we were to write a test, it'd look like this:

```
test("LikeButton", () => {  
  const $button = $("#like-button");  
  expect($button.text()).toBe("Like");  
  $button.trigger("click");  
  expect($button.text()).toBe("Liked");  
});
```

The only problem, is `$('#like-button')` will have returned `null` in the testing environment because it's not a real browser.

We'd have to mock out the browser environment to test this code, which is a lot of work. This is a common problem with jQuery: it's hard to test because it's hard to isolate and depends heavily on the browser environment.

The Birth of React

Facebook was no exception to this problem, and as a result created a number of internal solutions complementary to what already existed

at the time. Among the first of these was “BoltJS”: a tool Facebook engineers would say “bolted together” a bunch of things that each of them liked. It was a combination of tools that was assembled to make updates to Facebook’s web user interface more intuitively.

Around this time, Facebook engineer Jordan Walke had a radical idea that did away with the status quo of the time and instead entirely replaced minimal portions of web pages with new ones as updates happened. As we’ve seen previously, JavaScript libraries would manage relationships between views (user interfaces) and models (conceptually, sources of data) using a paradigm called “two-way data binding”. This was pretty complicated, and often proved difficult to keep the views and models in sync because of the way the web worked. Jordan’s idea was to instead use a paradigm called “one-way data flow”. This was a much simpler paradigm and it was much easier to keep the views and models in sync. This was the birth of the “Flux” architecture that would go on to be the foundation of React.

The Flux architecture was a simple idea: instead of binding views to models and then updating the views in place, we would instead destroy the old view and create a new one with the latest state in a minimal and efficient way—a process that would later be called “reconciliation”. This was a radical departure from the way we had been building web apps for years, and it was a departure that was met with skepticism. The fact that Facebook was a large company with a lot of

resources, a lot of users, and a lot of engineers with opinions made its upward climb a steep one. After much scrutiny, React was an internal success. It was adopted by Facebook, and then by Instagram.

It was then open sourced in 2013 and released to the world where it was met with tremendous amounts of backlash. People heavily criticized React for its use of JSX, accusing Facebook of “putting HTML in JavaScript” and breaking separation of concerns. Facebook became known as the company that “rethinks best practices” and breaks the web. Eventually, after slow and steady adoption by companies like Netflix, Airbnb, and The New York Times, React became the de facto standard for building user interfaces on the web.

There are a number of details that I’ve left out of this story because they fall out of the scope of this book, but I think it’s important to understand the context of React before we dive into the details: specifically the class of technical problems React was created to solve. Should you be more interested in the story of React, there is a full documentary on the history of React that is freely available on YouTube.

Now that we understand the motivation for React and the state of the world around its introduction, let’s explore a little bit how browsers work, and how React fits into this picture.

A More Complex Example

This book assumes we have a satisfactory understanding of this statement: browsers render web pages. Web pages are HTML documents that are styled by CSS and made interactive with JavaScript. This has worked great for decades and still does, but building modern web applications that are intended to service a significant (think millions) amount of users with these technologies requires a good amount of abstraction in order to do it safely and reliably with as little possibility for error as possible.

Let's consider another example that's a little bit more complex than our like button previously by exploring the problem with performing interactive application updates without React to understand this a little better. We'll start with a simple example: a list of items. Let's say we have a list of items and we want to add a new item to the list. We could do this with an HTML form that looks something like this.

```
<ul id="list-parent"></ul>

<form id="add-item-form" action="/api/add-item" r
  <input type="text" id="new-list-item-label" />
  <button type="submit">Add Item</button>
</form>
```

JavaScript gives us access to DOM APIs, where DOM stands for **D**ocument **O**bject **M**odel, and API stands for **A**pplication **P**rogramming **I**nterface. For the uninitiated, the DOM is an in-memory model of a web page's document structure: it's a tree of objects that represents the elements on your page, giving you ways to interact with them via JavaScript. The problem is, the DOM on some user of your web app's device is like an alien planet: we have no way of knowing what browser they're using, in what network conditions, and on what operating system (OS) they're working. The result? We have to write code that is resilient to all of these factors.

Moreover, application state becomes quite hard to predict when it updates without some type of state-reconciliation mechanism to keep track of things. To continue with our list example, let's consider some JavaScript code to add a new item to the list:

```
(function myApp() {  
  var listItems = ["I love", "React", "and", "TypeScript"];  
  var parentList = document.getElementById("list-items");  
  var addForm = document.getElementById("add-item-form");  
  var newListItemLabel = document.getElementById("new-item-label");  
  
  addForm.onsubmit = function (event) {  
    event.preventDefault();  
    listItems.push(newListItemLabel.value);  
    renderListItems();  
  };  
})
```

```
};

function renderListItems() {
  for (i = 0; i < listItems.length; i++) {
    var el = document.createElement("li");
    el.textContent = listItems[i];
    parentList.appendChild(el);
  }
}

renderListItems();
})();
```

This code snippet is written to look as similar as possible to early web applications. Why does this go haywire over time? It's mainly because building applications intended to scale this way over time presents some footguns making them:

1. **Unsafe:** `addForm`'s `onsubmit` attribute could be easily rewritten by other client-side JavaScript on the page. We could use `addEventListener` instead, but this presents more questions:
 - a. Where and when would we clean it up with `removeEventListener`?
 - b. Would we accumulate a lot of event listeners over time if we're not careful about this?

- c. What penalties will we pay because of it?
- 2. **Unpredictable:** Our sources of truth are mixed: we're holding list items in a JavaScript array, but relying on existing elements in the DOM (like an element with `id="list-parent"`) to complete our app. Because of these interdependencies between JavaScript and HTML, we have a few more things to consider:
 - a. What if there are mistakenly multiple elements with the same `id`?
 - b. What if the element doesn't exist at all?
 - c. What if it's not a `ul`? Can we append list items (`li` elements) to other parents?
 - d. What if we use class names instead?

Our sources of truth are mixed between JavaScript and HTML, thus making the truth unreliable. We'd benefit more from having a single source of truth. Moreover, elements are added and removed from the DOM by client-side JavaScript all the time. If we rely on the existence of these specific elements, our app has no guarantees of working reliably as the UI keeps updating. Our app in this case is full of "side-effects", where its success or failure depends on some userland concern. React has remedied this by advocating a functional programming-inspired model where side effects are intentionally marked and isolated.

- 3. **Inefficient:** `renderListItems` renders items on the screen sequentially. Each mutation of the DOM can be computationally

expensive, especially where layout shift and reflows are concerned. Since we're in an alien planet with unknown computational power, this can be quite unsafe for performance in case of large lists. Remember—we're intending our large-scale web application to be used by millions worldwide—including those with low-power devices from communities across the world without access to the latest and greatest Apple M2 Pro Max processors. What may be more ideal in this scenario, instead of sequentially updating the DOM per single list item, would be to batch these operations somehow and apply them all to the DOM at the same time. But maybe this isn't worth doing for us as engineers because perhaps browsers will eventually update the way they work with quick updates to the DOM and automatically batch things for us?

These are some of the problems that has plagued web developers like you and I for years before React and other abstractions appeared. Packaging code in a way that was maintainable, reusable, and predictable at scale was another problem without much standardized consensus in the industry at the time. Given that Facebook had a front-row seat to these problems at enormous scale, React pioneered a component-based approach to building user interfaces that would solve these problems and more, where each component would be a self-contained unit of code that could be reused and composed with other components to build more complex user interfaces.

React's Value Proposition

Okay, history lesson's over. Hopefully we now have enough context to begin to understand why React is a thing. Given how easy it was to fall into the pit of unsafe, unpredictable, and inefficient JavaScript code at scale, we needed a solution to steer us more towards a pit of success where we *accidentally win*. Let's talk about exactly how React does that.

Imperative vs. Declarative Updates

React provides a declarative abstraction on the DOM. We'll talk more about how it does this in more detail later in the book, but essentially it provides us a way to write code that expresses *what we want to see*, while then taking care of *how it happens*, ensuring our user interface is created and works in a safe, predictable, and efficient manner.

Let's consider the list app that we created above. In React, we could rewrite it like this:

```
function MyList() {  
  const [items, setItems] = useState(["I love"]);  
  
  return (  
    <div>  
      <ul>
```

```
        {items.map((i) => (  
            <li key={i /* keep items unique */}>{i  
        )))  
    </ul>  
    <NewItemForm />  
</div>  
);  
}
```

Notice how in the `return`, we literally write something that looks like HTML: it looks like what we want to see. I want to see a box with a `NewItemForm`, and a list. Boom. How does it get there? That's React's problem. Do we batch list items to add chunks of them at once? Do we add them sequentially, one-by-one? React deals with *how* this is done, while we merely describe *what* we want done. In further chapters, we'll dive into React and explore how exactly it does this.

Do we then depend on class names to reference HTML elements? Do we `getElementById` in JavaScript? Nope. React creates unique "React elements" for us under the hood that it uses to detect changes and make incremental updates so we don't need to read class names and other identifiers from user code whose existence we cannot guarantee: our source of truth becomes exclusively Java-

Script with React. We can then write code that is safe, predictable, and efficient.

We export our `MyList` component to React, and React gets it on the screen for us in a way that is safe, predictable, and performant—no questions asked. It does this by using a “virtual DOM”, which is a clone of the real DOM. It then compares the virtual DOM to the real DOM, and makes incremental updates to the real DOM to make it match the virtual DOM. This is how React is able to make updates to the DOM in a safe, predictable, and efficient manner.

The Virtual DOM

The virtual DOM is a programming concept that allows developers to update the UI without directly manipulating the actual DOM. React uses the virtual DOM to keep track of changes to a component and re-renders the component only when necessary. This approach is faster and more efficient than updating the entire DOM tree every time there is a change.

In React, the virtual DOM is a lightweight representation of the actual DOM tree. It is a plain JavaScript object that describes the structure and properties of the UI elements. React creates and updates the virtual DOM to match the actual DOM tree, and any changes made to

the virtual DOM are applied to the actual DOM using a process called reconciliation.

To understand how the virtual DOM works, let's take a simple example of a like button. We will create a React component that displays a like button and the number of likes. When the user clicks the button, the number of likes should increase by one.

Here is the code for our component:

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

function LikeButton() {
  const [likes, setLikes] = useState(0);

  function handleLike() {
    setLikes(likes + 1);
  }

  return (
    <div>
      <button onClick={handleLike}>Like</button>
      <p>{likes} Likes</p>
    </div>
  );
}

export default LikeButton;
```

In this code, we have used the `useState` hook to create a state variable `likes`, which holds the number of likes. We have also defined a function `handleLike` that increases the value of likes by one when the button is clicked. Finally, we render the like button and the number of likes using JSX.

Now, let's take a closer look at how the virtual DOM works in this example.

Initial Rendering

When the `LikeButton` component is first rendered, React creates a virtual DOM tree that mirrors the actual DOM tree. The virtual DOM contains a single `div` element that contains a `button` element and a `p` element.

```
{
  type: 'div',
  props: {},
  children: [
    {
      type: 'button',
      props: { onClick: handleLike },
      children: ['Like']
    }
  ]
}
```

```
    },  
    {  
      type: 'p',  
      props: {},  
      children: [0, ' Likes' ]  
    }  
  ]  
}
```

The children property of the `p` element contains the value of the likes state variable, which is initially set to zero.

Clicking the Like Button

When the user clicks the like button, the `handleLike` function is called, which updates the `likes` state variable. React then creates a new virtual DOM tree that reflects the updated state.

```
{  
  type: 'div',  
  props: {},  
  children: [  
    {  
      type: 'button',  
      props: { onClick: handleLike },  
      children: ['Like']  
    },  
  ],  
}
```

```
{
  type: 'p',
  props: {},
  children: [1, ' Likes' ]
}
]
```

Notice that the virtual DOM tree contains the same elements as before, but the children property of the `p` element has been updated to reflect the new value of likes.

Reconciliation

After updating the virtual DOM, React performs a process called reconciliation to update the actual DOM. Reconciliation is the process of comparing the old virtual DOM tree with the new virtual DOM tree and determining which parts of the actual DOM need to be updated.

In our example, React compares the old virtual DOM tree with the new virtual DOM tree and finds that the `p` element has changed. React then updates the actual DOM to reflect the changes made in the virtual DOM tree.

React updates only the necessary parts of the actual DOM to minimize the number of DOM manipulations. This approach is much

faster and more efficient than updating the entire DOM tree every time there is a change.

Using the virtual DOM provides several benefits in React. Some of the key benefits are:

1. **Faster Rendering:** The virtual DOM allows React to perform updates faster by reducing the number of DOM manipulations needed.
2. **More Efficient Memory Usage:** The virtual DOM uses a light-weight representation of the DOM tree, which reduces memory usage and improves performance.
3. **Easier to Maintain:** The virtual DOM makes it easier to maintain and update the UI components by abstracting the actual DOM and providing a simple programming interface.
4. **Improved Cross-Browser Compatibility:** The virtual DOM helps to ensure consistent behavior across different browsers by abstracting away the actual DOM and allowing us to write declarative code that is independent of the browser.

The virtual DOM is a critical concept in React that enables efficient rendering of user interfaces. React uses the virtual DOM to abstract the actual DOM tree and perform operations on it, reducing the number of DOM manipulations needed and improving performance. Our example of a like button is an indication of how React creates a virtu-

al DOM tree that mirrors the actual DOM tree and updates it to reflect changes in the UI. In coming chapters, we will explore how React then performs reconciliation to update the actual DOM only where necessary, resulting in faster and more efficient rendering.

We will cover more of the virtual DOM further in the book, but for now, let's move on to the next section.

The Component Model

React revolutionized the web because it highly encouraged “thinking in components”: that is, breaking your app into smaller pieces, and adding them to a larger tree to compose your application. The component model is a key concept in React, and it's what makes React so powerful. Let's talk about why.

1. It encourages reusing the same thing everywhere so that if it breaks, you fix it in one place and it's fixed everywhere. This is called “DRY” (don't repeat yourself) and is a key concept in software engineering. It's also a key concept in React. For example, if we have a `Button` component, we can use it in many places in our app, and if we need to change the style of the button, we can do it in one place and it's changed everywhere.
2. React is more easily able to keep track of components and do performance magic like memoization, batching, and other opti-

mizations under the hood if it's able to identify specific components over and over again and track updates to them over time. This is called "keying". For example, if we have a `Button` component, we can give it a `key` prop and React will be able to keep track of the `Button` component over time and "know" when to update it, or when to skip updating it and continue making minimal changes to the user interface. Most components have implicit keys, but we can also explicitly provide them if we want to.

3. It helps us separate concerns and colocate logic closer to the parts of the user interface that it affects. For example, if we have a `RegisterButton` component, we can put the logic for what happens when the button is clicked in the same file as the `RegisterButton` component, instead of having to jump around to different files to find the logic for what happens when the button is clicked. The `RegisterButton` component would wrap a more simple `Button` component, and the `RegisterButton` component would be responsible for handling the logic for what happens when the button is clicked. This is called "composition".

React's component model is a fundamental concept that underpins the framework's popularity and success. This approach to development has numerous benefits, including increased modularity, easier debugging, and more efficient code reuse.

Let's explore the advantages of React's component model in greater depth, using code examples to demonstrate its power. We will begin by examining the concept of modularity and how it is enabled by the component model. We will then move on to discuss how the component model facilitates efficient code reuse, and how it helps developers to debug their applications more easily.

Modularity

Modularity is a critical concept in software development, as it allows developers to break up an application into smaller, more manageable pieces. By doing so, developers can create more maintainable code that is easier to work with and modify over time. Modularity also facilitates code reuse, as developers can create smaller, self-contained pieces of code that can be reused throughout an application or even across multiple applications.

React's component model is designed to promote modularity by breaking up an application into smaller, reusable components. Each component is a self-contained piece of code that is responsible for rendering a specific part of the user interface. This approach to development is particularly well-suited to building complex applications that require a high degree of modularity and maintainability.

Let's look at a simple example of how the component model can be used to create a modular user interface. In this example, we will create a simple form that consists of a text input and a submit button. We will use React to create two separate components: one for the input field and another for the button.

```
import React from "react";

function InputField(props) {
  return <input type="text" name={props.name} />;
}

function SubmitButton(props) {
  return <button type="submit">{props.label}</button>;
}
```

In this example, we have defined two separate components:

`InputField` and `SubmitButton`. Each of these components takes in some props (short for “properties”) and returns a piece of JSX that renders the appropriate UI element. By separating these two pieces of functionality into separate components, we have created a more modular, reusable design.

Now let's look at how we can use these components to create our form:

```
import React from "react";

function InputField(props) {
  return <input type="text" name={props.name} />;
}

function SubmitButton(props) {
  return <button type="submit">{props.label}</button>;
}

function Form(props) {
  return (
    <form onSubmit={props.onSubmit}>
      <InputField name="email" />
      <SubmitButton label="Submit" />
    </form>
  );
}
```

In this example, we have created a new component called `Form` that combines our `InputField` and `SubmitButton` components into a single, reusable form component. The `Form` component takes in a single prop called `onSubmit`, which is a function that will be called when the form is submitted.

By breaking our user interface down into smaller, reusable components, we have created a more modular and maintainable design. If we need to modify the input field or the submit button in the future, we can do so without affecting the rest of our code.

In the past, this used to be a lot more difficult to achieve, for example in Knockout.js, where you would have to create a custom binding for each component. In React, you can just create a component and use it anywhere in your application.

Code Reuse

One of the primary benefits of React's component model is that it enables efficient code reuse. Because each component is a self-contained piece of code, it can be reused throughout an application or even across multiple applications. This makes it easier for developers to create maintainable, reusable code that can be leveraged in multiple contexts.

Let's look at a more complex example to see how the component model can facilitate code reuse. In this example, we will create a simple accordion component that can be used to display a list of items that can be expanded or collapsed.

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

```

function AccordionItem(props) {
  const [isOpen, setIsOpen] = useState(false);

  const toggleOpen = () => {
    setIsOpen(!isOpen);
  };

  return (
    <div>
      <div onClick={toggleOpen}>{props.title}</div>
      {isOpen && <div>{props.content}</div>}
    </div>
  );
}

function Accordion(props) {
  return (
    <div>
      {props.items.map((item) => (
        <AccordionItem title={item.title} content={item.content} />
      ))}
    </div>
  );
}

```

In this example, we have defined two separate components:

`AccordionItem` and `Accordion`. The `AccordionItem` com-

ponent is responsible for rendering a single item in the accordion list, while the `Accordion` component is responsible for rendering the entire list of items.

The `AccordionItem` component uses React's `useState` hook to manage its state. We use the state variable `isOpen` to keep track of whether the item is currently expanded or collapsed, and we define a `toggleOpen` function that toggles the value of `isOpen` when the item is clicked. We then use conditional rendering to display the item's content only when it is expanded.

The `Accordion` component takes in an array of items as a prop and uses the `map` function to render each item as an `AccordionItem` component. By breaking the accordion down into smaller, reusable components, we have created a design that is easy to modify and extend.

We can now use the `Accordion` component to display a list of items in our application:

```
import React from "react";
import { Accordion } from "../components";

const items = [
  {
    title: "Item 1",
    content: "Item 1 content"
  },
  {
    title: "Item 2",
    content: "Item 2 content"
  }
];
```

```

        content: "Lorem ipsum dolor sit amet, consectetur",
      },
      {
        title: "Item 2",
        content:
          "Sed do eiusmod tempor incididunt ut labore",
      },
      {
        title: "Item 3",
        content:
          "Ut enim ad minim veniam, quis nostrud exer",
      },
    ],
  );

function App() {
  return (
    <div>
      <Accordion items={items} />

    </div>
  );
}

export default App;

```

In this example, we have created an array of items and passed it as a prop to the `Accordion` component. The `Accordion` component then renders each item as an `AccordionItem` component, display-

ing its title and content. Because we have defined these components as reusable, self-contained units of code, we can easily reuse them in other parts of our application or even in other applications altogether.

Debugging

Another advantage of React's component model is that it makes debugging easier. Because each component is responsible for rendering a specific part of the user interface, it is easier to isolate and fix bugs when they occur.

Let's look at a simple example to see how this works. In this example, we will create a simple counter component that displays a number and allows the user to increment or decrement it.

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

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };
}
```

```
};

return (
  <div>
    <div>{count}</div>
    <button onClick={increment}>+</button>
    <button onClick={decrement}>-</button>
  </div>
);
}
```

In this example, we have defined a `Counter` component that uses React's `useState` hook to manage its state. We use the state variable `count` to keep track of the current count, and we define two functions, `increment` and `decrement`, that update the count when the user clicks the corresponding buttons.

Now let's say that we discover a bug in our `Counter` component: when the user clicks the decrement button and the count reaches zero, the count continues to decrease, resulting in negative numbers. Using the component model, we can isolate the bug to a single component and quickly identify the source of the problem.

We can start by adding some `console.log` statements to our decrement function to see what is happening:

```
const decrement = () => {  
  console.log("Before decrement: ", count);  
  setCount(count - 1);  
  console.log("After decrement: ", count);  
};
```

When we run our application and click the decrement button, we can see the output in the console:

```
Before decrement: 1  
After decrement: 1  
Before decrement: 0  
After decrement: 0  
Before decrement: -1  
After decrement: -1
```

We can see that the `count` variable is being updated correctly, but the value of `count` is not being updated immediately. This is because the `setCount` function is asynchronous and does not update the count variable immediately.

To fix this issue, we can modify our `decrement` function to use the previous state value instead of the current state value:

```
const decrement = () => {  
  setCount((prevCount) => prevCount - 1);  
};
```

By using the previous state value, we can ensure that the count variable is always updated correctly.

React's component model is a powerful tool that allows developers to build complex, modular user interfaces with ease. By breaking the user interface down into smaller, reusable components, we can create designs that are easy to modify and extend. The component model also makes debugging easier, as each component is responsible for rendering a specific part of the user interface.

So far, we have explored the advantages of React's component model and demonstrated how it can be used to build a variety of user interfaces. We have also looked at several examples of how the component model can be used to create reusable, self-contained units of code. There are more reasons why the component mode is so revolutionary that we'll get into throughout the course of this book as we become more and more fluent in React, but for now, let's wrap up and answer our own question "why is React a thing?"

Answering the Question "Why is React a Thing?"

React is a thing because it allows developers to build user interfaces with greater ease and reliability by enabling us to declaritively express *what we'd like on the screen* while React takes care of the *how*, as it makes incremental updates to the DOM in a safe, predictable, and efficient manner. It also encourages us to think in components, which helps us separate concerns and reuse code more easily. It is battle tested at Facebook, and designed to be used at scale. It's also open source and free to use.

React also has a vast and active ecosystem, with a wide range of tools, libraries, and resources available to developers. This ecosystem includes tools for testing, debugging, and optimizing React applications, as well as libraries for common tasks such as data management, routing, and state management. Additionally, the React community is highly engaged and supportive, with many online resources, forums, and communities available to help developers learn and grow.

React is platform agnostic, meaning that it can be used to build web applications for a wide range of platforms, including desktop, mobile, and VR. This flexibility makes React an attractive option for developers who need to build applications for multiple platforms, as it allows them to use a single codebase to build applications that run across multiple devices.

Finally, React is backed by Facebook, which is one of the largest and most influential technology companies in the world. This backing provides developers with confidence that React is a stable and well-supported technology, with a long-term roadmap for future development. Additionally, Facebook's backing has helped to promote and popularize React, making it a widely recognized and respected technology in the development community.

To conclude, React's value proposition is centered around its component-based architecture, declarative programming model, virtual DOM, JSX, extensive ecosystem, platform agnostic nature, and backing by Facebook. Together, these features make React an attractive option for developers who need to build fast, scalable, and maintainable web applications. Whether you're building a simple website or a complex enterprise application, React can help you achieve your goals more efficiently and effectively than many other technologies. Let's review.

Chapter Review

In this chapter, we covered a brief history of React, its initial value proposition, and how it solves the problems of unsafe, unpredictable, and inefficient user interface updates at scale. We also talked about the component model and why has been revolutionary for interfaces on the web. Let's recap what we've learned. Ideally after this chapter, you are more informed about the roots of React and where it comes from as well as its main strengths and value proposition.

Review Questions

Let's make sure you've fully grasped the topics we covered. Take a moment to answer the following questions:

1. What was the motivation to create React?
2. Why should I use React today? What happens if I don't?
3. How is React different from other JavaScript libraries like Knockout, Backbone, and jQuery?
4. What are the benefits of declarative programming abstractions?
5. What is the component model? Why is it so revolutionary?

If you have trouble answering these questions, this chapter may be worth another read. If not, let's explore the next chapter.

Up Next

In chapter 2 we will dive a little deeper into this declarative abstraction that allows us to express what we want to see on the screen: the syntax and inner workings of JSX—the language that looks like HTML in JavaScript that got React into a lot of trouble in its early days.

Chapter 2. JSX

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

In the previous chapter, we learned about the basics of React and its origin story, comparing it to other popular JavaScript libraries and frameworks of its time. We learned about the true value proposition of React and why it’s a thing. In this chapter, we’ll learn about JSX, which is a syntax extension for JavaScript that allows us to write HTML-like code within our JavaScript code.

JS is JavaScript. Does that mean JSX is JavaScript version 10? Like Mac OS X? Is it “JS Xtra”? You might be wondering what the X in JSX

stands for. I could understand you thinking it means '10' or 'Xtra', which would both be good guesses! But the X in JSX stands for **JavaScript Syntax eXtension**. It's also sometimes called **JavaScript XML**. Let's explore this language extension further while also exploring some creative ways we can leverage it to build powerful React applications.

JavaScript XML?

If you've been around the web for a while, you might remember the term **AJAX** or **Asynchronous JavaScript and XML** from around the 2000s. AJAX was essentially a new way of using existing technologies at the time to create highly interactive web pages that update asynchronously and in-place, instead of the status quo at the time: each interaction would load an entire new page.

Using tools like `XMLHttpRequest` in the browser, the browser would initiate an asynchronous (that is, non-blocking) request over HTTP (HyperText Transfer Protocol). The response to this request traditionally would be a response in XML. Today, we tend to respond with JSON instead. This is likely one of the reasons why `fetch` has overtaken `XMLHttpRequest`, since we don't even request XML anymore and the name can be misleading.

JSX is a syntax extension for JavaScript that allows developers to write HTML-like code within their JavaScript code. It was originally developed by Facebook to be used with React, but it has since been adopted by other libraries and frameworks as well. JSX is not a separate language, but rather a syntax extension that is transformed into regular JavaScript code by a compiler or transpiler. When JSX code is compiled, it is transformed into plain JavaScript code. More on this later.

JSX syntax looks similar to HTML, but there are some key differences. For example, JSX uses curly braces `{ }` to embed JavaScript expressions within the HTML-like code. Additionally, JSX attributes are written in camelCase instead of HTML attributes, and some HTML elements are written in lowercase instead of uppercase.

I should mention that **it's possible to create React applications without JSX at all**, but the code tends to become hard to read, reason about, and maintain. Still, if you want to, you can. Let's look at a React component expressed with JSX and without.

Here's an example of a list with JSX (figure 0):

```
const MyComponent = () => (  
  <section id="list">  
    <h1>This is my list!</h1>  
    <p>Isn't my list amazing? It contains amazing
```

```

    <ul>
      {amazingThings.map((t) => (
        <li key={t.id}>{t.label}</li>
      ))}
    </ul>
  </section>
);

```

Here's an example of a list without JSX:

```

const MyComponent = () =>
  React.createElement(
    "section",
    { id: "list" },
    React.createElement("h1", {}, "This is my list"),
    React.createElement(
      "p",
      {},
      "Isn't my list amazing? It contains amazing things"
    ),
    React.createElement(
      "ul",
      {},
      amazingThings.map((t) =>
        React.createElement("li", { key: t.id }, t.label)
      )
    )
  )

```

```
) ;
```

Do you see the difference? You might find the first example with JSX far more readable and maintainable than the latter. The former is JSX, the latter is vanilla JS. Let's talk about its tradeoffs.

Benefits of JSX

There are several benefits to using JSX in web development:

1. **Easy to Read and Write:** JSX syntax is easy to read and write, especially for developers who are familiar with HTML.
2. **Improved Performance:** JSX code can be compiled into more efficient JavaScript code than traditional HTML templates, resulting in improved performance.
3. **Strong Typing:** JSX allows for strong typing, which can help catch errors before they occur.
4. **Encourages Component-Based Architecture:** JSX encourages a component-based architecture, which can help make code more modular and easier to maintain.
5. **Widely Used:** JSX is widely used in the React community, and is also supported by other libraries and frameworks.

Drawbacks of JSX

There are also some drawbacks to using JSX:

1. **Learning Curve:** Developers who are not familiar with JSX may find it difficult to learn and understand.
2. **Requires Compilation:** JSX code must be compiled into regular JavaScript code before it can be executed, which adds an extra step to the development process.
3. **Mixing of Concerns:** Some developers argue that JSX mixes concerns by combining HTML-like code with JavaScript code, making it harder to separate presentation from logic.
4. **IDE Support:** Some IDEs do not have strong support for JSX, which can make development more difficult.

Despite its drawbacks, JSX has become a popular choice for web developers, particularly those of us working with React. It offers a powerful and flexible way to create components and build user interfaces, and has been embraced by a large and active community. In addition to its use with React, JSX has also been adopted by other libraries and frameworks, including Vue.js, Angular, and Ember.js. This shows that JSX has wider applications beyond just React, and its popularity is likely to continue to grow in the coming years.

Overall, JSX is a powerful and flexible tool that can help us build dynamic and responsive user interfaces. JSX was created with one job: make expressing, presenting, and maintaining the code for React

components simple while preserving powerful capabilities such as iteration, computation, and inline execution.

JSX becomes vanilla JavaScript before it makes it to the browser. How does it accomplish this? Let's take a look under the hood!

Under the Hood

How does one make a language extension? How do they work? To answer this question, we need to understand a little bit about programming languages themselves. Specifically, we need to explore how exactly code like this:

```
const a = 1;  
let b = 2;  
  
console.log(a + b);
```

outputs `3`. Understanding this will help us understand JSX better, which will in turn help us understand React deeper, thereby increasing our fluency with React.

How Does Code Work?

The code snippet in the section above is literally just text. How is this interpreted by a computer and then executed? For starters, it's not a big clever `RegExp` (Regular Expression) that can identify key words in a text file. I once tried to build a programming language this way and failed miserably, because regular expressions are often hard to get right, harder still to read back and mentally parse, and quite difficult to maintain.

Instead, this code is compiled using a compiler. A compiler is a software tool that translates source code written in a high-level programming language into machine code that can be executed by a computer. The process of compiling involves several steps, including lexical analysis, parsing, semantic analysis, optimization, and code generation. Let's explore each of these steps in more detail and discuss the role of compilers in the modern software development landscape.

A compiler uses a three-step process (at least in JavaScript anyway) that is in play here. These steps are called **tokenization**, **parsing**, and **code generation**. Let's look at each of these steps in more detail.

- **Tokenization** is essentially breaking up a string of characters into meaningful **tokens**. When a tokenizer is stateful and each token contains state about its parents and/or children, a tokenizer is called a **lexer**. Lexing is basically stateful tokenization.

Lexers have **lexer rules** that, in common cases, use a regular expressions or similar to detect key words in a text string representing a programming language. The lexer then maps these key words to some type of enumerable value. For example,

- `const` becomes `0`
- `let` becomes `1`
- `function` becomes `2`
- etc.

Once a string is tokenized or lexed, we move on to the next step.

- **Parsing** is the process of taking the tokens and converting them into an abstract syntax tree (AST). The syntax tree is a data structure that represents the structure of the code. For example, the code snippet we looked at in the section above would be represented as a syntax tree like this:

```
{
  type: "Program",
  body: [
    {
      type: "VariableDeclaration",
      declarations: [
        {
          type: "VariableDeclarator",
          id: {
            type: "Identifier",
            name: "a"
          }
        }
      ]
    }
  ]
}
```

```
    },
    init: {
      type: "Literal",
      value: 1,
      raw: "1"
    }
  },
],
kind: "const",
},
{
  type: "VariableDeclaration",
  declarations: [
    {
      type: "VariableDeclarator",
      id: {
        type: "Identifier",
        name: "b"
      },
      init: {
        type: "Literal",
        value: 2,
        raw: "2"
      }
    }
  ]
},
kind: "let",
},
```

```
{
  type: "ExpressionStatement",
  expression: {
    type: "CallExpression",
    callee: {
      type: "Identifier",
      name: "console"
    },
    arguments: [
      {
        type: "BinaryExpression",
        left: {
          type: "Identifier",
          name: "a"
        },
        right: {
          type: "Identifier",
          name: "b"
        },
        operator: "+"
      }
    ]
  }
}
]
```

The string, thanks to the parser, becomes effectively a JSON object. As programmers, when we have a data structure like this, we can do some really fun things. Language engines use these data structures to complete the process with the third step,

- **Code Generation**, where the compiler generates machine code from the AST. This involves translating the code in the AST into a series of instructions that can be executed directly by the computer's processor. The resulting machine code is then executed by the JavaScript engine. Overall, the process of converting an AST into machine code is complex and involves a lot of different steps. However, modern compilers are highly sophisticated and can produce highly optimized code that runs efficiently on a wide range of hardware architectures.

There are several types of compilers, each with different characteristics and use cases. Some of the most common types of compilers include:

- **Native Compilers** - These are compilers that produce machine code that can be executed directly by the target platform's processor. Native compilers are typically used to create standalone applications or system-level software.
- **Cross-Compilers** - These are compilers that produce machine code for a different platform than the one on which the compiler

is running. Cross-compilers are often used in embedded systems development or when targeting specialized hardware.

- **Just-In-Time (JIT) Compilers** - These are compilers that translate code into machine code at runtime, rather than ahead of time. JIT compilers are commonly used in virtual machines, such as the Java Virtual Machine, and can offer significant performance advantages over traditional interpreters.
- **Interpreters** - These are programs that execute source code directly, without the need for compilation. Interpreters are typically slower than compilers, but offer greater flexibility and ease of use.

To run JavaScript code in browsers, we use a Just-In-Time (JIT) compiler. In a JIT compiler, the source code is first compiled into an intermediate representation, such as bytecode, which is then translated into machine code as needed. This allows the compiler to optimize the code based on runtime information, such as the values of variables and the execution paths taken by the program. JavaScript engines inside web browsers include a JIT compiler. The JIT compiler translates JavaScript code into machine code on the fly, as the code is executed.

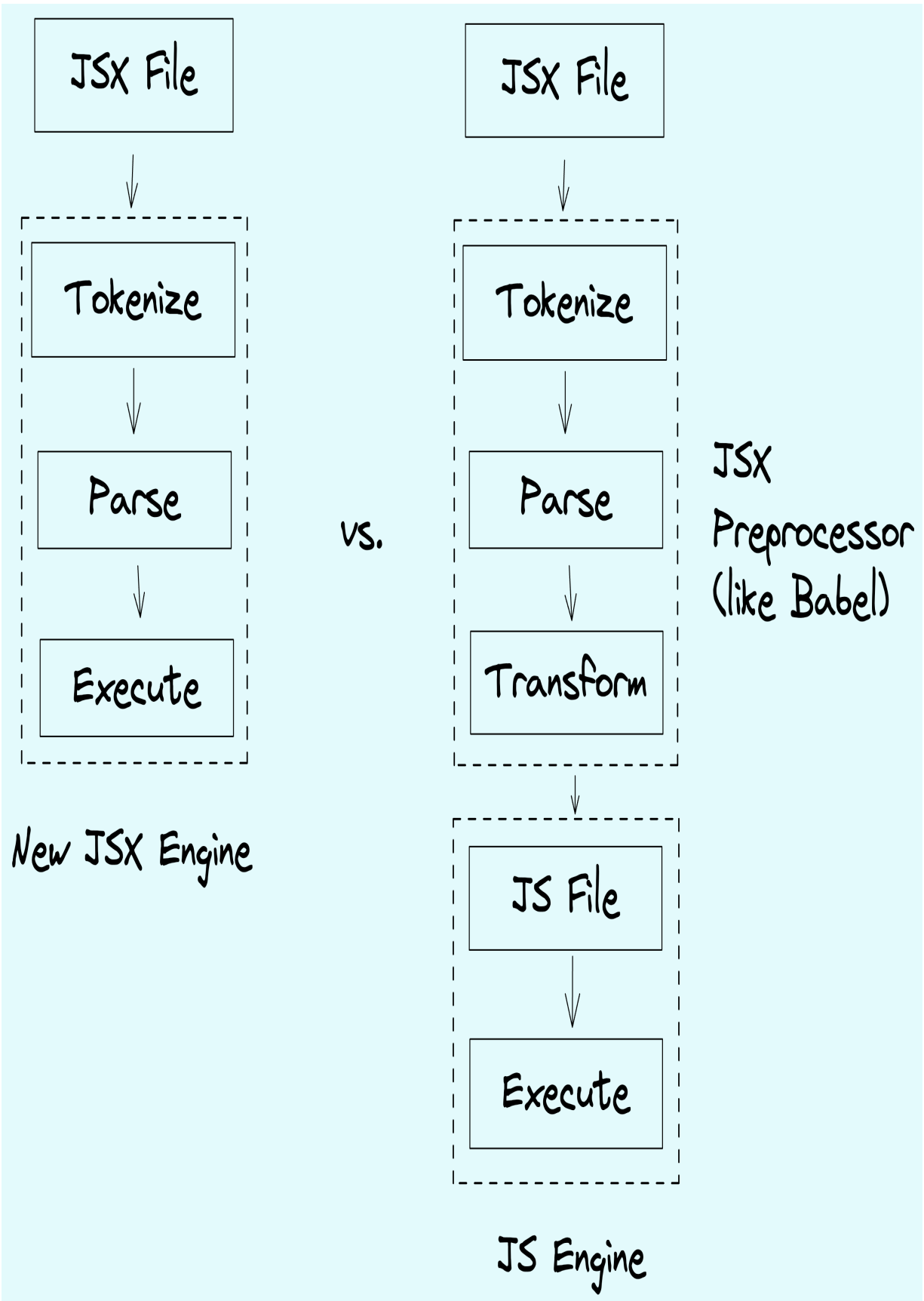
Runtimes usually interface with engines to provide more contextual helpers and features for their specific environment. The most popular JavaScript runtime, by far, is the common web browser; like Google

Chrome. This runtime gives JavaScript context like the `window` object and the `document` object. Another very popular runtime is Node.js, which is a JavaScript runtime that runs on servers. If you've worked with both browsers and Node.js before, you may have noticed Node.js does not have a global `window` object. This is because it's a different runtime and, as such, provides different context. Cloudflare created a similar runtime called **Workers** whose sole responsibility is executing JavaScript on **globally distributed edge servers**, but we're digressing. How does this all relate to JavaScript eXtended syntax?

Extending JavaScript Syntax

Now that we understand how one would extend JavaScript syntax, how does JSX work? How would we do it? To extend JavaScript syntax, we'd need to either have a different engine that can understand our new syntax, or deal with our new syntax before it reaches the engine. The former is nearly impossible to do because engines require a lot of thought to create and maintain since they tend to be widely used. If we decided to go with that option, it might take years or decades before we can use our extended syntax! We'd then even have to make sure our "bespoke special engine™" is used everywhere. How would we convince browser vendors et al to switch to our unpopular new thing? This wouldn't work.

The latter is quicker: let's explore how we can deal with our new syntax before it reaches the engine. To do this, we'd need to create our own lexer and parser that can understand our extended language: that is, take a text string of code and understand it. Then, instead of generating machine code as is traditional, we can take this syntax tree and instead generate plain old regular vanilla JavaScript that all current engines can understand. This is precisely what **Babel** in the JavaScript ecosystem does, along with other tools like TypeScript, Traceur, and swc.



Because of this, JSX cannot be used directly in the browser, but instead requires a “build step” where a custom parser runs against it, then compiles it into a syntax tree. This code is then transformed into vanilla JavaScript in a final distributable bundle. This is called “**transpilation**”: “**trans**formed, then **comp**iled code”.

Whew! What a deep dive! Now that we understand how we can build our own extension of JavaScript, let’s look at what we can do with this specific extension JSX.

The JSX Pragma

It all starts with `<`, which is an unrecognizable character in JavaScript. When a JavaScript engine encounters this, it throws a `SyntaxError: Unexpected token '<'`. In JSX, this “JSX Pragma” can be transpiled into a function call. The name of the function to call when a parser sees `<` is configurable, and defaults to the function `React.createElement`. The signature of this function is expected to be this:

```
function pragma(tag, props, ...children)
```

That is, it receives a `tag`, `props`, and `children` as arguments. Here's how JSX maps to regular JavaScript syntax:

- The following JSX code:

```
<MyComponent prop="value">contents</MyComponent>
```

- Becomes the following JavaScript code:

```
React.createElement(MyComponent, { prop: "value"}, "contents")
```

Notice the mapping between the tag (`MyComponent`), the props (`prop="value"`) and the children (`"contents"`)? This is essentially the role of the JSX pragma: syntax sugar over multiple, recursive function calls. The JSX pragma is effectively an alias: `<` instead of `React.createElement`.

Expressions

One of the most powerful features of JSX is the ability to execute code inside a tree of elements. To iterate over a list as we did under the section titled “Under the Hood”, we can put executable code in-

side curly brackets like we did with our `map` earlier in this chapter. If we want to show a sum of two numbers in JSX, we'd do it like this:

```
const a = 1;
const b = 2;

const MyComponent = () => <Box>Here's an expression: 3
```

This will render `Here's an expression: 3`, because the stuff inside curly brackets is executed as an expression. Using JSX expressions, we can iterate over lists, and execute a variety of expressions including conditional checks with ternary operations, string replacement, and more.

Here's another example with a conditional check using a ternary operation:

```
const a = 1;
const b = 2;

const MyComponent = () => <Box>Is b more than a? YES
```

This will render `Is b more than a? YES` since the comparison is an evaluated expression. For posterity, it's worth mentioning here

that JSX expressions are exactly that: expressions. It is not possible to execute statements inside of a JSX element tree. This will not work:

```
const MyComponent = () => <Box>Here's an express:
  const a = 1;
  const b = 2;

  if (a > b) {
    3
  }
}</Box>;
```

It doesn't work because statements do not return anything and are considered side effects: they set state without yielding a value. After statements and computations, how would we print a value inline? Notice in the previous example, we just put the number 3 in there on line 6. How is our renderer supposed to know we intend to print 3? This is why expressions are evaluated, but statements are not.

Okay, we're getting pretty good at this JSX thing. Let's dive into some patterns we can use with JSX to boost our React fluency.

JSX Patterns

Software design patterns are commonly used solutions to recurring problems in software development. They provide a way to solve problems that have been encountered and solved by other developers, saving time and effort in the software development process. They are often expressed as templates or guidelines for creating software that can be used in different situations. Software design patterns are typically described using a common vocabulary and notation, which makes them easier to understand and communicate among developers. They can be used to improve the quality, maintainability, and efficiency of software systems.

Software design patterns are important for several reasons:

1. **Reusability:** Design patterns provide reusable solutions to common problems, which can save time and effort in software development.
2. **Standardization:** Design patterns provide a standard way of solving problems, which makes it easier for developers to understand and communicate with each other.
3. **Maintainability:** Design patterns provide a way to structure code that is easy to maintain and modify, which can improve the longevity of software systems.
4. **Efficiency:** Design patterns provide efficient solutions to common problems, which can improve the performance of software systems.

Software design patterns usually naturally arrive over time in response to real-world needs. These patterns solve specific problems that engineers experience, and find their way into an “engineer’s arsenal” of tools to use in different use case. **One pattern is not inherently worse than the other**, each has its place.

Most patterns help us identify ideal levels of abstraction: how we can write code that ages like fine wine instead of accruing extra state and configuration to the point where it becomes unreadable and/or unmaintainable. This is why a common consideration when picking a design pattern is **control**: how much of it we give to users vs. how much of it our program handles.

With that, let’s dive in to some popular React patterns, following a rough chronological order of when these patterns emerged.

Presentational/Container Components

It’s common to see a React design pattern that is a combination of two components: a **presentational component** and a **container component**. The presentational component is the one that renders the UI, and the container component is the one that handles the state of the UI. Consider a counter. This is how a counter would look implementing this pattern:

```
const PresentationalCounter = (props) => {
```

```
    return (
      <section>
        <button onClick={props.increment}>+</button>
        <button onClick={props.decrement}>-</button>
        <button onClick={props.reset}>Reset</button>
        <h1>Current Count: {props.count}</h1>
      </section>
    );
  };

const ContainerCounter = () => {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  const reset = () => setCount(0);

  return (
    <PresentationalCounter
      count={count}
      increment={increment}
      decrement={decrement}
      reset={reset}
    />
  );
};
```

In this example, we've got two components: a presentational component (`PresentationCounter`) and a container component (`ContainerCounter`). The presentational component is the one that renders the UI, and the container component is the one that handles the state.

Why is this a thing? This pattern is quite useful because of the principle of **single responsibility**. Instead of having a component be responsible for how it should look and how it should work, we split these concerns. The result? `PresentationCounter` can be passed between other stateful containers and preserve the look we want, while `ContainerCounter` can be replaced with another stateful container and preserve the functionality we want.

We can also unit test `ContainerCounter` in isolation and instead visually test (using Storybook or similar)

`PresentationCounter` in isolation. We can also assign engineers or engineering teams more comfortable with visual work to `PresentationCounter` while assigning engineers who prefer data structures and algorithms to `ContainerCounter` .

We have so many more options because of this decoupled approach. For these reasons, the Presentational/Container component pattern has gained quite a lot of popularity and is still in use today.

Higher Order Components (HOC)

From Wikipedia,

In mathematics and computer science, a higher-order function (HOF) is a function that does at least one of the following: takes one or more functions as arguments (i.e. a procedural parameter, which is a parameter of a procedure that is itself a procedure), returns a function as its result.

In the JSX world, an HOC is basically this: a component that takes another component as an argument and returns a new component that is the result of the composition of the two. HOC's are great for **shared behavior across components that we'd rather not repeat**.

For example: many web applications need to request data from some data source asynchronously. Loading and error states are often inevitable, but we sometimes forget to account for them in our software. If we manually add `loading`, `data`, and `error` props to our components, the chance we miss a few gets even higher. Let's consider a basic todo list app:

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

```

        fetch("https://mytodolist.com/items")
            .then((res) => res.json())
            .then(setData);
    }, []);

    return <BasicTodoList data={data} />;
};

```

This app has a few problems. We don't account for loading or error states. Let's fix this.

```

const App = () => {
    const [isLoading, setIsLoading] = useState(true);
    const [data, setData] = useState([]);
    const [error, setError] = useState([]);

    useEffect(() => {
        fetch("https://mytodolist.com/items")
            .then((res) => res.json())
            .then((data) => {
                setIsLoading(false);
                setData(data);
            })
            .catch(setError);
    }, []);

    return isLoading ? (

```

```
      "Loading..."
    ) : error ? (
      error.message
    ) : (
      <BasicTodoList data={data} />
    );
  };
};
```

Yikes. This got pretty unruly pretty fast. Moreover, **this solves the problem for just one component**. Do we need to add these pieces of state (i.e. loading, data, and error) to each component that interacts with a foreign data source? This is a **cross-cutting concern**, and exactly where higher-order components shine.

Instead of repeating this loading, error, data pattern for each component that talks to a foreign data source asynchronously, we can use a higher-order component factory to deal with these states for us. Let's consider a `withAsync` higher-order component factory that remedies this.

```
const TodoList = withAsync(BasicTodoList);
```

`withAsync` will deal with loading and error states, and render any component when data is available. Let's look at its implementation.

```

const withAsync = (Component) => (props) => {
  if (props.loading) {
    return "Loading...";
  }

  if (props.error) {
    return error.message;
  }

  return (
    <Component
      // Pass through whatever other props we give
      {...props}
    />
  );
};

```

So now, when any `Component` is passed into `withAsync`, we get a new component that renders appropriate pieces of information based on its props. This changes our initial component into something more workable:

```

const TodoList = withAsync(BasicTodoList);

const App = () => {
  const [isLoading, setIsLoading] = useState(true);

```

```
const [data, setData] = useState([]);
const [error, setError] = useState([]);

useEffect(() => {
  fetch("https://mytodolist.com/items")
    .then((res) => res.json())
    .then((data) => {
      setIsLoading(false);
      setData(data);
    })
    .catch(setError);
}, []);

return <TodoList loading={isLoading} error={error} data={data}>;
```

No more nested ternaries, and the `TodoList` itself can show appropriate information depending on whether it's loading, has an error, or has data. Since the `withAsync` HOC factory deals with this cross-cutting concern, we can wrap any component that talks to an external data source with it and get back a new component that responds to `loading` and `error` props. Consider a blog:

```
const Post = withAsync(BasicPost);
const Comments = withAsync(BasicComments);
```

```
const Blog = ({ req }) => {
  const { loading: isPostLoading, error: postLoadError } = useQuery(
    req.query.postId
  );
  const { loading: areCommentsLoading, error: commentLoadError } = useQuery(
    req.query.postId,
  );

  return (
    <>
      <Post
        id={req.query.postId}
        loading={isPostLoading}
        error={postLoadError}
      />
      <Comments
        postId={req.query.postId}
        loading={areCommentsLoading}
        error={commentLoadError}
      />
    </>
  );
};

export default Blog;
```

In this example, both `Post` and `Comments` use the `withAsync` HOC pattern which returns a newer version of `BasicPost` and `BasicComments` respectively that now respond to `loading` and `error` props. The behavior for this cross-cutting concern is centrally managed in `withAsync`'s implementation, so we account for loading and error states “for free” just by using the HOC pattern here.

Render Props

Since we've talked about JSX expressions above, a common pattern is to have props that are functions that receive component-scoped state as arguments to facilitate code reuse. Here's a simple example:

```
<WindowSize
  render={({ width, height }) => (
    <div>
      Your window is {width}x{height}px
    </div>
  )}
/>
```

Notice how there's a prop called `render` that receives a function as a value? This prop even outputs some JSX markup that's actually rendered. But why? Turns out `WindowSize` does some magic inter-

nally to compute the size of a user's window, and then calls `props.render` to return the structure we declare.

Let's take a look at `WindowSize` to understand this a bit more:

```
const WindowSize = (props) => {  
  const [size, setSize] = useState({ width: -1, height: -1 });  
  
  useEffect(() => {  
    const handleResize = () => {  
      setSize({ width: window.innerWidth, height: window.innerHeight });  
    };  
    window.addEventListener("resize", handleResize);  
    return () => window.removeEventListener("resize", handleResize);  
  }, []);  
  
  return props.render(size);  
};
```

From this example, what we can see is that `WindowSize` uses an event listener to store some stuff in state on every resize, but the component itself is headless: it has no opinions about what UI to present. Instead, it yields control to whatever parent is rendering it and calls the **render prop** it's supplied: effectively inverting control to its parent for the rendering job.

This helps a component that depends on the window size for rendering receive this information without duplicating the `useEffect` blocks and keeping our code a little bit more DRY (Don't Repeat Yourself). This pattern is no longer as popular, and has since been effectively replaced with React Hooks (more on them later in the book).

Children as a Function

Since `children` is a prop, some have preferred to drop the `render` prop name altogether and instead just use `children`. This would change the use of `WindowSize` to look like this:

```
<WindowSize>
  ({ { width, height } }) => (
    <div>
      Your window is {width}x{height}px
    </div>
  )
</WindowSize>
```

Some React authors prefer this, because it's truer to the intent of the code: `WindowSize` in this case looks a bit like a React Context, and whatever we display tends to feel like children that consume this con-

text. Still, React hooks eliminate the need for this pattern altogether, so maybe proceed with caution.

Control Props

The Control Props pattern is a powerful way to control multiple components dependent on a common piece of state. React's own documentation is pretty clear about the concept of **controlled components**, where a component's state is controlled by React instead of the component itself, which is called an uncontrolled component.

Here's an example of an uncontrolled component:

```
const Form = () => {  
  return (  
    <form>  
      <input type="text" />  
      <button type="submit">Submit</button>  
    </form>  
  );  
};
```

The `input`'s value is self-contained by the browser, and React doesn't have access to it nor controls it. This is an **uncontrolled component** because React does not manage its state. Let's control it.

```

const Form = () => {
  const [value, setValue] = useState("");

  return (
    <form>
      <input
        type="text"
        value={value}
        onChange={(e) => setValue(e.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
};

```

By binding this component's `value` prop to the state managed by React, **this component is now controlled by React**. But you might be wondering, why we're talking about controlled components instead of **control props**. The control props pattern is close to this: a control prop is a prop on a child component whose value is state managed by a parent component.

Let's illustrate this with an example:

```

const ListPage = ({ items }) => {
  // ...

```

```

const [filter, setFilter] = useState("");
return (
  <section>
    <HeaderBar>
      <input
        type="checkbox"
        checked={filter.length > 0} // <- control prop to keep
      />{ " " }
      Filtered by term
      {filter}
      <input
        type="text"
        value={filter} // <- control prop to keep
        onChange={(e) => setFilter(e.target.value)}
      />
    </HeaderBar>
    <List
      items={items}
      filter={filter} // <- control prop to keep
    />
  </section>
);
};

```

From the above example, we use three control props based on one piece of state to keep our entire component tree in sync with our page's "filtered" state. Looking at this, we see that the control props

pattern helps us manage state and keep it synchronized across our React applications.

Prop Collections

We often need to bundle a whole bunch of props together. For example, when creating drag-and-drop user interfaces, there are quite a few props to manage:

- `onDragStart` to tell the browser what to do when a user starts dragging an element.
- `onDragOver` to identify a dropzone.
- `onDrop` to execute some code when an element is dropped on this element.
- `onDragEnd` to tell the browser what to do when an element is done being dragged.

Moreover, data/elements cannot be dropped in other elements by default. To allow an element to be dropped on another, we must prevent the default handling of the element. This is done by calling the `event.preventDefault` method for the `onDragOver` event for possible drop zone.

Since these props usually go together, and since `onDragOver` usually defaults to `event => { event.preventDefault();`

`moreStuff(); }`, we can collect these props together and reuse them in various components like so:

```
export const droppableProps = {
  onDragOver: (event) => {
    event.preventDefault();
  },
  onDrop: (event) => {},
};

export const draggableProps = {
  onDragStart: (event) => {},
  onDragEnd: (event) => {},
};
```

Now, if we have a React component we expect to behave like a dropzone, we can use the prop collection on it like this:

```
<Dropzone {...droppableProps} />
```

This is the prop collection pattern, and it makes a number of props reusable. This is often quite widely used in the accessibility space to include a number of `aria-*` props on accessible components. One problem that's still present though is that if we write a custom `onDragOver` prop and override the collection, we lose the

`event.preventDefault` call that we get out of the box using the collection.

This can cause unexpected behavior, removing the ability to drop a component on `Dropzone` :

```
<Dropzone
  {...draggableProps}
  onDragOver={() => {
    alert("Dragged!");
  }}
/>
```

It removes the ability to drop something on `Dropzone` since we no longer call `event.preventDefault` on it. Thankfully, we can fix this using prop getters.

Prop Getters

Prop getters essentially compose prop collections with custom props and merge them. From the example above, we'd like to preserve the `event.preventDefault` call in the `draggableProps` collection's `onDragOver` handler, while also adding a custom `alert("Dragged!");` call to it. We can do this using prop getters like so.

First, we'll change `droppableProps` the collection to a prop getter:

```
export const getDroppableProps = () => {
  return {
    onDragOver: (event) => {
      event.preventDefault();
    },
    onDrop: (event) => {},
  };
};
```

At this point, nothing has changed besides where we once exported a prop collection, we now export a function that returns a prop collection. This is a prop getter. Since this is a function, it can receive arguments—like a custom `onDragOver`. We can compose this custom `onDragOver` with our default one like so:

```
const compose =
  (...functions) =>
  (...args) =>
    functions.forEach((fn) => fn?.(...args));

export const getDroppableProps = ({
  onDragOver: replacementOnDragOver,
  ...replacementProps
}) => {
```

```
const defaultOnDragOver = (event) => {
  event.preventDefault();
};

return {
  onDragOver: compose(replacementOnDragOver, defaultOnDragOver),
  onDrop: (event) => {},
  ...replacementProps,
};
};
```

Now, we can use the prop getter like this:

```
<Dropzone
  {...getDroppableProps({
    onDragOver: () => {
      alert("Dragged!");
    },
  })}
/>
```

This custom `onDragOver` will compose into our default `onDragOver`, and both things will happen: `event.preventDefault()`, and `alert("Dragged!")`. This is the prop getter pattern.

Compound Components

Sometimes, we have accordion components like this:

```
<Accordion
  items={[
    { label: "One", content: "lorem ipsum for mor"
    { label: "Two", content: "lorem ipsum for mor"
    { label: "Three", content: "lorem ipsum for r
  ]}
/>
```

This component is intended to render a list similar to this, except **only one item** can be open at a given time.

- One
- Two, or
- Three

The inner workings of this component would look something like this:

```
export const Accordion = ({ items }) => {
  const [activeItemIndex, setActiveItemIndex] = u

  return (
    <ul>
```

```

        {items.map((item, index) => (
          <li onClick={() => setActiveItemIndex(index)}>
            <strong>{item.label}</strong>
            {index === activeItemIndex && i.content}
          </li>
        ))}
      </ul>
    );
  };

```

But what if we wanted a custom separator in between items **Two** and **Three**? What if we wanted the third link to be red or something? We'd probably resort to some type of hack like this:

```

<Accordion
  items={[
    { label: "One", content: "lorem ipsum for mor" },
    { label: "Two", content: "lorem ipsum for mor" },
    { label: "---" },
    { label: "Three", content: "lorem ipsum for r" }
  ]}
/>

```

But that wouldn't look the way we want. So we'd probably do more hacks on our current hack:

```

export const Accordion = ({ items }) => {
  const [activeItemIndex, setActiveItemIndex] = useState(0)

  return (
    <ul>
      {items.map((item, index) =>
        item === "----" ? (
          <hr />
        ) : (
          <li onClick={() => setActiveItemIndex(index)}>
            <strong>{item.label}</strong>
            {index === activeItemIndex && item.content}
          </li>
        )
      )}
    </ul>
  );
};

```

Now is that code we'd be proud of? I'm not sure. This is why we need **Compound Components**: they allow us to have a grouping of interconnected, distinct components that share state, but are atomically renderable, giving us more control of the element tree.

This accordion, expressed using the compound components pattern, would look like this:

```

<Accordion>
  <AccordionItem item={{ label: "One" }} />
  <AccordionItem item={{ label: "Two" }} />
  <AccordionItem item={{ label: "Three" }} />
</Accordion>

```

Let's explore how this pattern can be implemented in React. First, we'll start with a context that each part of the accordion can read from:

```

const AccordionContext = createContext({
  activeItemIndex: 0,
  setActiveItemIndex: () => 0,
});

```

Then, our `Accordion` component will just provide context to its children:

```

export const Accordion = ({ items }) => {
  const [activeItemIndex, setActiveItemIndex] = useState(0);

  return (
    <AccordionContext.Provider value={{ activeItemIndex, setActiveItemIndex }}>
      <ul>{children}</ul>
    </AccordionContext.Provider>
  );
};

```

```
);  
};
```

Now, let's create discrete `AccordionItem` components that consume and respond to this context as well:

```
export const AccordionItem = ({ item, index }) =>  
  // Note we're using the context here, not state  
  const { activeItemIndex, setActiveItemIndex } = useContext()  
  
  return (  
    <li onClick={() => setActiveItemIndex(index)}>  
      <strong>{item.label}</strong>  
      {index === activeItemIndex && i.content}  
    </li>  
  );  
};
```

Now that we've got multiple parts for our `Accordion` making it a compound component, our usage of the `Accordion` goes from this:

```
<Accordion  
  items=[  
    { label: "One", content: "lorem ipsum for mor  
    { label: "Two", content: "lorem ipsum for mor
```

```
    { label: "Three", content: "lorem ipsum for r  
  ]}  
/>
```

to this:

```
<Accordion>  
  {items.map((item, index) => (  
    <AccordionItem key={item.id} item={item} index={index} />  
  ))}  
</Accordion>
```

The benefit of this that we have far more control, while each `AccordionItem` is aware of the larger state of `Accordion`. So now, if we wanted to include a horizontal line between items `Two` and `Three`, we could break out of the `map` and go more manual if we wanted to:

```
<Accordion>  
  <AccordionItem key={items[0].id} item={items[0]} />  
  <AccordionItem key={items[1].id} item={items[1]} />  
  <hr />  
  <AccordionItem key={items[2].id} item={items[2]} />  
</Accordion>
```

Or, we could do something more hybrid like:

```
<Accordion>
  {items.slice(0, 2).map((item, index) => (
    <AccordionItem key={item.id} item={item} index={index} />
  ))}
  <hr />
  {items.slice(2).map((item, index) => (
    <AccordionItem key={item.id} item={item} index={index} />
  ))}
</Accordion>
```

This is the benefit of compound components: **they invert control of rendering to the parent, while preserving contextual state awareness among children.** The same approach could be used for a tab UI, where tabs are aware of the current tab state while having varying levels of element nesting.

Context Module

The context module pattern is a powerful pattern at scale where application state can be fairly complex. In essence, it works by exporting utility functions that mutate state by accepting `dispatch` functions as arguments in a tree-shakeable and lazy-loadable way.

Here's a contrived example with a counter, coming full circle from our first pattern which was also a counter. Let's start by creating a context:

```
import React from "react";

const CounterContext = React.createContext(null);
CounterContext.displayName = "CounterContext";
```

Now, let's create a provider so we can share this context with our React app:

```
export function CounterProvider({ children }) {
  const [state, dispatch] = useState({ counter: 0 });

  return (
    <CounterContext.Provider value={{ state, dispatch }}>
      {children}
    </CounterContext.Provider>
  );
}
```

The final step to implement this pattern is to export a few utility functions that can be used to manipulate state:

```
export const increment = (dispatch, by = 1) =>
  dispatch((oldValue) => oldValue + by);

export const decrement = (dispatch, by = 1) =>
  dispatch((oldValue) => oldValue - by);

export const reset = (dispatch) => dispatch(0);
```

Notice how they accept `dispatch` as their first argument. This makes them pluggable, and able to run in a higher degree of isolation. Now, we can use these functions in our application, and import them on-demand, reducing bundle size and preserving performance.

```
const PresentationalCounter = (props) => {
  return (
    <section>
      <button onClick={props.increment}>+</button>
      <button onClick={props.decrement}>-</button>
      <button onClick={props.reset}>Reset</button>
      <h1>Current Count: {props.count}</h1>
    </section>
  );
};

const ContainerCounter = () => {
  const { dispatch } = useContext(CounterContext);
```

```
return (  
  <PresentationalCounter  
    count={count}  
    increment={() => increment(dispatch)}  
    decrement={() => decrement(dispatch)}  
    reset={() => reset(dispatch)}  
  />  
);  
};
```

We can lazy load `increment`, `decrement`, and `reset` using **dynamic import** in JavaScript, which would mean shipping a smaller amount of first-load JavaScript to our users, and making things faster for them: a UI can render closer to instant, and then immediately become interactive via our context modules shortly after. This is a powerful pattern that can be used to build a large application while shipping a smaller amount of code.

Chapter Review

Okay, we've covered a fair amount of ground on the topic of JSX. At this point we should be feeling pretty confident (or even fluent, if you will) about the topic to the point where we can confidently explain aspects of it to people.

Review Questions

Let's make sure you've fully grasped the topics we covered. Take a moment to answer the following questions:

1. What is JSX?
2. What are some pros and cons of JSX?
3. What is the difference between JSX and HTML?
4. How do programming languages work?
5. What's the value of design patterns?

If you have trouble answering these questions, this chapter may be worth another read. If not, let's explore the next chapter.

Up Next

Now that we're pretty fluent with JSX, let's turn our attention to the next aspect of React and see how we can squeeze the most knowledge out of it to further boost our fluency. Let's explore the virtual DOM.

Chapter 3. The Virtual DOM

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

In this chapter, we’ll dive deep into the concept of virtual DOM (Document Object Model) and its significance in React. We’ll also explore how React uses the virtual DOM to make web development easier and more efficient.

As web applications become more complex, it becomes increasingly difficult to manage the “real DOM”, which is a complex and error-prone process as we’ll see soon enough. React’s virtual DOM provides an alternative solution to this problem.

Throughout this chapter, we'll explore the workings of React's virtual DOM, its advantages over the real DOM, and how it is implemented to make web development easier and more efficient. We'll also cover how React optimizes performance around the real DOM.

Through a series of code examples and detailed explanations, we'll understand the virtual DOM's role in React and how to take advantage of its benefits to create robust and efficient web applications. Let's get started!

An Intro to the Virtual DOM

The virtual DOM is a programming concept that allows web developers to create user interfaces in a more efficient and performant way. It does this by creating a virtual representation of the real DOM. The real DOM is a tree-like data structure that represents the HTML elements on a web page.

The virtual DOM is essentially a lightweight copy of the real DOM that is kept in memory. Whenever a change is made to the UI, the virtual DOM is updated first, and then the real DOM is updated to match the changes in the virtual DOM.

The reason for this is that updating the real DOM can be a slow and expensive process. We'll cover this in the next section, but the gist of

it is every time a change is made to the real DOM, the browser has to recalculate the layout of the page, repaint the screen, and perform other operations that can be time-consuming.

On the other hand, updating the virtual DOM is much faster since it doesn't involve any changes to the actual page layout. Instead, it is a simple JavaScript object that can be manipulated quickly and efficiently.

When updates are made to the virtual DOM, React uses a diffing algorithm to identify the differences between the old and new versions of the virtual DOM. This algorithm then determines the minimal set of changes required to update the real DOM, and these changes are applied in a batched and optimized way to minimize the performance impact.

By using the virtual DOM, React allows us to create more responsive and efficient user interfaces that are faster and more performant than traditional UIs that rely solely on the real DOM. This allows for a better user experience and can also improve the scalability and maintainability of web applications.

A better user experience in the context of the virtual DOM means that the web application feels more responsive and faster to the user. Not only that, but the use of the virtual DOM can also lead to a more con-

sistent and reliable user experience. Since updates are applied in a controlled and optimized manner, there is less chance of errors or inconsistencies in the UI.

Finally, we can think of the virtual DOM as a technique used by React to reduce the performance cost of updating a web page. In this chapter, we will explore the differences between the virtual DOM and the real DOM, the pitfalls of the real DOM, and how the virtual DOM helps in creating better user interfaces. We will also dive into React's implementation of the virtual DOM and the algorithms it uses for efficient updates.

The “Real” DOM

The Document Object Model, or the “Real” DOM, is the programming interface for web pages. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects, which can be modified with a scripting language such as JavaScript.

When an HTML page is loaded into a web browser, it is parsed and converted into a tree of nodes and objects, which is the DOM. The DOM is a live representation of the web page, meaning that it is constantly being updated as users interact with the page.

Here is an example of the real DOM for a simple HTML page:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example Page</title>
  </head>
  <body>
    <h1>Welcome to my page!</h1>
    <p>This is an example paragraph.</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </body>
</html>
```

In this example, the real DOM is represented by a tree-like structure that consists of nodes for each HTML element in the page. Here is what the tree structure would look like:

```
Document
- html
  - head
    - title
```

- body
 - h1
 - p
 - ul
 - li
 - li
 - li

Each node in the tree represents an HTML element, and it contains properties and methods that allow it to be manipulated through JavaScript. For example, we can use the `document.querySelector()` method to retrieve a specific node from the real DOM and modify its contents:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example Page</title>
  </head>
  <body>
    <h1 class="heading">Welcome to my page!</h1>
    <p>This is an example paragraph.</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </body>
</html>
```

```
</ul>
<script>
  // Retrieve the "heading" element from the
  const heading = document.querySelector(".he

  // Change the contents of the element
  heading.innerHTML = "Hello, world!";
</script>
</body>
</html>
```

In this example, we retrieve the `h1` element with the `class` of `"heading"` using the `document.querySelector()` method. We then modify the contents of the element by setting its `innerHTML` property to `"Hello, world!"`. This changes the text displayed on the page from `"Welcome to my page!"` to `"Hello, world!"`.

That doesn't seem too complicated, but there are a few things to note here. First, we are using the `document.querySelector()` method to retrieve the element from the real DOM. This method accepts a CSS selector as an argument and returns the first element that matches the selector. In this case, we are passing in the class selector `.heading`, which matches the `h1` element with the `class` of `"heading"`.

There's a bit of a danger here, because while the `document.querySelector` method is a powerful tool for selecting elements in the real DOM based on CSS selectors, one potential performance issue with this method is that it can be slow when working with large and complex documents because the method has to start at the top of the document and traverse downward to find the desired element, which can be a time-consuming process.

When we call `document.querySelector()` with a CSS selector, the browser has to search the entire document tree for matching elements. This means that the search can be slow, especially if the document is large and has a complex structure. In addition, the browser has to evaluate the selector itself, which can be a complex process depending on the complexity of the selector.

The performance of `document.querySelector` can also be impacted by the location of the element we are trying to find in the document tree. If the element is located deep within the tree, the browser has to traverse a large number of nodes before it can find the element, which can be a slow process.

Another potential performance issue with `document.querySelector` is that it returns only a single element, even if multiple elements match the selector. This means that if we want to select multiple elements with the same selector, we have to

call the method multiple times, which can further slow down our applications.

To mitigate the performance issues of

`document.querySelector`, there are several strategies we can use. One approach is to optimize our selectors to make them more efficient. For example, we can use IDs instead of classes to narrow down the scope of the search, or use more specific selectors to avoid unnecessary node traversal.

Another strategy is to use other methods like

`document.getElementsByTagName` or `document.getElementById` instead of `document.querySelector`, depending on our specific use case.

These methods are more limited in their scope, but they can be faster and more efficient when we know exactly what type of element we are looking for.

Overall, the performance of `document.querySelector` can be impacted by a number of factors, including the size and complexity of the document, the location of the element we are trying to find, and the specificity of the selector we are using. By optimizing our selectors and using other methods when appropriate, we can help to mitigate these issues and improve the performance of our web applications.

Another factor that can impact the performance of `document.querySelector` is the specificity of the selector. More complex and specific selectors require more computation to evaluate and may require the browser to search a larger portion of the document tree.

In contrast, `document.getElementById` always searches a specific area of the document tree, so it is generally more efficient and in most cases, `document.getElementById` is faster than `document.querySelector` because it is a much simpler and more specific method.

`document.getElementById` is specifically designed to locate elements based on their unique `id` attribute. This means that it only needs to search a limited portion of the document tree, since `id` attributes are unique to a single element. As a result, the browser can find the desired element more quickly and with less computational overhead than with `document.querySelector`.

On the other hand, `document.querySelector` is a more general-purpose method that allows us to search for elements using complex CSS selectors. While this flexibility is useful in many cases, it also requires the browser to do more work to evaluate the selector and search the document tree for matching elements.

It's worth noting that the performance difference between `document.getElementById` and `document.querySelector` may be negligible in small documents or when searching for elements in specific areas of the document tree. However, in larger and more complex documents, the difference can become more pronounced.

Overall, if we know the `id` attribute of the element we want to select, `document.getElementById` is the faster and more efficient method. However, if we need to search for elements using more complex selectors, `document.querySelector` is still a powerful and flexible tool that can help we accomplish our goals.

I'm sharing all of these nuanced details because I want you to understand the overall complexity of the DOM: working intelligently with the DOM is no small feat and, with React, we have a choice: do we navigate this minefield ourselves and occasionally step on landmines? Or do we use a tool that helps us navigate the DOM safely—the virtual DOM?

While we've discussed some small nuances in how we select elements here, we haven't had an opportunity to dive deeper into the pitfalls of working with the DOM directly. Let's do this quickly to fully understand the value that React's virtual DOM provides.

Pitfalls of the real DOM

The real DOM has several pitfalls that can make it difficult to build high-performance web applications. Some of these pitfalls include performance issues, cross-browser compatibility, and complexity.

Performance

One of the biggest issues with the real DOM is its performance. Whenever a change is made to the DOM, such as adding or removing an element, or changing the text or attributes of an element, the browser has to recalculate the layout and repaint the affected parts of the page. This can be a slow and resource-intensive process, especially for large and complex web pages.

For example, reading a DOM element's `offsetWidth` property may seem like a simple operation, but it can actually trigger a costly recalculation of the layout by the browser. This is because `offsetWidth` is a computed property that depends on the layout of the element and its ancestors, which means that the browser needs to ensure that the layout information is up-to-date before it can return an accurate value.

Consider the following example, where we have a simple HTML document with a single div element:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Reading offsetWidth example</title>
    <style>
      #my-div {
        width: 100px;
        height: 100px;
        background-color: red;
      }
    </style>
  </head>
  <body>
    <div id="my-div"></div>
    <script>
      var div = document.getElementById("my-div");
      console.log(div.offsetWidth);
    </script>
  </body>
</html>
```

When we load this document in a browser and open the developer console, we can see that the `offsetWidth` property of the `div` element is logged to the console. However, what we don't see is the behind-the-scenes work that the browser has to do to compute the value of `offsetWidth`.

To understand this work, we can use the Performance panel in our developer tools to record a timeline of the browser's activities as it loads and renders the page. When we do that, we can see that the browser is performing several layout and paint operations as it processes the document. In particular, we can see that there are two layout operations that correspond to the reading of `offsetWidth` in the script.

Each of these layout operations takes a significant amount of time to complete (in this case, about 2ms), even though they are just reading the value of a property. This is because the browser needs to ensure that the layout information is up-to-date before it can return an accurate value, which requires it to perform a full layout of the document.

In general, we should be careful when reading layout-dependent properties like `offsetWidth`, because they can cause unexpected performance problems. If we need to read the value of such properties multiple times, we should consider caching the value in a variable to avoid triggering unnecessary layout recalculations. Alternatively, we can use the `requestAnimationFrame` API to defer the reading of the property until the next animation frame, when the browser has already performed the necessary layout calculations.

To understand more about accidental performance issues with the real DOM, let's take a look at some examples. Consider the following

HTML document:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <ul id="list">
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </body>
</html>
```

Suppose we want to add a new item to the list using JavaScript. We might write the following code:

```
const list = document.getElementById("list");
const newItem = document.createElement("li");
newItem.textContent = "Item 4";
list.appendChild(newItem);
```

Notice we're using `getElementById` instead of `querySelector` here because:

1. We know the ID, and
2. We now know the performance tradeoffs

Let's keep going.

This code selects the `ul` element with the ID `"list"`, creates a new `li` element, sets its text content to `"Item 4"`, and appends it to the list. When we run this code, the browser has to recalculate the layout and repaint the affected parts of the page to display the new item.

This process can be slow and resource-intensive, especially for larger lists. For example, suppose we have a list with 1000 items, and we want to add a new item to the end of the list. We might write the following code:

```
const list = document.getElementById("list");
const newItem = document.createElement("li");
newItem.textContent = "Item 1001";
list.appendChild(newItem);
```

When we run this code, the browser has to recalculate the layout and repaint the entire list, even though only one item has been added. This can take a significant amount of time and resources, especially on slower devices or with larger lists.

To further illustrate this issue, consider the following example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
    <style>
      #list li {
        background-color: #f5f5f5;
      }
      .highlight {
        background-color: yellow;
      }
    </style>
  </head>
  <body>
    <ul id="list">
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
    <button onclick="highlight()">Highlight Item
    <script>
      function highlight() {
        const item = document.querySelector("#list
        item.classList.add("highlight");
      }
    </script>
```

```
</body>  
</html>
```

In this example, we have a list with three items and a button that highlights the second item when clicked. When the button is clicked, the browser has to recalculate the layout and repaint the entire list, even though only one item has changed. This can cause a noticeable delay or flicker in the UI, which can be frustrating for users.

Overall, the performance issues of the real DOM can be a significant challenge for us, especially when dealing with large and complex web pages. While there are techniques for mitigating these issues, such as optimizing selectors, using event delegation, or using CSS animations, they can be complex and difficult to implement.

As a result, many of us have turned to the virtual DOM as a solution to these issues. The virtual DOM allows us to create UIs that are more efficient and performant by abstracting away the complexities of the real DOM and providing a more lightweight way of representing the UI.

But... is it really necessary to save a few milliseconds? Well, CPU/processing performance is a critical factor that can greatly impact the success of an application. In today's digital age, where users expect fast and responsive websites, it's essential for us web developers to

prioritize CPU efficiency to ensure that our applications run smoothly and responsively.

Direct DOM manipulation that triggers layout recalculation (called reflows) and repaints can lead to increased CPU usage and processing times, which can cause delays and even crashes for users. This can be particularly problematic for users on low-powered devices, such as smartphones or tablets, which may have limited processing power and memory. In many parts of the world, users may be accessing our web apps on older or less capable devices, which can further compound the problem.

By prioritizing CPU efficiency, we can create applications that are accessible to users on a wide range of devices, regardless of their processing power or memory. This can lead to increased engagement, higher conversion rates, and ultimately, a more successful online presence.

React's virtual DOM has emerged as a powerful tool for building CPU-efficient web applications, using its efficient rendering algorithms can help minimize processing times and improve overall performance.

Cross Browser Compatibility

Another issue with the real DOM is cross-browser compatibility. Different browsers implement the DOM API differently, which can lead to inconsistencies and bugs in web applications. This was far more common around the time React was released, however, and is far less common now. Still, this can and did make it difficult for developers to create web applications that work seamlessly across different browsers and platforms.

One of the primary issues with cross-browser compatibility is that certain DOM elements and attributes may not be supported by all browsers. For example, Internet Explorer does not support the HTML5 `<canvas>` element, while other browsers may not support certain CSS properties or JavaScript methods. As a result, we must spend additional time and effort implementing workarounds and fallbacks to ensure that their applications function correctly on all target platforms.

Consider this example, where we want to draw a circle on a `<canvas>` element:

```
<!-- Example of a HTML5 <canvas> element -->
<canvas id="myCanvas"></canvas>
```

```
// Example of checking for support for <canvas> :
var canvas = document.getElementById("myCanvas");
```

```
if (canvas.getContext) {  
    var ctx = canvas.getContext("2d");  
    // <canvas> is supported  
} else {  
    // <canvas> is not supported  
}
```

We perform a feature detection check to see if the browser supports the `<canvas>` element. If it does, we can draw a circle on the canvas using the `arc` method. If it doesn't, we can display a message to the user indicating that their browser does not support the `<canvas>` element. Dealing with these kinds of things can be a pain, and it's easy to forget to implement these checks. This can lead to bugs and inconsistencies in web applications, which can be frustrating for users.

Furthermore, different browsers may interpret CSS and JavaScript differently, leading to inconsistencies in layout and behavior. For example, different browsers may apply different default styles to HTML elements, resulting in variations in appearance between browsers. Additionally, JavaScript methods may behave differently or have different performance characteristics on different browsers, leading to varying levels of responsiveness and interactivity.

While not directly related to React, we often have to deal with these issues when using the DOM API. For example, consider the following example, where we want to create a button element with different styles on different browsers:

```
<!-- Example of an HTML button element with different styles on different browsers -->
<button id="myButton">Click me</button>
```

```
/* Example of applying different styles to a button */
#myButton {
  color: blue;
  background-color: white;
  border: 1px solid black;
}

/* Fix for Safari */
@media screen and (-webkit-min-device-pixel-ratio: 1) {
  #myButton {
    padding: 1px 6px;
  }
}

/* Fix for Internet Explorer */
@media screen and (min-width: 0\0) {
  #myButton {
    padding: 2px 5px;
  }
}
```

To address these issues, we can take several steps to ensure cross-browser compatibility in their web applications. One approach is to use browser detection and feature detection to determine which features and capabilities are available on the user's browser and adjust the code accordingly.

```
// Example of detecting the user's browser in JavaScript
if (navigator.userAgent.indexOf("Firefox") !== -1) {
    // Code for Firefox
} else if (navigator.userAgent.indexOf("Chrome") !== -1) {
    // Code for Chrome
} else if (navigator.userAgent.indexOf("Safari") !== -1) {
    // Code for Safari
} else if (navigator.userAgent.indexOf("Trident") !== -1) {
    // Code for Internet Explorer
}
```

Additionally, we can use polyfills and shims to provide fallback functionality for unsupported features. Polyfills are JavaScript code that emulate the behavior of modern web standards on older browsers, while shims are small pieces of code that enable features not supported by a particular browser.

The virtual DOM provides a consistent, abstracted interface that is implemented the same way across all browsers and platforms, eliminating the need for browser-specific workarounds like polyfills—only around the DOM. When writing React code, you probably will have your fair share of browser-specific quirks and polyfills to deal with around other APIs your application uses, but since it abstracts away the DOM, we don't have to worry about that specific set of APIs: instead, we get to lean in to React's API and focus on the application logic we're trying to build.

Complexity

The real DOM can also be complex and difficult to work with. Modifying the DOM directly with JavaScript can be a verbose and error-prone process, especially for complex web applications. This can make it difficult to maintain and update web applications over time.

Imagine we have a web application that displays a list of blog posts. Each blog post is represented as a `div` element containing a title and some text. We want to add a button to each blog post that allows the user to expand or collapse the text. Here's how we might implement this with the real DOM:

```
<div id="blog-posts">  
  <div class="blog-post">
```

```
<h2 class="blog-post-title">Post 1</h2>
<p class="blog-post-text">
  Lorem ipsum dolor sit amet, consectetur ad
  sed enim scelerisque auctor.
</p>
<button class="expand-button">Expand</button>
</div>
<div class="blog-post">
  <h2 class="blog-post-title">Post 2</h2>
  <p class="blog-post-text">
    Morbi at neque ut metus malesuada tempus. I
    congue mauris. Donec sit amet ornare eros.
  </p>
  <button class="expand-button">Expand</button>
</div>
<!-- More blog posts... -->
</div>
```

```
const blogPosts = document.querySelectorAll(".blo

blogPosts.forEach((blogPost) => {
  const expandButton = blogPost.querySelector(".e
  const blogPostText = blogPost.querySelector(".k

  let isExpanded = false;

  expandButton.addEventListener("click", () => {
```

```
    if (isExpanded) {
        blogPostText.style.display = "none";
        expandButton.textContent = "Expand";
        isExpanded = false;
    } else {
        blogPostText.style.display = "block";
        expandButton.textContent = "Collapse";
        isExpanded = true;
    }
});
});
```

Here, we're using the `querySelectorAll` method to select all the blog posts on the page. Then, we're using the `querySelector` method to select the expand button and blog post text within each blog post. We're attaching a click event listener to the expand button that toggles the display of the blog post text and changes the text of the button between "Expand" and "Collapse".

While this code works, it can be quite verbose and error-prone. We have to manually manipulate the DOM to show or hide the text, and we have to keep track of the state of each blog post's expansion manually using the `isExpanded` variable.

Moreover, the state here is shared between the JavaScript and the HTML. This means that if we want to change the initial state of the

blog posts, we have to change both the JavaScript and the HTML. If, for some reason, the HTML is out of sync with the JavaScript, we will have problems.

If we wanted to consolidate our state of the blog into just JavaScript, we'd update the HTML to look like this:

```
<div id="blog-posts"></div>
```

```
const blogPostsContainer = document.getElementById("blog-posts");

const blogPostsData = [
  {
    title: "Post 1",
    text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
  },
  {
    title: "Post 2",
    text: "Morbi at neque ut metus malesuada tempus.",
  },
  // More blog posts...
];

blogPostsData.forEach((blogPostData) => {
  const blogPost = document.createElement("div");
  blogPost.className = "blog-post";
```

```
const blogPostTitle = document.createElement("h2");
blogPostTitle.className = "blog-post-title";
blogPostTitle.textContent = blogPostData.title;
blogPost.appendChild(blogPostTitle);

const blogPostText = document.createElement("p");
blogPostText.className = "blog-post-text";
blogPostText.textContent = blogPostData.text;
blogPost.appendChild(blogPostText);

const expandButton = document.createElement("button");
expandButton.className = "expand-button";
expandButton.textContent = "Expand";
blogPost.appendChild(expandButton);
blogPostsContainer.appendChild(blogPost);

let isExpanded = false;

expandButton.addEventListener("click", () => {
  if (isExpanded) {
    blogPostText.style.display = "none";
    expandButton.textContent = "Expand";
    isExpanded = false;
  } else {
    blogPostText.style.display = "block";
    expandButton.textContent = "Collapse";
    isExpanded = true;
  }
})
```

```
});  
});
```

In this example, we loop over an array containing a bunch of blog posts and for each object, we create a new `div` for the blog post, and append it as a child to the container `div`.

Inside each blog post `div`, we create the title and text elements using `document.createElement`, set their classes and text content appropriately, and append them as children to the blog post `div`. We also create the expand button element, set its class and text content, and append it as a child to the blog post `div`.

Finally, we add a click event listener to each expand button that toggles the display of the blog post text and changes the button text between “Expand” and “Collapse” accordingly. The state for whether the text is expanded or not is kept in a state variable `isExpanded`.

Take note that we have not even considered cleaning up the event listeners we’ve attached to the buttons! That’ll be a whole other can of worms!

This example demonstrates how, by keeping all data and state in JavaScript, we can avoid the pitfalls of shared state between HTML and JS. This is actually *very* close to React’s approach of owning the

entire DOM and all its state and protecting us further. Let's take a look at how we might implement the same functionality using React:

```
const blogPostsData = [
  {
    title: "Post 1",
    text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
  },
  {
    title: "Post 2",
    text: "Morbi at neque ut metus malesuada tempus.",
  },
  // More blog posts...
];

function BlogPost({ title, text, isExpanded }) {
  return (
    <div className="blog-post" data-title={title}>
      <h2 className="blog-post-title">{title}</h2>
      {isExpanded ? <p className="blog-post-text">{text}</p> : ""}
      <button className="expand-button">
        {isExpanded ? "Collapse" : "Expand"}
      </button>
    </div>
  );
}

function BlogPosts() {
```

```

const [expandedPosts, setExpandedPosts] = useSt

const expand = (e) => {
  const target = e.target;
  if (target.className !== "expand-button") {
    return;
  }
  const blogPost = target.parentElement;
  const title = blogPost.dataset.title;
  if (expandedPosts.includes(title)) {
    setExpandedPosts(expandedPosts.filter((t) =
  } else {
    setExpandedPosts([...expandedPosts, title]);
  }
};

return (
  <div className="blog-posts" onClick={expand}>
    {blogPostsdata.map((post) => (
      <BlogPost
        isExpanded={expandedPosts.includes(post
        title={post.title}
        text={post.text}
      />
    ))}
    {/* More blog posts */}
  </div>

```

```
    );  
  }
```

The React example above is better than dealing with the DOM directly for several reasons.

First, it abstracts away the complexity of directly manipulating the DOM, making it easier for us to focus on the application's logic instead of the presentation. We only need to write components and define their behavior, and React takes care of updating the necessary parts of the DOM when the state of the component changes.

Second, the React virtual DOM allows for efficient and performant updates to the UI. When a component's state changes, React compares the new virtual DOM with the previous one and calculates the minimal number of changes needed to update the actual DOM. This means that updates can be made quickly and without the need to recalculate the entire layout of the page.

Third, the use of JSX allows for a more declarative and readable syntax for creating UI components. Instead of creating elements and appending them to the DOM, we can write code that resembles HTML and let React handle the rendering.

Finally, we make use of event delegation to avoid having to add event listeners to each expand button. Instead, we add a single event listener to the parent element and use event bubbling to handle the click event at a higher level: we attach *one* event listener instead of n event listeners, where n is the number of expand buttons. While we can do this with the real DOM too, React's JSX brings a certain clarity that naturally leads us to think about these things and write them in an approachable way. This *may be* subjective, so take it with a grain of salt.

In summary, the complexity of the real DOM can make it difficult to maintain and update web applications over time, especially for complex applications with a lot of interactivity. React's virtual DOM makes this process much simpler and more intuitive by allowing developers to work with components and by abstracting away the complexities of the real DOM.

We just discussed event handling. Let's pull on this thread a little more to fully appreciate the complexity of working with the DOM. Imagine we have a list and we want to add an event listener to each new `<li>` element that we create so that when a user clicks on it, an alert is shown with the text of the item. With the real DOM, this would require even more code and potentially nested loops:

```
const items = ["apple", "banana", "orange"];

const list = document.createElement("ul");
document.body.appendChild(list);

for (let i = 0; i < items.length; i++) {
  const listItem = document.createElement("li");
  const itemText = document.createTextNode(items[i]);
  listItem.appendChild(itemText);
  list.appendChild(listItem);

  listItem.addEventListener("click", () => {
    alert(items[i]);
  });
}
```

In this example, we've added an event listener to each `<li>` element that displays the text of the item when clicked. However, because of the way closures work in JavaScript, the value of `i` will always be the length of `items` instead of the index of the item that was clicked. This is a common mistake that can be difficult to debug.

Now, let's contrast this with how React's virtual DOM makes this process much simpler. In React, we can create a component for the list that takes in the `items` as a prop and renders the list items as components:

```
const items = ["apple", "banana", "orange"];

function ListItem(props) {
  function handleClick() {
    alert(props.item);
  }

  return <li onClick={handleClick}>{props.item}</li>;
}

function List(props) {
  const listItems = props.items.map((item) => (
    <ListItem key={item} item={item} />
  ));

  return <ul>{listItems}</ul>;
}
```

In this example, we've created two components: `ListItem` and `List`. `ListItem` is responsible for rendering a single item and attaching an event listener to it. `List` takes in the items as a prop, maps over them to create an array of `ListItem` components, and renders them in an unordered list.

This approach is much simpler and more intuitive than manipulating the real DOM directly. Because we're working with components, we

can encapsulate the logic for each item in its own component and reuse it throughout our application. Additionally, because we're not manipulating the real DOM directly, we don't have to worry about inconsistencies between different browsers and platforms.

React, using the virtual DOM reduces complexity by having us reason about smaller, individual, and more isolated components instead of larger and more complex documents. Being so eagerly criticized in its early days for breaking separation of concerns, React actually *helps* separation of concerns this way, through components.

We've looked in detail at the real DOM and its pitfalls, as well as seen *some* of the virtual DOM's benefits by contrast. Let's zero in and laser focus on this, exploring more concretely *how* the virtual DOM works in React.

How the virtual DOM Works

The virtual DOM is a technique that helps to mitigate the pitfalls of the real DOM. By creating a virtual representation of the DOM in memory, changes can be made to the virtual representation without directly modifying the real DOM. This allows the framework or library to update the real DOM in a more efficient and performant way, without

causing the browser to do any work in recomputing the layout of the page and repainting the elements.

The virtual DOM also helps to improve cross-browser compatibility by providing a consistent API that abstracts away the differences between different browser implementations of the real DOM. This makes it easier for developers to create web applications that work seamlessly across different browsers and platforms.

React uses the virtual DOM to build user interfaces. In this section, we will explore how React's implementation of the virtual DOM works.

React Elements

In React, user interfaces are represented as a tree of React Elements. React Elements are lightweight representations of a component or HTML element. They are created using the `React.createElement` function and can be nested to create complex user interfaces.

Here is an example of a React Element:

```
const element = React.createElement(  
  "div",  
  { className: "my-class" },
```

```
    "Hello, world!"  
  );
```

This creates a React Element that represents a `<div>` element with a `className` of `my-class` and the text content of `Hello, world!`.

From here, we can see the actual created element if we `console.log(element)`. It looks like this:

```
{  
  $$typeof: Symbol(react.element),  
  type: "div",  
  key: null,  
  ref: null,  
  props: {  
    className: "my-class",  
    children: "Hello, world!"  
  },  
  _owner: null,  
  _store: {}  
}
```

This is a representation of a React element. React elements are the smallest building blocks of a React application, and they describe what should appear on the screen. Each element is a plain Java-

Script object that describes the component it represents, along with any relevant props or attributes.

The React element shown in the code block is represented as an object with several properties:

- `$$typeof` : This is a symbol used by React to ensure that an object is a valid React element. In this case, it is `Symbol(react.element)`. `$$typeof` can have other values depending on the type of the element:
 - `Symbol(react.fragment)` : When the element represents a React fragment.
 - `Symbol(react.portal)` : When the element represents a React portal.
 - `Symbol(react.profiler)` : When the element represents a React profiler.
 - `Symbol(react.provider)` : When the element represents a React context provider.
 - In general, `$$typeof` serves as a type marker that identifies the type of the React element.

We'll cover more of these later in the book.

- `type` : This property represents the type of the component that the element represents. In this case, it is `"div"`, indicating that this is a `<div>` DOM element. The `type` property of a React element can be either a string or a function. If it is a string, it rep-

resents the HTML tag name, like `"div"`, `"span"`, `"button"`, etc. When it is a function, it represents a custom React component.

Here is an example of an element with a custom component type:

```
const MyComponent = (props) => {  
  return <div>{props.text}</div>;  
};  
  
const myElement = <MyComponent text="Hello, w
```

In this case, the type property of `myElement` is `MyComponent`, which is a function that defines a custom component. The value of `myElement` as a React element object would be:

```
{  
  $$typeof: Symbol(react.element),  
  type: MyComponent,  
  key: null,  
  ref: null,  
  props: {  
    text: "Hello, world!"  
  },  
}
```

```
  _owner: null,  
  _store: {}  
}
```

Note that `type` is set to the `MyComponent` function, which is the type of the component that the element represents, and `props` contains the props passed to the component, in this case `{ text: "Hello, world!" }`.

When React encounters an element with a function type, it will invoke that function with the element's `props` and return value will be used as the element's `children`, in this case, that's a `div`. This is how custom React components are rendered: React continually goes deeper and deeper and deeper with elements until scalar values are reached, which are then rendered as text nodes, or if `null` or `undefined` is reached, nothing is rendered.

Here is an example of an element with a string type:

```
const myElement = <div>Hello, world!</div>;
```

In this case, the `type` property of `myElement` is `"div"`, which is a string that represents an HTML tag name. When React encounters an element with a string type, it will create a corre-

sponding HTML element with that tag name and render its children within that element.

- `key` : This property is used by React to identify each element in a list. If two elements have the same key, React knows that they are the same element and will not re-render them. In this case, the key is `null`.
- `ref` : This property is used to create a reference to the underlying DOM node. It is generally used in cases where direct manipulation of the DOM is necessary. In this case, the ref is `null`.
- `props` : This property is an object that contains all of the attributes and props that were passed to the component. In this case, it has two properties: `className` and `children`. `className` specifies the class name of the element, and `children` contains the content of the element.
- `_owner` : This property is used internally by React to track the component that created this element. This information is used to determine which component should be responsible for updating the element when its props or state change.

Here is an example that demonstrates how the `_owner` property is used:

```
function Parent() {  
  return <Child />;  
}
```

```
function Child() {  
  const element = <div>Hello, world!</div>;  
  console.log(element._owner); // Parent  
  return element;  
}
```

In this example, the `Child` component creates a React element representing a `<div>` element with the text `"Hello, world!"`. The `_owner` property of this element is set to the `Parent` component, which is the component that created the `Child` component.

React uses this information to determine which component should be responsible for updating the element when its props or state change. In this case, if the `Parent` component updates its state or receives new props, React will update the `Child` component and its associated element.

It's important to note that the `_owner` property is an internal implementation detail of React and should not be relied upon in application code.

- `_store`: The `_store` property of a React element object is an object that is used internally by React to store additional data about the element. The specific properties and values stored in `_store` are not part of the public API and should not be accessed directly.

Here's an example of what the `_store` property might look like:

```
{
  validation: null,
  key: null,
  originalProps: { className: 'my-class', children: '...' },
  props: { className: 'my-class', children: '...' },
  _self: null,
  _source: { fileName: 'MyComponent.js', lineNumber: 10 },
  _owner: { _currentElement: [Circular], _debugOwner: null },
  _isStatic: false,
  _warnedAboutRefsInRender: false,
}
```

As you can see, `_store` includes various properties such as `validation`, `key`, `originalProps`, `props`, `_self`, `_source`, `_owner`, `_isStatic`, and `_warnedAboutRefsInRender`. These properties are used by React internally to track various aspects of the element's state and context.

For example, `_source` is used to track the file name and line number where the element was created, which can be helpful for debugging. `_owner` is used to track the component that created the element, as discussed earlier. And `props` and `originalProps` are used to store the props passed to the component.

Again, it's important to note that `_store` is an internal implementation detail of React and should not be accessed directly in application code.

Virtual DOM vs. Real DOM

React's `createElement` function and the DOM API's `createElement` method are similar in that they both create new elements in the document. `React.createElement` is a function provided by React that creates a new virtual element in memory, whereas `document.createElement` is a method provided by the DOM API that creates a new element also in memory until it is attached to the DOM with `parent.appendChild`. Both functions take a tag name as their first argument, while `React.createElement` takes additional arguments to specify attributes and children.

For example, let's compare how we would create a simple `<div>` element using both methods:

```
// Using React's createElement
const divElement = React.createElement(
  "div",
  { className: "my-class" },
  "Hello, World!"
);
```

```
// Using the DOM API's createElement
const divElement = document.createElement("div");
divElement.className = "my-class";
divElement.textContent = "Hello, World!";
```

In both cases, we create a new `div` element with a class of `'my-class'` and some text content. However, the React version returns a virtual element that can be used in a React component, whereas the DOM API version creates a real DOM element that can be added to the document directly using `document.appendChild`.

The virtual DOM in React is similar in concept to the real DOM in that both represent a tree-like structure of elements. When a React component is rendered, React creates a new virtual DOM tree, compares it to the previous virtual DOM tree, and calculates the minimum number of changes needed to update the real DOM to match the new virtual DOM. This is known as the “reconciliation” process.

Here’s an example of how this might work in a React component:

```
function App() {
  const [count, setCount] = useState(0);

  return (
    <div>
```

```

    <h1>Count: {count}</h1>
    <button onClick={() => setCount(count + 1)}>
  </div>
);
}

```

For clarity, this component can also be expressed like so:

```

function App() {
  const [count, setCount] = React.useState(0);

  return React.createElement(
    "div",
    null,
    React.createElement("h1", null, "Count: ", count),
    React.createElement(
      "button",
      { onClick: () => setCount(count + 1) },
      "Increment"
    )
  );
}

```

In the `createElement` calls, the first argument is the name of the HTML tag or React component, the second argument is an object of

properties (or null if no properties are needed), and any additional arguments represent child elements.

When the component is first rendered, React creates a virtual DOM tree that looks like this:

```
div
├─ h1
│   └─ "Count: 0"
└─ button
    └─ "Increment"
```

When the button is clicked, React creates a new virtual DOM tree that looks like this:

```
div
├─ h1
│   └─ "Count: 1"
└─ button
    └─ "Increment"
```

React then calculates that only the text content of the `h1` element needs to be updated, and updates only that part of the real DOM.

The use of a virtual DOM in React allows for efficient updates to the real DOM, as well as allowing React to work seamlessly with other

libraries that also manipulate the DOM directly.

Efficient Updates

When a React component's state or props change, React creates a new tree of React Elements that represents the updated user interface. This new tree is then compared to the previous tree to determine the minimal set of changes required to update the real DOM using a diffing algorithm.

This algorithm compares the new tree of React Elements with the previous tree and identifies the differences between the two. It is a recursive comparison. If a node has changed, React updates the corresponding node in the real DOM. If a node has been added or removed, React adds or removes the corresponding node in the real DOM.

Diffing involves comparing the new tree with the old one node-by-node to find out which parts of the tree have changed.

React's diffing algorithm is highly optimized and aims to minimize the number of changes that need to be made to the real DOM. The algorithm works as follows:

- If the nodes at the root level of the two trees are different, React will replace the entire tree with the new one.

- If the nodes at the root level are the same, React will update the attributes of the node if they have changed.
- If the children of a node are different, React will update only the children that have changed. React does not re-render the entire subtree; it only updates the nodes that have changed.
- If the children of a node are the same, but their order has changed, React will reorder the nodes in the real DOM without actually recreating them.
- If a node has been removed from the tree, React will remove it from the real DOM.
- If a new node has been added to the tree, React will add it to the real DOM.
- If a node's type has changed (e.g., from a div to a span), React will remove the old node and create a new node of the new type.

React's diffing algorithm is pretty efficient and allows React to update the real DOM quickly and with minimal changes. This helps to improve the performance of React applications and makes it easier to build complex, dynamic user interfaces.

Batching

To further improve performance, React batches updates to the real DOM. This means that multiple updates are combined into a single

update, reducing the number of times the real DOM has to be updated.

When React is rendering a component, it doesn't immediately update the real DOM with every change. Instead, it batches the updates and performs them all at once, reducing the number of times it has to update the real DOM, further improving performance.

React uses a technique called “transaction” to batch updates. The concept of a transaction is similar to a database transaction. It groups related operations together and executes them as a single unit of work. React uses a similar approach to batch updates to the real DOM.

For example, suppose we have a component that updates its state multiple times in quick succession:

```
function Example() {  
  const [count, setCount] = useState(0);  
  
  const handleClick = () => {  
    setCount(count + 1);  
    setCount(count + 1);  
    setCount(count + 1);  
  };  
  
  return (  

```

```
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}
```

In this example, the `handleClick` function calls `setCount` three times in quick succession. Without batching, React would update the real DOM three separate times, even though the value of `count` only changed once. This would be wasteful and slow.

However, because React uses a transaction to batch updates, it only updates the real DOM once, after all three `setCount` calls have completed. This results in better performance and a smoother user experience.

The transaction API that React uses is not exposed to developers, but we can see its effects in action. To demonstrate, we can modify the previous example to log the current transaction state:

```
function Example() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    console.log(
```

```

        console.log(
            "before",
            React.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED
                .currentDispatcher
        );
        setCount(count + 1);
        setCount(count + 1);
        setCount(count + 1);
        console.log(
            "after",
            React.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED
                .currentDispatcher
        );
    };

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={handleClick}>Increment</button>
        </div>
    );
}

```

In this example, we're logging the current transaction state before and after the `setCount` calls. We're using a non-public API to access the current transaction state, so don't use this technique in production code.

If we run this example, we'll see that the before and after logs are the same:

```
before Object { ... }  
after Object { ... }
```

This indicates that React is batching the updates together into a single transaction.

React's batching behavior can sometimes be a source of confusion for developers, especially when dealing with event handlers. For example, suppose we have a component that updates its state when a button is clicked:

```
function Example() {  
  const [count, setCount] = useState(0);  
  
  const handleClick = () => {  
    setCount(count + 1);  
    console.log("count", count);  
  };  
  
  return (  
    <div>  
      <p>Count: {count}</p>  
      <button onClick={handleClick}>Increment</button>  
    </div>  
  );  
}
```

```
    );  
  }
```

In this example, we're logging the current value of `count` after the `setCount` call. However, because React batches the updates, the value of `count` logged to the console will always be one less than the current value of `count`.

When React's render function is called, it schedules a batch of updates to the real DOM, rather than immediately applying changes to the DOM. This allows React to optimize updates by batching them together and reducing the number of actual changes made to the DOM.

The batched updates work as follows: whenever a state change is made or a prop is updated in a React component, React adds the component to a "dirty" list. This list is essentially a queue of components that need to be updated. React then waits for a brief moment to allow for additional updates to be added to the queue. This is called the "debounce time."

After the debounce time has passed, React checks the dirty list and begins updating the real DOM. React first generates a new virtual DOM tree, as described earlier, and then compares it to the previous virtual DOM tree to determine what changes need to be made to the

real DOM. These changes are then batched together and applied to the real DOM all at once.

This is why the value of `count` logged to the console is always one less than the current value of `count`. When the `handleClick` function is called, it immediately logs the current value of `count` to the console before actually updating the state of the `Example` component. The state update happens later, after the debounce time has passed.

Here's another example to illustrate this batching behavior. Suppose we have a simple React component that updates its state whenever a button is clicked:

```
function Example() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  console.log("Render Example component");

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}
```

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

When the `handleClick` function is called, it updates the state of the `Example` component by calling the `setCount` function. This causes the component to be added to the dirty list.

Now, suppose we rapidly click the button multiple times. Each time we click the button, the `handleClick` function is called and adds the `Example` component to the dirty list. However, React doesn't immediately apply these changes to the real DOM. Instead, React waits for a brief moment to allow for additional updates to be added to the dirty list.

Once the debounce time has passed, React checks the dirty list and begins updating the real DOM. In this case, React generates a new virtual DOM tree for the `Example` component and compares it to the previous virtual DOM tree. React determines that the only change that needs to be made to the real DOM is to update the text of the `p` element to reflect the new count value.

React then applies this change to the real DOM all at once, resulting in a smoother and more performant user interface. By batching updates together and reducing the number of actual changes made to

the DOM, React is able to optimize performance and improve the user experience.

Fiber Architecture

React Fiber is a newer (since React 16) implementation of the React rendering engine. It is a complete rewrite of the previous rendering engine and is designed to be more efficient and flexible.

One of the main benefits of React Fiber is that it allows for incremental updates to the virtual DOM. This means that updates can be broken down into smaller chunks, allowing the browser to process user input and other events without blocking the UI.

We'll discuss this in detail in the next chapter.

Chapter Review

Throughout this chapter, we have explored the differences between the real DOM and the virtual DOM in web development, as well as the advantages of using the latter in React.

We first talked about the real DOM and its limitations, such as slow rendering times and cross-browser compatibility issues, which can make it difficult for developers to create web applications that work seamlessly across different browsers and platforms. To illustrate this, we examined how to create a simple web page using the real DOM APIs, and how these APIs can quickly become unwieldy and difficult to manage as the complexity of the page increases.

Next, we dove into the virtual DOM and how it addresses many of the limitations of the real DOM. We explored how React leverages the virtual DOM to improve performance by minimizing the number of updates needed to the real DOM, which can be expensive in terms of rendering time. We also looked at how React uses a elements to compare the virtual DOM with the previous version and calculate the most efficient way to update the real DOM.

To illustrate the benefits of the virtual DOM, we examined how to create the same simple web page using React components. We compared this approach to the real DOM approach and saw how React

components were more concise and easier to manage, even as the complexity of the page increased.

We also looked at the differences between

`React.createElement` and `document.createElement` and how React's version simplifies the creation of virtual DOM elements. We saw how we could create components using JSX, which provides a syntax similar to HTML, making it easier to reason about the structure of the virtual DOM.

Finally, we explored how React batches updates to the real DOM to improve performance. We used a lot of code examples to illustrate the concepts we discussed. We saw how to create simple web pages using the real DOM and React components, and how the latter approach can make the code more manageable and easier to maintain.

Overall, we have learned about the benefits of using the virtual DOM in web development, and how React leverages this concept to make building web applications easier and more efficient.

Review Questions

1. What is the real DOM?
2. What are some issues with the real DOM?
3. How does React use the virtual DOM to update the real DOM?
4. What are the benefits of using the virtual DOM?

5. How does React batch updates to the real DOM?

Up Next

In the next chapter, we will dive deep into React reconciliation and its Fiber architecture.

Chapter 4. Inside Reconciliation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

To be truly fluent in React, we need to understand *what its functions do*. So far, we’ve understood JSX and `React.createElement`. We’ve also understood the virtual DOM. Let’s explore the practical applications of it in React in this chapter, and understand what `ReactDOM.createRoot(element).render` does. Specifically, we’ll explore *how* React builds and interacts with its virtual DOM through a process called reconciliation.

Understanding Reconciliation

As a quick recap, React's virtual DOM is a blueprint of our desired UI state. React takes this blueprint and, through a process called **reconciliation**, makes it a reality in a given host environment; usually a web browser.

Consider the following code snippet,

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

const App = () => {
  const [count, setCount] = useState(0);

  return (
    <main>
      <div>
        <h1>Hello, world!</h1>
        <span>Count: {count}</span>
        <button onClick={() => setCount(count + 1)}>Increment</button>
      </div>
    </main>
  );
};
```

This code snippet contains a declarative description of what we want our UI state to be: a tree of elements. Both our teammates *and* React can read this and understand we're trying to create a counter app with an increment button that increments the counter. To understand reconciliation, let's understand what React does on the inside when faced with a component like this.

First, the JSX becomes a tree of React elements. This is what we saw in the last chapter. The `App` component is a React element, and so are its children. React elements are immutable and represent the desired state of the UI. They are not the actual UI state. React elements are created by `React.createElement` or JSX, so this would be transpiled into:

```
const App = () => {
  const [count, setCount] = useState(0);

  return React.createElement(
    "main",
    null,
    React.createElement(
      "div",
      null,
      React.createElement("h1", null, "Hello, world"),
      React.createElement("span", null, "Count: " + count),
      React.createElement(
        "button",
        null,
        "Increment"
      )
    )
  )
}
```

```

        "button",
        { onClick: () => setCount(count + 1) },
        "Increment"
      )
    )
  );
};

```

This would give us a tree of created React elements that looks something like this:

```

{
  type: "main",
  props: {
    children: {
      type: "div",
      props: {
        children: [
          {
            type: "h1",
            props: {
              children: "Hello, world!",
            },
          },
          {
            type: "span",
            props: {

```

```
        children: ["Count: ", count],
      },
    },
    {
      type: "button",
      props: {
        onClick: () => setCount(count + 1),
        children: "Increment",
      },
    },
  ],
},
},
}
```

Oh look, we have a virtual DOM. Since this is the first render, this tree is now committed to the browser using minimal calls to imperative DOM APIs. This is what we saw in the last chapter. Now if an update happens, React will create a new tree with updated values. This tree will need to be *reconciled* with what we've already committed to the browser. This is where reconciliation comes in.

Before we understand what modern-day React does under the hood, let's explore how React used to perform reconciliation before version

16, with a legacy “stack” reconciler. This will help us understand the need for today’s popular fiber reconciler.

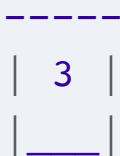
Prior Art

Previously, React used a stack data structure for rendering. To make sure we’re on the same page, let’s briefly discuss the stack data structure.

Stack Data Structure

In computer science, a stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. A stack has two fundamental operations—push and pop—that allow elements to be added and removed from the top of the stack, respectively.

A stack can be visualized as a collection of elements that are arranged vertically, with the topmost element being the most recently added one. Here’s an ASCII illustration of a stack with three elements:



```
  -----  
  | 3  |  
  |____|
```

The diagram shows a stack with three elements. The top element is the number 3, which is colored purple. The stack is represented by a vertical container with a dashed line at the top and a solid line at the bottom. The number 3 is centered within the container.



In this example, the most recently added element is 3, which is at the top of the stack. The element 1, which was added first, is at the bottom of the stack.

Stack Operations

As mentioned earlier, a stack has two fundamental operations—push and pop—that allow elements to be added and removed from the top of the stack, respectively.

The push operation adds an element to the top of the stack. In code, this can be implemented using an array and the `push` method, like this:

```
const stack = [];  
  
stack.push(1); // stack is now [1]  
stack.push(2); // stack is now [1, 2]  
stack.push(3); // stack is now [1, 2, 3]
```

The pop operation removes the top element from the stack. In code, this can be implemented using an array and the `pop` method, like this:

```
const stack = [1, 2, 3];  
  
const top = stack.pop(); // top is now 3, and stack is [1, 2]
```

In this example, the `pop` method removes the top element (3) from the stack and returns it. The stack array now contains the remaining elements (1 and 2).

Common Uses of Stacks

Stacks are a fundamental data structure used in many different areas of computer science and programming. Here are some common uses of stacks:

Function Call Stack

In programming languages, functions are executed using a call stack. When a function is called, its arguments and local variables are pushed onto the call stack. When the function returns, its arguments and local variables are popped off the stack.

Here's an example of a function that uses a stack to keep track of the call stack:

```
function foo() {  
  console.log("foo");  
  bar();  
}  
  
function bar() {  
  console.log("bar");  
}  
  
foo();
```

When this code is executed, the output will be:

```
bar  
foo
```

This happens because the `foo` function calls `bar`, which is added to the top of the call stack. When `bar` returns, it is popped off the stack, and control is returned to `foo`.

Expression Evaluation

Stacks are often used in compilers and interpreters to evaluate expressions. In this context, a stack can be used to keep track of operands and operators in an expression.

Here's an example of how a stack can be used to evaluate a simple expression:

```
const expression = "2 + 3 * 4";
const operands = [];
const operators = [];

for (const token of expression.split(" ")) {
  if (!isNaN(token)) {
    operands.push(parseFloat(token));
  } else if (token === "+" || token === "-") {
    while (operators.length > 0) {

      const op = operators[operators.length - 1];
      if (op === "+" || op === "-" || op === "*") {
        const b = operands.pop();
        const a = operands.pop();
        const result = evaluate(a, b, operators.pop());
        operands.push(result);
      } else {
        break;
      }
    }
    operators.push(token);
  }
}
```

```
    } else if (token === "*" || token === "/") {
      while (operators.length > 0) {
        const op = operators[operators.length - 1];
        if (op === "*" || op === "/") {
          const b = operands.pop();
          const a = operands.pop();
          const result = evaluate(a, b, operators.pop());
          operands.push(result);
        } else {
          break;
        }
      }
      operators.push(token);
    }
  }

  while (operators.length > 0) {
    const b = operands.pop();
    const a = operands.pop();
    const result = evaluate(a, b, operators.pop());
    operands.push(result);
  }

  console.log(operands.pop()); // output: 14
```

In this example, the expression `"2 + 3 * 4"` is split into tokens, which are then evaluated using a stack. The operands are pushed onto the `operands` stack, while the operators are pushed onto the `operators` stack. The `evaluate()` function is called whenever an operator is encountered, and it pops the top two operands from the `operands` stack, applies the operator, and pushes the result back onto the stack.

At the end of the evaluation, the final result is popped off the `operands` stack and printed to the console.

Anyway, without digressing too much, a stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It has two fundamental operations—push and pop—that allow elements to be added and removed from the top of the stack, respectively. Stacks are commonly used in computer science and programming for a variety of purposes, such as function call stacks, expression evaluation, and undo/redo operations.

When implementing a stack, it's important to choose an appropriate data structure. In JavaScript, an array is often used as the underlying data structure for a stack because it already provides the necessary operations (push and pop) and has good performance characteristics.

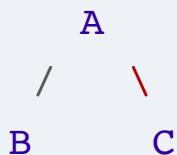
The Stack in React Reconciliation

React's original reconciler was a stack-based algorithm that was used to compare the old and new virtual trees and update the DOM accordingly. While the stack reconciler worked well in simple cases, it presented a number of challenges as applications grew in size and complexity. In this section, we'll discuss some of the challenges that the stack reconciler presented and how they were addressed with the new fiber reconciler.

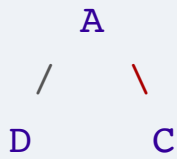
Before we dive into the challenges of the stack reconciler, let's take a quick look at how it worked. The stack reconciler used a depth-first search algorithm to traverse the virtual tree and update the DOM. When a component was updated, React would create a new virtual tree and compare it to the previous tree using a diffing algorithm. The stack reconciler would then traverse the two trees and update the DOM as necessary.

Here's a simple example of how the stack reconciler works:

Original virtual tree:



Updated virtual tree:



Stack reconciler operations:

- Update A
- Create D
- Append D to A's children
- Remove B

In this example, the stack reconciler updates component A and replaces its child B with the new child D. It then updates component C as usual. Here's a high-level summary of the problems the stack reconciler presented:

Consider an example wherein you've got a list of updates to make:

1. A non-essential computationally expensive component consumes CPU and renders
2. A user types a text update in an `input`
3. `Button` becomes enabled if the input is valid
4. A containing `Form` component holds the state, so it rerenders

In this case, the stack reconciler would render the updates sequentially without being able to pause or defer work. If the computationally expensive component from step 1 blocks rendering, user input will appear on screen with an observable lag. This leads to poor user experience. Instead, it would be far more pleasant to be able to recognize the user input from step 2 as a higher-priority update than step 1 and update the screen to reflect the input, deferring rendering step 1's computationally expensive component.

There is a need to be able to:

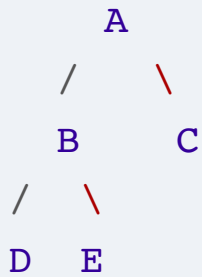
1. Bail out of current rendering work if interrupted by higher priority rendering work, and
2. Assign priority to certain types of rendering work over others

Of course, this is a high-level overview of the problem. There are many other challenges that the stack reconciler presented. Let's get a bit deeper into the details.

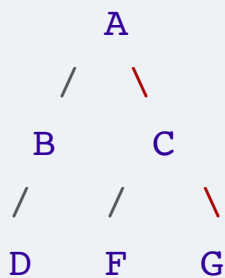
The stack reconciler had poor performance characteristics for large virtual trees. Because it used a depth-first search algorithm, it would traverse the entire virtual tree before updating the DOM. This meant that even small changes to the virtual tree could result in a large number of DOM updates.

Here's an example of how the stack reconciler could result in many unnecessary DOM updates:

Original virtual tree:



Updated virtual tree:



Stack reconciler operations:

- Update A
- Remove E
- Create F
- Append F to C's children
- Create G
- Append G to C's children

In this example, the only change to the virtual tree is the removal of node E and the addition of node F and G. However, the stack reconciler still traverses the entire virtual tree and updates the DOM accordingly.

Synchronous Execution

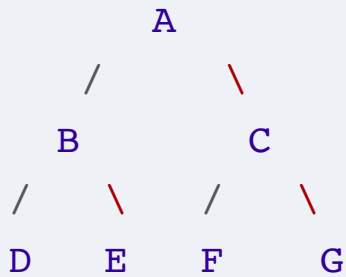
The stack reconciler executed synchronously and blocked the main thread, which could result in jank or unresponsive user interfaces. Because the reconciler would traverse the entire virtual tree before updating the DOM, any changes to the virtual tree would be delayed until the reconciler had finished its work.

In JavaScript, all code is executed on a single thread, which is called the main thread. This means that any long-running tasks (such as rendering a large virtual tree) will block the thread and prevent other tasks (such as handling user input) from executing until the long-running task has finished.

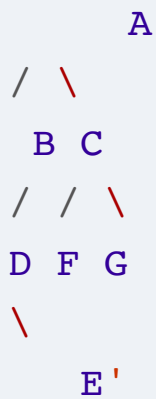
In React, the stack reconciler executed synchronously on the main thread, which meant that any updates to the virtual tree would block the thread and prevent other tasks (such as handling user input) from executing until the update had finished. This could result in a janky or unresponsive user interface, especially on slower devices or when dealing with large virtual trees.

Here's an example of how synchronous execution could affect the performance of a React application:

Original virtual tree:



Updated virtual tree:



Stack reconciler operations:

- Update A
- Remove E
- Remove C
- Create F
- Append F to C's children

- Create G
- Append G to C's children
- Create E'
- Append E' to B's children

In this example, we're updating the virtual tree by removing node E and adding node E' as a child of node B. However, because the stack reconciler executes synchronously on the main thread, any updates to the virtual tree will block other tasks from executing until the update has finished, like user input. You could very easily feel a delayed response from a UI element, such as a button, while the stack reconciler is updating the virtual tree.

Lack of Prioritization

The stack reconciler did not prioritize updates, which meant that less important updates could block more important updates. For example, a low-priority update to a tooltip might block a high-priority update to a text input. Updates to the virtual tree were executed in the order they were received.

In a React application, updates to the virtual tree can have different levels of importance. For example, an update to a form input might be more important than an update to a tooltip, because the user is directly interacting with the input and expects it to be responsive.

In the stack reconciler, updates were executed in the order they were received, which meant that less important updates could block more important updates. For example, if a tooltip update was received before a form input update, the tooltip update would be executed first and could block the form input update.

Here's an example of how update prioritization could affect the performance of a React application:

Original virtual tree:

```
      A
     / \
    B  C
   / \ / \
  D  E F  G
```

Updated virtual tree:

```
      A
     / \
    B  C
   / /  \
  D F  G
   \
    
```

E'

Stack reconciler operations:

- Update A
- Remove E
- Remove C
- Create F
- Append F to C's children
- Create G
- Append G to C's children
- Create E'
- Append E' to B's children
- Update tooltip

In this example, we're updating the virtual tree by removing node E and adding node E' as a child of node B. However, before this update can be executed, a tooltip update is received and executed first. This means that the tooltip update blocks the update to the virtual tree, even though it's less important.

If the tooltip update takes a long time to execute (e.g. because it's performing an expensive computation), this could result in a noticeable delay or jank in the user interface, especially if the user is interacting with the application during the update.

Lack of Interruptibility

Another challenge with the stack reconciler was that it did not allow updates to be interrupted or cancelled. This meant that updates that were no longer needed (e.g. because the user had navigated to a different page) would still be executed. In this section, we'll explore this challenge in more detail and look at how it could affect the performance of a React application.

In a React application, updates to the virtual tree can become unnecessary if the user navigates to a different page or otherwise interacts with the application in a way that renders the update irrelevant. For example, if the user navigates to a different page while a tooltip update is being executed, the tooltip update is no longer necessary and should be cancelled.

In the stack reconciler, updates could not be interrupted or cancelled, which meant that unnecessary updates could still be executed. This could result in unnecessary work being performed on the virtual tree and the DOM, which could negatively impact the performance of the application.

Here's an example of how unnecessary updates could affect the performance of a React application:

```
Original virtual tree:
```

```

      A
    /  \
   B    C
  / \  / \
 D  E F  G

```

Updated virtual tree:

```

      A
    /  \
   B    C
  /  /  \
 D  F  G
  \
   E'

```

Stack reconciler operations:

- Update A
- Remove E
- Remove C
- Create F
- Append F to C's children
- Create G
- Append G to C's children
- Create E'

- Append E' to B's children
- Update tooltip

In this example, we're updating the virtual tree by removing node E and adding node E' as a child of node B. However, before this update can be executed, a tooltip update is received and executed first. If the user navigates to a different page while the tooltip update is being executed, the tooltip update is no longer necessary and should be cancelled. However, because the stack reconciler does not allow updates to be interrupted or cancelled, the tooltip update will still be executed.

In summary, the stack reconciler presented a number of challenges as applications grew in size and complexity. The main challenges were poor performance, synchronous execution, lack of prioritization, and lack of interruptibility. To address these challenges, the React team developed a new reconciler called the fiber reconciler, which is based on a different data structure called a fiber tree. Let's explore this data structure in the next section.

The Fiber Reconciler

The fiber Reconciler involves the use of a different data structure and takes inspiration from “double buffering” in the game world. Let's un-

derstand both of these concepts before we move further.

Double Buffering

Double buffering is a technique used in computer graphics and video processing to reduce flicker and improve performance. The technique involves creating two buffers (or memory spaces) for storing images or frames, and switching between them at regular intervals to display the final image or video.

Here's how double buffering works in practice:

1. The first buffer is filled with the initial image or frame.
2. While the first buffer is being displayed, the second buffer is updated with new data or images.
3. When the second buffer is ready, it is switched with the first buffer and displayed on the screen.
4. The process continues, with the first and second buffers being switched at regular intervals to display the final image or video.

By using double buffering, flicker and other visual artifacts can be reduced, since the final image or video is displayed without interruptions or delays.

Double buffering is similar to Fiber reconciliation: the fiber reconciler uses double buffering to update the virtual DOM in a way that is both

efficient and user-friendly.

In the fiber reconciler, the virtual DOM is updated asynchronously in a series of small, incremental steps. The fiber reconciler uses a work loop to process updates, and each iteration of the work loop updates a small portion of the virtual DOM.

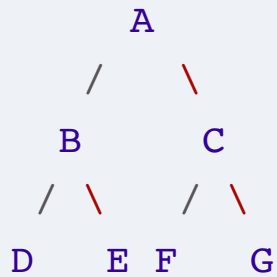
As each portion of the virtual DOM is updated, the fiber reconciler creates a new Fiber node to represent the updated component. However, instead of immediately updating the real DOM with the new Fiber node, the fiber reconciler stores the Fiber node in a “work in progress tree”.

The work in progress tree is a copy of the real DOM that is used for rendering the updated content. By using a work in progress tree, the fiber reconciler can avoid making unnecessary updates to the real DOM, which can improve performance and reduce flicker.

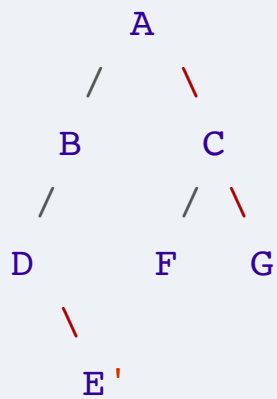
Once the work loop is complete, the fiber reconciler switches the real DOM with the work in progress tree, using double buffering to ensure that the update is smooth and seamless.

Here’s an example of how double buffering works in React Fiber:

Original virtual DOM:



Updated virtual DOM:



Fiber reconciler operations:

- Create Fiber node for A
- Create Fiber node for B
- Create Fiber node for C
- Create Fiber node for D
- Create Fiber node for E
- Update Fiber node for A
- Update Fiber node for B
- Create Fiber node for F
- Append F to C's children
- Create Fiber node for G

- Append G to C's children
- Create Fiber node for E'
- Append E' to B's children
- Switch real DOM with work in progress tree

In this example, the fiber reconciler updates the virtual DOM by creating a new Fiber node for each component and updating the relevant Fiber nodes with the new content. Instead of immediately updating the real DOM with the new Fiber nodes, the fiber reconciler stores the Fiber nodes in a work in progress tree. Once the update is complete, the fiber reconciler switches the real DOM with the work in progress tree.

This approach has several advantages:

- **Improves performance:** The fiber reconciler can update the virtual DOM incrementally, without making unnecessary updates to the real DOM. This can improve performance and reduce the risk of visual artifacts or flicker.
- **Provides a smooth user experience:** The fiber reconciler can update the virtual DOM in a way that is seamless and smooth, without interrupting the user experience.
- **Enables asynchronous updates:** By updating the virtual DOM asynchronously, the fiber reconciler can prioritize updates and ensure that less important updates don't block more important

ones. This can improve the responsiveness of React applications and make them more user-friendly.

There are also some limitations with this approach:

- **Complexity:** This can be complex to implement, particularly in complex React applications with many components and states. This can make it difficult for developers to debug issues and optimize performance.
- **Requires more memory:** Having two trees requires more memory than traditional rendering techniques, since it involves storing two copies of the virtual DOM. This can be a concern for applications with limited memory resources.
- **Not suitable for all applications:** Having two trees may not be suitable for all React applications, particularly those that require real-time updates or low latency. In these cases, other rendering techniques may be more appropriate.

React Fiber draws inspiration from double buffering to update the virtual DOM incrementally and provide a smooth, responsive user experience. While double buffering has several advantages, it also has some limitations, particularly in complex React applications. By understanding the advantages and limitations of double buffering, developers can make informed decisions about how to optimize performance and improve the user experience of their React applications.

Fiber as a Data Structure

The Fiber data structure in React is a key component of the fiber reconciler. The fiber reconciler allows updates to be prioritized and executed asynchronously, which improves the performance and responsiveness of React applications. Let's explore the Fiber data structure in more detail and look at how it works.

At its core, the Fiber data structure is a lightweight representation of a component and its state in a React application. The Fiber data structure is designed to be mutable and can be updated and rearranged as needed during the reconciliation process.

Each Fiber node contains information about the component it represents, including its props, state, and child components. The Fiber node also contains information about its position in the component tree, as well as metadata that is used by the fiber reconciler to prioritize and execute updates.

Here's an example of a simple Fiber node:

```
{  
  tag: 3, // 3 = ClassComponent  
  type: App,  
  key: null,  
  ref: null,
```

```

    props: {
      name: "Tejas",
      age: 30
    },
    stateNode: AppInstance,
    return: FiberParent,
    child: FiberChild,
    sibling: FiberSibling,
    index: 0,
    ...
  }

```

In this example, we have a Fiber node that represents a `ClassComponent` called `App`. The Fiber node contains information about the component's:

- `tag ( 3 = ClassComponent )`
- `type ( App )`
- `props ( {name: "John", age: 32} )`
- `stateNode` (an instance of the App component)
- and its position in the component tree (`return`, `child`, `sibling`, and `index`).

The Fiber data structure works by breaking up the work of updating the virtual DOM into smaller, more manageable chunks that can be executed asynchronously on the main thread. This allows updates to

be prioritized and executed based on their importance, which improves the performance and responsiveness of React applications.

Fiber reconciliation involves comparing the previous virtual DOM (the “old” tree) with the current virtual DOM (the “new” tree) and figuring out which nodes need to be updated, added, or removed.

During the reconciliation process, the fiber reconciler creates a Fiber node for each element in the virtual DOM. There is literally a function called `createFiberFromTagAndProps` that does this. Of course, another way of saying “tag and props” is by calling them React elements. As we recall, a React element is this:

```
{
  type: "div",
  props: {
    className: "container"
  }
}
```

Type (tag) and props. This function returns a fiber derived from elements. The Fiber node contains information about the component’s props, state, and child components, as well as metadata that is used by the fiber reconciler to prioritize and execute updates.

Once the Fiber nodes have been created, the fiber reconciler uses a “work loop” to update the virtual DOM. The work loop starts at the root Fiber node and works its way down the component tree, marking each Fiber node as “dirty” if it needs to be updated. Once it reaches the end, it walks back up, creating a new DOM tree in memory, detached from the browser, that will eventually be committed (flushed) to the screen.

At each step in the work loop, the fiber reconciler checks the priority of the update and decides whether to continue executing or yield control to the browser. This allows updates to be prioritized based on their importance, and less important updates can be delayed or skipped if necessary.

With the fiber reconciler, two trees are derived from a user-defined tree of JSX elements: one tree containing “current” fibers, and another tree containing “work in progress” fibers. Now that we get what fibers are, let’s look at how they work in trees.

Fiber Trees

React’s Fiber Reconciliation is made possible by two trees: the “current” tree and the “work in progress” tree. These trees are derived from a user-defined tree of JSX elements. Let’s look at how these trees work.

The Work in Progress Tree

React's "work in progress" fiber tree is a key component of the fiber reconciler in React. The work in progress fiber tree is a copy of the real DOM that is used for rendering the updated content. The fiber reconciler uses the work in progress tree to avoid making unnecessary updates to the real DOM, which can improve performance and reduce flicker.

When an update is triggered in a React application, the fiber reconciler creates a new work in progress tree that reflects the updated content. The fiber reconciler then uses a work loop to process the update, creating new Fiber nodes and updating existing ones as needed.

As each portion of the virtual DOM is updated, the fiber reconciler stores the new Fiber nodes in the work in progress tree, rather than immediately updating the real DOM. This allows the fiber reconciler to avoid making unnecessary updates to the real DOM, which can improve performance and reduce flicker.

It can also throw away an existing work in progress tree at any time if a higher priority update comes in because the work in progress tree is never committed to the real DOM until it's ready. This allows the fiber

reconciler to prioritize updates based on their importance, which improves the performance and responsiveness of React applications.

Once the update is complete, the fiber reconciler switches the real DOM with the work in progress tree, to ensure that the update is smooth and seamless.

The Current Tree

Here's an example of how the work in progress tree relates to the current tree:

```
// Current virtual DOM
<div>
  <h1>Hello, world!</h1>
  <p>Current count: 0</p>
  <button>Increment</button>
</div>

// Work in progress virtual DOM
<div>
  <h1>Hello, world!</h1>
  <p>Current count: 1</p>
  <button>Increment</button>
</div>
```

In this example, we have a current virtual DOM that contains a heading, a paragraph, and a button. When an update is triggered to increment the count, the fiber reconciler creates a new work in progress virtual DOM that reflects the updated content.

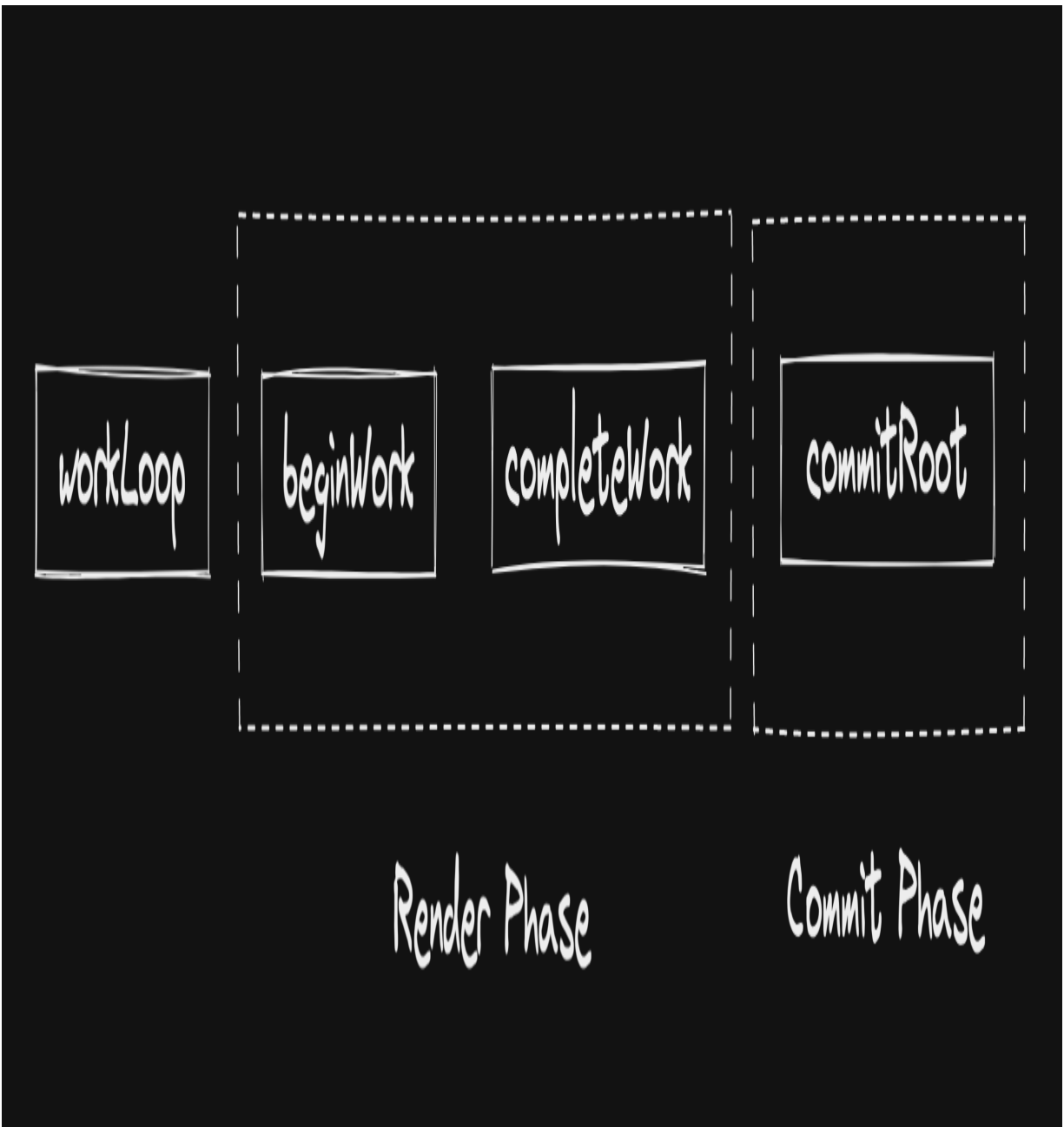
The work in progress virtual DOM contains the same heading and button as the current virtual DOM, but the paragraph has been updated to reflect the new count. Once the update is complete, the fiber reconciler switches the current virtual DOM with the work in progress virtual DOM.

Let's look at the current tree and its role in Fiber reconciliation.

Fiber Reconciliation

Fiber reconciliation happens in two phases:

1. The render phase, and
2. The commit phase

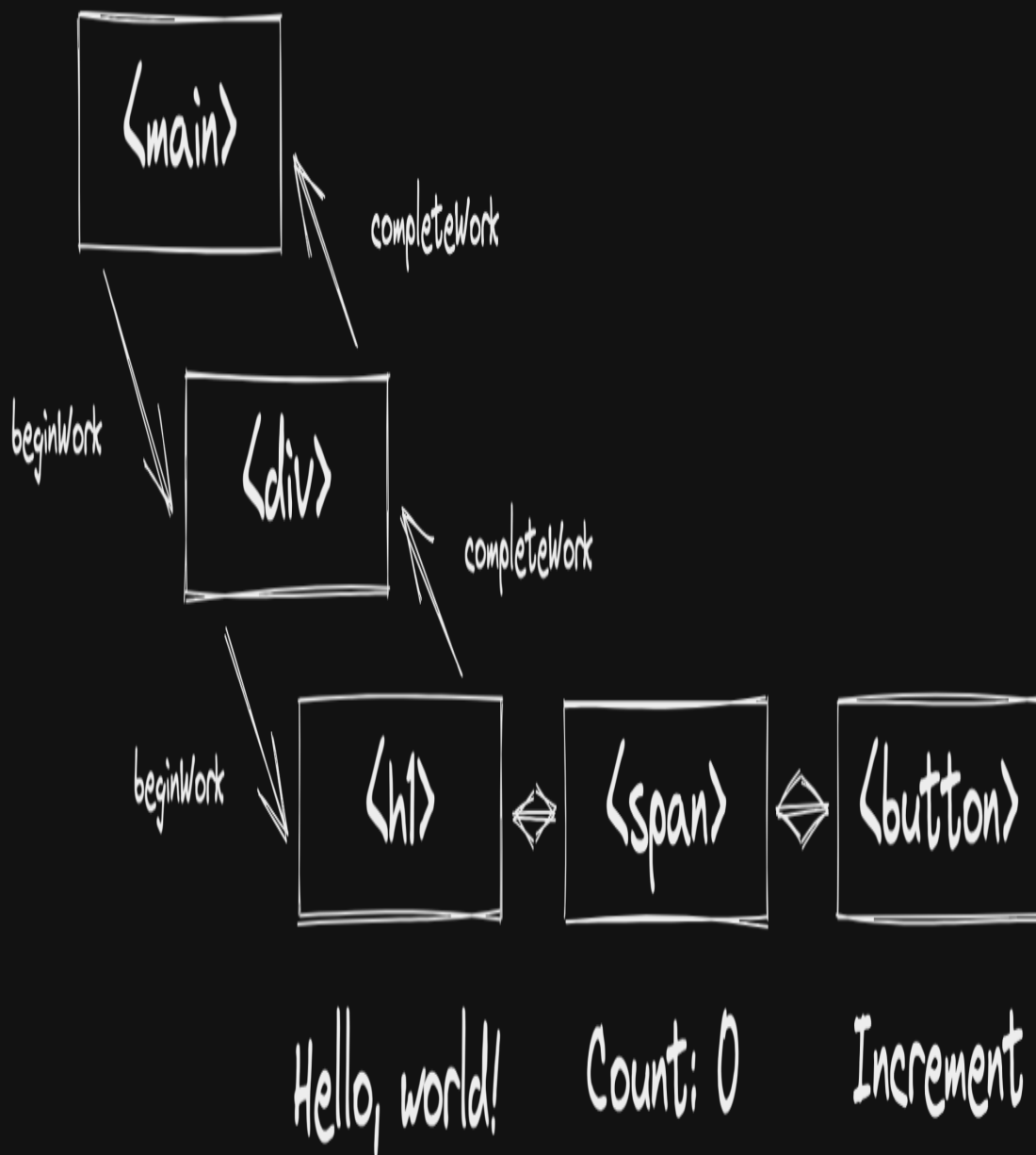


This two phase approach allows React to do rendering work that can be disposed of n-times before committing it to the DOM and showing a new state to users: it makes rendering interruptible.

Let's walk through these phases of reconciliation.

The Render Phase

The render phase starts when a state-change event occurs in the `current` tree, React does the work of making the changes *off-screen* in the `alternate` tree by recursively stepping through each fiber and setting flags that signal updates are pending. This happens in a function called `beginWork` internally in React.



beginWork

```
function beginWork(  
  current: Fiber | null,  
  workInProgress: Fiber,  
  renderLanes: Lanes  
): Fiber | null;
```

`beginWork` is a key function in the fiber reconciler of React, responsible for setting flags on Fiber nodes in the work in progress tree about whether or not they should update. It sets a bunch of flags, and then recursively goes to the next Fiber node doing the same thing until it reaches the bottom of the tree. When it finishes, we start calling `completeWork` on the Fiber nodes and walk back up.

More on `completeWork` later. For now, let's dive into `beginWork`. Its signature includes the following arguments:

- `current` : A reference to the Fiber node in the current tree that corresponds to the work in progress node being updated. This is used to determine what has changed between the previous version and the new version of the tree, and what needs to be updated in the real DOM. This is *never* mutated, and is only used for comparison.
-

- `workInProgress` : The Fiber node being updated in the work in progress tree. This is the node that will be marked as “dirty” if updated and returned by the function.
- `renderLanes` : Render lanes is a new concept in React’s fiber reconciler that replaces the older `renderExpirationTime` . It’s a bit more complex than the old `renderExpirationTime` concept, but it allows React to better prioritize updates and make the update process more efficient.

It is essentially a bitmask that represents the lanes at which an update is being processed. Lanes are a way of categorizing updates based on their priority and other factors. When a change is made to a React component, it is assigned a lane based on its priority and other characteristics. The higher the priority of the change, the higher the lane it is assigned to.

The `renderLanes` value is passed to the `beginWork` function in order to ensure that updates are processed in the correct order. Updates that are assigned to higher-priority lanes are processed before updates that are assigned to lower-priority lanes. This ensures that high-priority updates, such as updates that affect user interaction or accessibility, are processed as quickly as possible.

In addition to prioritizing updates, `renderLanes` also helps React to better manage asynchrony. React uses a technique called “time slicing” to break up long-running updates into smaller, more

manageable chunks. `renderLanes` plays a key role in this process, as it allows React to determine which updates should be processed first, and which updates can be deferred until later. Here's an example of how `renderLanes` might be used in the fiber reconciler. This is 100% pseudocode and not actual code from React, but is intended to illustrate the concept:

```
function updateComponent(element: ReactElement) {
  // Create a new Fiber node for the element
  const newFiber = createFiberFromElement(element)

  // Reconcile the new Fiber node with the current one
  const [rootFiber] = render(newFiber, document)

  // Begin the work loop to update the work in progress
  let nextUnitOfWork = rootFiber;
  let lanes = NoLanes; // Start with no lanes
  while (nextUnitOfWork !== null) {
    // Perform the next step in the reconciliation
    nextUnitOfWork = beginWork(nextUnitOfWork, lanes)

    // If there is no next unit of work, we have reached the end
    if (nextUnitOfWork === null) {
      // Commit the updated work in progress
      commitRoot(rootFiber);
    }
  }
}
```

```
    // Update the lanes based on the work we
    lanes = getNextLanes(rootFiber, lanes);
  }
}
```

In this example, we have a function called `updateComponent` that takes an element and updates it in the virtual DOM. The function creates a new Fiber node for the element using the `createFiberFromElement` function, and reconciles it with the current tree using the render function.

The function then begins the work loop to update the work in progress tree, using the updated `beginWork` function to perform the next step in the reconciliation process. The lanes variable is initially set to `NoLanes`, which means that no updates have been assigned to any lanes yet.

After each call to `beginWork`, the `getNextLanes` function is called to update the lanes variable based on the work that was just performed. This ensures that updates are processed in the correct order, and that high-priority updates are processed first. Overall, `renderLanes` is an important new concept in React's fiber reconciler that helps to make the update process more efficient and reliable. By allowing React to prioritize updates and manage asynchrony more effectively, `renderLanes` helps to

ensure that React applications are fast and responsive, even as they become more complex and handle larger amounts of data. In addition to helping with priority and asynchrony management, `renderLanes` also allows React to better handle “work in progress” updates. As we know, in the fiber reconciler, updates are processed in two phases: the “render” phase and the “commit” phase. During the render phase, updates are performed on the work in progress tree, while during the commit phase, updates are applied to the actual DOM.

`renderLanes` plays a key role in this two-phase process, as it helps React to determine when to apply updates to the work in progress tree and when to apply updates to the actual DOM. Updates that are assigned to lower-priority lanes are deferred until later, which means that they may not be applied until just before the commit phase.

Here’s an example of how `renderLanes` might be used to handle work in progress updates:

```
function updateComponent(element: ReactElement) {
  // Create a new Fiber node for the element
  const newFiber = createFiberFromElement(element)

  // Reconcile the new Fiber node with the current tree
  const [rootFiber] = render(newFiber, document)

  // Begin the work loop to update the work in progress tree
```

```

let nextUnitOfWork = rootFiber;
let lanes = NoLanes; // Start with no lanes
while (nextUnitOfWork !== null) {
  // Perform the next step in the reconciling
  nextUnitOfWork = beginWork(nextUnitOfWork, lanes, time);

  // If there is no next unit of work, we have finished
  if (nextUnitOfWork === null) {
    // Commit the updated work in progress
    commitRoot(rootFiber);
  }

  // Update the lanes based on the work we did
  lanes = getNextLanes(rootFiber, lanes);
}

// After the work loop is finished, we need to
// to handle any deferred updates that may have
// the render phase.
lanes = getLanesToRetrySynchronouslyOnError(lanes);
if (lanes !== NoLanes) {
  // Start a new work loop to handle the deferred
  nextUnitOfWork = rootFiber;
  while (nextUnitOfWork !== null) {
    nextUnitOfWork = beginWork(nextUnitOfWork, lanes, time);
    if (nextUnitOfWork === null) {
      commitRoot(rootFiber);
    }
  }
}

```

```
        lanes = getNextLanes(rootFiber, lanes);  
    }  
}  
}
```

In this example, the `updateComponent` function performs a two-phase update on the work in progress tree. During the render phase, updates are processed using the `beginWork` function, and `renderLanes` is used to prioritize updates and handle asynchrony.

After the render phase is complete, the `getLanesToRetrySynchronouslyOnError` function is called to determine if any deferred updates were created during the render phase. If there are deferred updates, the `updateComponent` function starts a new work loop to handle them, using `beginWork` and `getNextLanes` to process the updates and prioritize them based on their lanes.

`renderLanes` is a key concept in React's fiber reconciler that helps to make the update process more efficient, reliable, and responsive. By allowing React to better prioritize updates, manage asynchrony, and handle work in progress updates, `renderLanes` plays a crucial role in ensuring that React applications are fast and responsive, even as they become mo

`completeWork`

The `completeWork` function is a critical part of React's fiber reconciler, and it plays a key role in updating the work in progress tree. When called, it applies the updates to the fiber node and constructs a new real DOM tree that represents the updated state of the application in a detached way, fully in memory and invisible to the browser.

If the host environment is a browser, this means doing things like `document.createElement` or `newElement.appendChild`. Keep in mind, this tree of elements is not yet attached to the in-browser document: React is just creating the next version of the UI off-screen. Doing this work off-screen makes it interruptable: whatever next state React is computing is not yet painted to the screen, so it can be thrown away in case some higher priority update becomes scheduled. This is the whole point of the Fiber reconciler.

The signature of `completeWork` is as follows:

```
function completeWork(  
  current: Fiber | null,  
  workInProgress: Fiber,  
  renderLanes: Lanes  
): Fiber | null;
```

Here, `current` is a reference to the current fiber node in the tree, `workInProgress` is the fiber node being worked on, and

`renderLanes` is an integer value that represents the priority level of the update being completed.

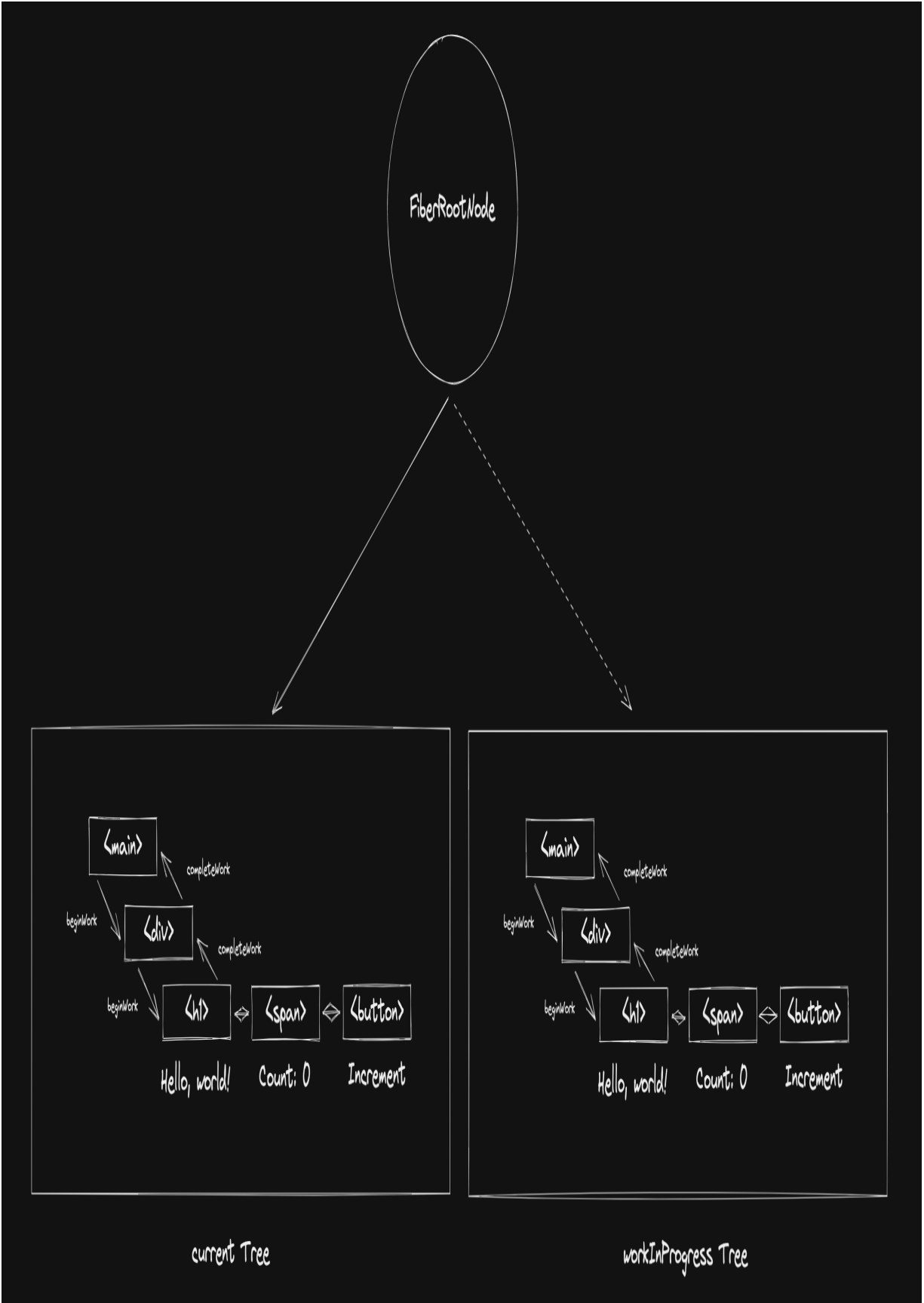
It is the same signature as `beginWork`.

The `completeWork` function is closely related to the `beginWork` function. While `beginWork` is responsible for setting flags about “should update” state on a fiber node, `completeWork` is responsible for constructing a new tree to be committed to the host environment.

When `completeWork` reaches the top and has constructed the new DOM tree, we say **the render phase is completed**. Now, React moves on to the **commit phase**.

`completeWork` prepares for the commit phase by returning the next fiber node to be processed. Once all fiber nodes in the work in progress tree have been processed by `completeWork`, the commit phase can begin. During the commit phase, the new virtual DOM tree is committed to the host environment, and the work in progress tree is replaced with the current tree.

The Commit Phase



The commit phase is responsible for updating the actual DOM with the changes that were made to the virtual DOM during the render phase. The commit phase is divided into two parts: the mutation phase and the layout phase.

The Mutation Phase

The mutation phase is the first part of the commit phase, and it is responsible for updating the actual DOM with the changes that were made to the virtual DOM. During this phase, React walks through the fiber tree, looking for nodes that have been marked as “dirty” (i.e., nodes that have been updated).

For each dirty node, React calls a special function called `commitMutationEffects`. This function applies the updates that were made to the node during the render phase to the actual DOM.

Here’s an full-pseudocode example of how `commitMutationEffects` might be implemented:

```
function commitMutationEffects(fiber) {  
  switch (fiber.tag) {  
    case HostComponent: {  
      // Update DOM node with new props and/or children  
      break;  
    }  
  }  
}
```

```
,  
case HostText: {  
    // Update text content of DOM node  
    break;  
}  
case ClassComponent: {  
    // Call lifecycle methods like componentDidMount  
    break;  
}  
// ... other cases for different types of nodes  
}  
}
```

During the mutation phase, React also calls other special functions, such as `commitUnmount` and `commitDeletion`, to remove nodes from the DOM that are no longer needed.

The Layout Phase

The layout phase is the second part of the commit phase, and it is responsible for calculating the new layout of the updated nodes in the DOM. During this phase, React walks through the fiber tree again, looking for nodes that have been marked as “dirty” (i.e., nodes that have been updated).

For each dirty node, React calls a special function called `commitLayoutEffects`. This function calculates the new layout of

the updated node in the DOM.

Like `commitMutationEffects`, `commitLayoutEffects` is also a massive switch statement that calls different functions depending on the type of node being updated.

Once the layout phase is complete, React has successfully updated the actual DOM to reflect the changes that were made to the virtual DOM during the render phase.

By dividing the commit phase into two parts (mutation and layout), React is able to apply updates to the DOM in a more efficient and optimized way. By working in concert with other key functions in the reconciler, such as `beginWork` and `completeWork`, the commit phase helps to ensure that React applications are fast, responsive, and reliable, even as they become more complex and handle larger amounts of data.

Effects

During the commit phase of React's reconciliation process, side effects are performed in a specific order, depending on the type of effect. There are several types of effects that can occur during the commit phase, including:

- **Placement effects:** These effects occur when a new component is added to the DOM. For example, if a new button is added to a form, a placement effect will occur to add the button to the DOM.
- **Update effects:** These effects occur when a component is updated with new props or state. For example, if the text of a button changes, an update effect will occur to update the text in the DOM.
- **Deletion effects:** These effects occur when a component is removed from the DOM. For example, if a button is removed from a form, a deletion effect will occur to remove the button from the DOM.
- **Layout effects:** These effects occur before the browser has a chance to paint, and are used to update the layout of the page. Layout effects are managed using the `useLayoutEffect` hook in functional components and the `componentDidUpdate` lifecycle method in class components.

In contrast to these commit-phase effects, passive effects are user-defined effects that are scheduled to run after the browser has had a chance to paint. Passive effects are managed using the `useEffect` hook with an empty dependency array. This allows the effect to be scheduled as a passive effect, rather than as a commit-phase effect.

Passive effects are useful for performing actions that are not critical to the initial rendering of the page, such as fetching data from an API

or performing analytics tracking. Because passive effects are not performed during the render phase, they do not affect the performance or perceived speed of the user interface.

Putting Everything on the Screen

React maintains a `FiberRootNode` atop both trees that points to one of the two trees: the `current` or the `workInProgress` trees. The `FiberRootNode` is a key data structure that sits atop both trees, and is responsible for managing the commit phase of the reconciliation process. The `FiberRootNode` contains a reference to the current tree, as well as a reference to the `workInProgress` tree, which is used during the render phase.

When updates are made to the virtual DOM, React updates the `workInProgress` tree, while leaving the current tree unchanged. This allows React to continue rendering and updating the virtual DOM, while also preserving the current state of the application.

When the rendering process is complete, React calls a function called `commitRoot`, which is responsible for committing the changes made to the `workInProgress` tree to the actual DOM. `commitRoot` switches the pointer of the `FiberRootNode` from the current tree to the `workInProgress` tree, making the `workInProgress` tree the new current tree.

From this point on, any future updates are based on the new current tree. This process ensures that the application remains in a consistent state, and that updates are applied correctly and efficiently.

All of this happens apparently instantly in the browser. This is the work of reconciliation.

Chapter Review

In this chapter, we explored the concept of React Reconciliation and learned about the Fiber Reconciler, which is a highly optimized implementation of the reconciliation algorithm. We also learned about Fibers, which are a new data structure introduced in the Fiber Reconciler that represent a unit of work in the reconciliation process. We also learned about the render phase and the commit phase, which are the two main phases of the reconciliation process. Finally, we learned about the `FiberRootNode`, which is a key data structure that sits atop both trees, and is responsible for managing the commit phase of the reconciliation process.

Review Questions

Let's ask ourselves a few questions to test our understanding of the concepts in this chapter:

1. What is React Reconciliation?
2. What's the role of the Fiber data structure?
3. Why do we need two trees?
4. What happens when an application updates?
5. What are render lanes?

If we can answer these questions, we should be well on our way to understanding the Fiber Reconciler and the reconciliation process in React.

Up Next

In the next chapter, we'll look at common questions in React and explore some advanced patterns. We'll answer questions around how often to `useMemo` and when to use `React.lazy`. We'll also explore how to use `useReducer` and `useContext` to manage state in React applications.

See you there!

Chapter 5. Common Questions and Powerful Patterns

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

Now that we’re more aware of what React does and how it works under the hood, let’s explore its practical applications a little deeper in how we write React applications. In this chapter, we’ll explore the answers to common React questions to boost our fluency like:

- What is memoization in React and how can it improve performance?

- How does `React.memo()` work to memoize a functional component?
- Can you provide an example of when memoization would be useful in a React application?
- What is lazy loading in React and how can it improve performance?
- How can you implement lazy loading in a React application using `React.lazy()`?
- What are the benefits of using reducers over `useState` in a React application?
- What is the difference between a reducer and a state setter function in React?
- Can you provide an example of when using a reducer would be better than `useState` in a - React application?
- How can you combine multiple reducers in a React application using `combineReducers()`?
- What are the best practices for using memoization, lazy loading, and reducers in a React application?

Let's get started by talking about memoization.

Memoization with `React.memo`

Memoization is a technique used in computer science to optimize the performance of functions by caching their previously computed results. In simple terms, memoization stores the output of a function based on its inputs so that if the function is called again with the same inputs, it returns the cached result rather than recomputing the output. This can significantly reduce the time and resources needed to execute a function, especially for functions that are computationally expensive or called frequently.

In the context of React, memoization can be applied to functional components using the `React.memo()` higher-order component. This function returns a new component that only re-renders if its props have changed. By memoizing functional components, we can prevent unnecessary re-renders, which can improve the overall performance of our React application. Memoization is particularly useful when dealing with expensive calculations or when rendering large lists of items.

Consider a function:

```
let result = null;
const doHardThing = () => {
  if (result) return result;

  // ...do hard stuff
```

```
    result = hardStuff;  
    return hardStuff;  
};
```

Calling `doHardThing` once might take a few minutes to do the hard thing, but calling it a second, third, fourth, *nth* time, doesn't actually do the hard thing but instead returns the stored result. This is the gist of **memoization**. React enables us to memoize components using an exported function called `memo`. It also enables us to memoize values inside of our components using a hook called `useMemo`. We'll talk about both in detail further in this chapter, but let's first understand why React enables these usages.

We already know that React components are functions that are often invoked for reconciliation, as discussed in Chapter 4. Sometimes, reconciliation (that is, invoking a component function) can take a long time due to intense computations. This would slow down our application and present a bad user experience. Memoization is a way to avoid this by storing the results of expensive computations—either UI elements or concrete values—and returning them when the same inputs are passed to the function, or the same props are passed to the component.

To understand why `React.memo` is important, let's consider a common scenario where we have a list of items that need to be rendered

in a component. For example, let's say we have a list of todos that we want to display in a component like this:

```
function TodoList({ todos }) {  
  return (  
    <ul>  
      {todos.map((todo) => (  
        <li key={todo.id}>{todo.title}</li>  
      ))}  
    </ul>  
  );  
}
```

If the list of todos is large, and the component is re-rendered frequently, this can cause a performance bottleneck in the application. One way to optimize this component is to memoize it using `React.memo()`:

```
const MemoizedTodoList = React.memo(function TodoList({ todos }) {  
  return (  
    <ul>  
      {todos.map((todo) => (  
        <li key={todo.id}>{todo.title}</li>  
      ))}  
    </ul>  
  );  
});
```

By wrapping the `TodoList` component with `React.memo`, React will only re-render the component if its props have changed. This means that if the list of todos remains the same, the component will not re-render, and the cached output will be used instead. This can save significant resources and time, especially when the component is complex and the list of todos is large.

In addition to improving performance, `React.memo` can also make the code more maintainable and easier to reason about. By memoizing components, we can ensure that their outputs remain consistent, which can reduce the risk of introducing bugs in the application.

Let's consider another example where we have a complex component with multiple nested components that are expensive to render:

```
function Dashboard({ data }) {  
  return (  
    <div>  
      <h1>Dashboard</h1>  
      <UserStats user={data.user} />  
      <RecentActivity activity={data.activity} />  
      <ImportantMessages messages={data.messages} />  
    </div>  
  );  
}
```

If the `data` prop changes frequently, this component can be expensive to render, especially if the nested components are also complex. We can optimize this component using `React.memo` to memoize each nested component:

```
const MemoizedUserStats = React.memo(function UserStats({ data }) {
  // ...
});

const MemoizedRecentActivity = React.memo(function RecentActivity({ data }) {
  activity,
}); {
  // ...
});

const MemoizedImportantMessages = React.memo(function ImportantMessages({ data }) {
  messages,
}); {
  // ...
});

function Dashboard({ data }) {
  return (
    <div>
      <h1>Dashboard</h1>
```

```
    <MemoizedUserStats user={data.user} />
    <MemoizedRecentActivity activity={data.act:
    <MemoizedImportantMessages messages={data.r
  </div>
);
}
```

By memoizing each nested component, React will only re-render the components that have changed, and the cached outputs will be used for the components that have not changed. This can significantly improve the performance of the `Dashboard` component and reduce unnecessary re-renders.

Thus, we can see that `React.memo` is an essential tool for optimizing the performance of functional components in React. By memoizing components, we can significantly reduce the number of unnecessary re-renders and improve the overall performance of the application. Memoization can be particularly useful for components that are expensive to render or have complex logic.

Getting Fluent in `React.memo`

Let's briefly walk through how `React.memo` works. When an update happens in React, your component is compared with the results of the element returned from its previous render. If these results are different—i.e, if its props change—the reconciler runs an update effect if

the element already exists in the host environment (usually the browser DOM), or a placement effect if it doesn't. If its props are the same, the component still re-renders and the DOM is still updated.

Let's explore this with an example:

```
export const Avatar = ({ url, name }) => {  
  return <img alt={name} src={url} />;  
};  
  
export const MemoizedAvatar = React.memo(Avatar);
```

Above we have 2 components: `Avatar` and `MemoizedAvatar`. `React.memo` tells React to avoid re-rendering this component (i.e, don't invoke its function) unless and until its props change. Let's assume a form like this:

```
const Form = () => {  
  const [data, setData] = useState({  
    name: "",  
    avatarUrl: "",  
  });  
  
  return (  
    <form>  
      <label>  
        Your Name
```

```

        <input
          type="text"
          value={data.name}

          onChange={(e) => setData(e.target.value)}
        />
      </label>
      <label>
        Your Avatar
        <input
          type="file"
          onChange={(e) => {
            /* set url */
          }}
        />
      </label>
      <section>
        Preview
        <Avatar alt="An avatar" url={data.avatarUrl} />
      </section>
    </form>
  );
};

```

When entering a name, the form and all its children **re-render on every keystroke**. Specifically, `Avatar` re-renders on every keystroke—even if its props are the same, even if `data.avatarUrl`

doesn't change. If we swap `Avatar` with our new `MemoizedAvatar`, then the component itself doesn't re-render on every keystroke. Instead, it only rerenders when its props change: i.e, when `data.avatarUrl` is updated. Now, our application behaves as we'd expect: only what changes is recomputed.

This is what `React.memo` is good for: avoiding unnecessary re-renders when a component's props are identical between renders. Since we can do this in React, it begs the question: how much and how often should we memoize stuff? Surely if we memoize every component our application might be faster overall, no?

When to (and when to not) Memoize

Our user profile example is a good usage of component memoization on the `Avatar` component because the component's props do not change as often as the component's state, which changes on every keystroke.

Normally, using the virtual DOM and its efficient reconciliation algorithms, React handles most performance concerns for us out of the box because that's what it's designed to do and we often do not need to manually memoize things: it is considered valuable to use memoization in React sparingly because of the following 2 key reasons.

1. **Everything has a cost.** While implementing memoization itself has a minimal cost, using React's memoization primitives unfortunately comes with some overhead that may result in our applications doing unnecessary work: like importing, then invoking `React.memo` which would in turn execute its own logic, further consuming stack frames and resources that are not needed. While `React.memo` is a useful tool for optimizing the performance of functional components in React, it's important to understand the potential overhead of using this technique and how to use it effectively. Let's explore this further with some code examples.

One potential overhead is the additional memory usage required to store the memoized outputs. If the component is large or has many dependencies, this can result in increased memory usage. For example, consider the following memoized component:

```
const MemoizedComponent = React.memo(({ items
  // perform expensive computation based on i
  const result = items.reduce((total, item) =

return (
  <div>
    <p>Result: {result}</p>
    <button onClick={handleClick}>Click me<
  </div>
);
```

```
});
```

In this example, the `MemoizedComponent` performs an expensive computation based on the `items` prop. While memoization can reduce unnecessary re-renders, it also requires storing the memoized output in memory. This can result in increased memory usage, especially if the component is large or has many dependencies.

Another potential overhead of using `React.memo` is the additional time required to perform the memoization. When a component is memoized, React needs to check whether the inputs to the component have changed before re-rendering. This can result in additional computation time, especially for complex components or those with many dependencies.

It's important to remember that memoization may not provide significant performance improvements for all components. In some cases, memoization may not be necessary, especially for small or simple components.

By weighing the benefits and overhead, developers can determine whether memoization is necessary and implement it effectively to optimize the performance of their React applications.

If we have a component tree wherein a component's props regularly change, memoizing this component may cost more than not. Things get even worse if we pass a comparison function to

`React.memo` that runs every time, even though the props are frequently different. The general wisdom on when to memoize a component and when to forego it lies within quantification: if we are unable to yield measurable performance improvements with memoization, then it's best to forego it and trust React.

2. **Bugs.** Memoization is a powerful tool, but it can also be a double-edged sword. If we memoize a component that depends on enclosing state, we may end up with stale props that are not updated when the enclosing state changes. Trying to understand why a component is not updating can be a frustrating experience, especially if we're not aware of the memoization.

Here's an example of how memoization can lead to stale props when a component depends on enclosing state:

```
function Counter() {  
  const [count, setCount] = useState(0);  
  
  const increment = () => {  
    setCount(count + 1);  
  };  
  
  const memoizedIncrement = useCallback(() =>  
    setCount(count + 1);  
  , []);  
  
  console.log("Counter rendered");  
}
```

```

    return (
      <div>
        <p>Count: {count}</p>
        <button onClick={increment}>Increment</button>
        <MemoizedChild increment={memoizedIncrement}></MemoizedChild>
      </div>
    );
  }

  const MemoizedChild = React.memo(({ increment }) => {
    console.log("Child rendered");
    return <button onClick={increment}>Memoized Increment</button>
  });

  function App() {
    return <Counter />;
  }

```

In this example, we have a `Counter` component that uses state to track the count value. The `Counter` component also provides a memoized version of the increment function to a `MemoizedChild` component, which is also memoized using `React.memo`. The `MemoizedChild` component renders a button that calls the memoized increment function when clicked.

While this code may appear to work correctly at first, it actually has a bug: the `MemoizedChild` component may not update correctly when the count state changes. This is because the memoized increment function depends on the count state, but it is not included in the dependencies array of the `useCallback` hook. We'll discuss `useCallback` more in detail later, but as a result, the memoized increment function will not update when the count state changes, and the `MemoizedChild` component will receive stale props.

To fix this bug, we can include the memoized increment function in the dependencies array of the `useCallback` hook, like this:

```
const memoizedIncrement = useCallback(() => {
  setCount(count + 1);
}, [count]);
```

With this change, the memoized increment function will update whenever the count state changes, and the `MemoizedChild` component will receive updated props.

Memoization can be a powerful tool for optimizing the performance of functional components in React, but it's important to understand its limitations and potential pitfalls, such as stale props. By being mindful of these issues and using memoization effectively, developers can

optimize the performance of their React applications and provide a better user experience.

Memoized components that still re-render

`React.memo` performs what is called a *shallow* comparison of the props to determine whether they've changed or not. The problem with this is while scalar types can be compared quite accurately in JavaScript, non-scalars cannot. Consider the following example:

```
// Scalar types
"a" === "a"; // string; true
3 === 3; // number; true

// Non-scalar types
[1, 2, 3] === [1, 2, 3]; // array; false
```

What happens with the array comparison above is that the arrays are compared *by value*. While they look the same to us, the left and right hand sides of the array comparison are two different *instances* of arrays. To combat this, we can compare a *reference* to the array instead of a new array:

```
const myArray = [1, 2, 3];
```

```
myArray === myArray; // true
myArray === [1, 2, 3]; // false
```

This is why it's generally a good practice to pass references to memoized components as props, and not values. For example, do:

```
const Parent = () => {
  const myArray = [1, 2, 3];
  return <MemoizedComponent myArray={myArray} />;
};
```

instead of:

```
const Parent = () => {
  return <MemoizedComponent myArray={[1, 2, 3]} />;
};
```

`React.memo` often also gets circumvented quite commonly by another non-scalar type: functions. Consider the following case:

```
<MemoizedAvatar
  name="Tejas"
  url="https://github.com/tejasq.png"
```

```
    onChange={() => save()}  
  />
```

While the props don't appear to change or depend on enclosing state with props `name`, `url`, and `onChange` all having constant values, if we compare the props we see the following:

```
"Tejas" === "Tejas"; // <- `name` prop; true  
"https://github.com/tejasq.png" === "https://github.com/tejasq.png"; // <- `url` prop; true  
(() => save()) === (() => save()); // <- `onChange` prop; true
```

Once again, this is because we're comparing functions *by value*. Remember as long as props differ, our component will not be memoized. We can combat this two ways:

1. By using the `useCallback` hook inside `MemoizedAvatar`'s parent:

```
const Parent = () => {  
  const onAvatarChange = useCallback(() => save(), []);  
  
  return;  
  <MemoizedAvatar  
    name="Tejas"  
    url="https://github.com/tejasq.png"  
    onChange={onAvatarChange} />  
}
```

```
url="https://github.com/tejasq.png"
onChange={onAvatarChange}
/>
};
```

Now, we can be confident that `onAvatarChange` will never change unless one of the things in its dependency array (second argument) changes. Since it's empty though, our memoization is fully complete and reliable. This is the recommended way to memoize components that have functions as props. Another way we can get around this is as follows.

2. By passing a comparison function to `React.memo` :

`React.memo` takes a second argument, a comparison function, which is used to compare the previous props to the next props. If the comparison function returns `true`, then the component will not re-render—indicating the props are equal. If it returns `false`, then the component will re-render. We can use this to our advantage to ensure that our component only re-renders when the props we care about change:

```
const MemoizedAvatar = React.memo(Avatar, (prevProps, nextProps) => {
  return (
    prevProps.name === nextProps.name &&
    prevProps.url === nextProps.url &&
    prevProps.onChange === nextProps.onChange
  );
});
```

```
    );  
  });
```

In this scenario however, `nextProps.onChange` will contain a new function reference every time its parent re-renders because functions are passed by value. We can get around this by serializing the function in the comparison function:

```
const MemoizedAvatar = React.memo(Avatar, (prevProps, nextProps) => {  
  return (  
    prevProps.name === nextProps.name &&  
    prevProps.url === nextProps.url &&  
    prevProps.onChange.toString() === nextProps.onChange.toString()  
  );  
});
```

Now, comparing strings to strings, we can be confident that `onChange` will never change unless the actual function implementation represented as a string changes. This is a bit of a hack because there are usually better ways to compare functions than serializing them to strings: like comparing references to functions instead.

We could literally rewrite our hack to compare references to `onChange` instead of serializing it to a string like so:

```
const MemoizedAvatar = React.memo(Avatar, (prevProps, nextProps) => {
  return (
    prevProps.name === nextProps.name &&
    prevProps.url === nextProps.url &&
    prevProps.onChange === nextProps.onChange
  );
});
```

Regardless, we're doing this to illustrate the point of shallow comparison in a dependency array. Please be aware that the `useCallback` hook is the recommended way to memoize functions passed as props to components instead of serializing them to strings.

Whew! Great! This now means that our memoized components will never unnecessarily re-render. Right? Wrong! There's one more thing we need to be aware of.

It's a guideline, not a rule

React uses `React.memo` as a hint to its reconciler that we don't want our components to re-render if their props stay the same.

They're just hints to React. Ultimately, what React does is up to React. To echo back to the beginning of this book, React is intended to be a declarative abstraction of our user interface where we describe *what*

we want, and React figures out the best *how* to do it. `React.memo` is a part of this.

`React.memo` does not guarantee consistently avoided re-renders. This is because React may decide to re-render a memoized component for various reasons, such as changes to the component tree or changes to the global state of the application.

To understand why `React.memo` does not guarantee consistently avoided re-renders, let's take a look at some code snippets from React's source code.

First, let's look at the implementation of `React.memo`:

```
function memo(type, compare) {  
  return {  
    $$typeof: REACT_MEMO_TYPE,  
    type,  
    compare: compare === undefined ? null : compare  
  };  
}
```

In this implementation, `React.memo` returns a new object that represents the memoized component. The object has a `$$typeof` property that identifies it as a memoized component, a `type` proper-

ty that references the original component, and a `compare` property that specifies the comparison function to use for memoization.

Next, let's look at the implementation of the React reconciler's memoization algorithm:

```
function performWorkOnRoot(root, expirationTime,
  // ...

  const updateExpirationTimeBeforeCommit = workIn

  // ...

  if (workInProgress.memoizedProps !== null && !
    // Memoized component, compare new props to r
    if (nextChildren === null || !shouldSetTextCo
      bailoutOnAlreadyFinishedWork(
        workInProgress,
        expirationTime,
        updateExpirationTimeBeforeCommit
      );
      return;
    }
  }

  // ...

  // Render new children
```

```
    reconcileChildren(current, workInProgress, next)

    // ...
}
```

In this implementation, the React reconciler checks whether a component is memoized by comparing its current props to its previous props. If the props have not changed and the component is not expired, the reconciler will skip rendering the component and reuse the previous output instead.

However, there are cases where the React reconciler may still re-render a memoized component even if its props have not changed. For example, if the component's parent has been updated or if the global state of the application has changed, the memoized component may need to be re-rendered to reflect these changes. Additionally, if a component is expired, the React reconciler will ignore its memoization and re-render it from scratch.

In React, a component is considered “expired” when its previous render is no longer valid and cannot be reused. This can happen for a variety of reasons, such as changes to the component's props, state, or context, or changes to the global state of the application.

When a component is expired, React will not reuse its previous output and will instead re-render the component from scratch. This can be a performance hit, especially for complex components, as it requires more computation and can cause unnecessary re-renders.

Thus, `React.memo` does not guarantee consistently avoided re-renders. Developers should use `React.memo` effectively and understand its limitations to optimize the performance of their React applications and provide a better user experience.

Memoization with `useMemo`

What `React.memo` does to components, the `useMemo` hook does to values inside components. Let's briefly explore `useMemo` while we're here. Consider a component:

```
const People = ({ unsortedPeople }) => {
  const [name, setName] = useState("");
  const sortedPeople = unsortedPeople.sort((a, b) => {
    if (a.name < b.name) return -1;
    if (a.name > b.name) return 1;
    return 0;
  });

  return (
    <div>
      <div>
        Enter your name: { " " }
        <input type="text" value={name} onChange={setName} />
      </div>
      <div>
        {sortedPeople.map((person) => {
          return (
            <div>
              {person.name}
            </div>
          );
        })}
      </div>
    </div>
  );
};
```

```

        type="text"
        placeholder="Obinna Ekwuno"
        onChange={(e) => setName(e.target.value)}
      />
    </div>
    <h1>Hi, {name}! Here's a list of people sorted by age</h1>
    <ul>
      {sortedPeople.map((p) => (
        <li key={p.id}>
          {p.name}, age {p.age}
        </li>
      ))}
    </ul>
  </div>
);
};

```

This component is going to slow down our application because it has **linear runtime complexity**: that is, it iterates through every person in the list to compare their age to the previous person. If our list has 1000000 people, it will perform 1000000 iterations at least. In computer science, this is called $O(N)$ time complexity.

What makes things even worse is that this component will re-render whenever its state updates: which is on every keystroke inside the input field for a person's name. If the name is 5 characters and our list

has 1000000 people, we'd be making our app do 5 million operations in total! We can avoid this using `useMemo`.

Let's rewrite that code snippet a little bit:

```
const People = ({ unsortedPeople }) => {
  const [name, setName] = useState("");
  const sortedPeople = useMemo(
    () => unsortedPeople.sort((a, b) => b.age - a.age),
    [unsortedPeople]
  );

  return (
    <div>
      <div>
        Enter your name: {name}
        <input
          type="text"
          placeholder="Obinna Ekwuno"
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <h1>Hi, {name}! Here's a list of people sorted by age</h1>
      <ul>
        {sortedPeople.map((p) => (
          <li key={p.id}>
            {p.name}, age {p.age}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

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

There! Much better! We wrapped the value of `sortedPeople` in a function that was passed to the first argument of `useMemo`. The second argument we pass to `useMemo` represents an array of values that, if changed, re-sorts the array. Since the array contains only `unsortedPeople`, it will only sort the array once, and everytime the list of people changes—not whenever someone types in the name input field. This is a great example of how to use `useMemo` to avoid unnecessary re-renders.

`useMemo` considered harmful

While it might now be tempting to wrap all variable declarations inside a component with `useMemo`, this is not a good idea. `useMemo` is a great tool for memoizing expensive computations, but it's not a good idea to use it for memoizing scalar values.

In JavaScript, a scalar values are a primitive values that are not an object. Scalar values include `undefined`, `boolean s`, `number s`,

and `strings`. These values are immutable, meaning that they cannot be modified once they are created.

A string is a scalar value, but an array is not. A number is a scalar value, but an object is not. In these cases, `useMemo` is not necessary because the value is not a reference to another value. It's a value in and of itself—and in this case, not only is it inexpensive to compute, but we also *want* to recompute the value every time the component re-renders.

Consider the following example:

```
const MyComponent = () => {
  const dateOfBirth = "1993-02-19";
  const isAdult =
    new Date().getFullYear() - new Date(dateOfBirth).getFullYear();

  if (isAdult) {
    return <h1>You are an adult!</h1>;
  } else {
    return <h1>You are a minor!</h1>;
  }
};
```

We're not using `useMemo` here anywhere, mainly because the component is stateless. It's a pure function that takes in a date of birth

and returns a string. This is good! But what if we have some input that triggers rerenders like this:

```
const MyComponent = () => {
  const [birthYear, setBirthYear] = useState(1990)
  const isAdult = new Date().getFullYear() - birthYear

  return (
    <div>
      <label>
        Birth year:
        <input
          type="number"
          value={birthYear}
          onChange={(e) => setBirthYear(e.target.value)}
        />
      </label>
      {isAdult ? <h1>You are an adult!</h1> : <h1>You are not an adult!</h1>}
    </div>
  );
};
```

Now, we're recomputing `new Date()` on every keystroke. Let's fix this with `useMemo`:

```
const MyComponent = () => {
  const [birthYear, setBirthYear] = useState(1990)
```

```

const [birthYear, setBirthYear] = useState(1990);
const isAdult = today.getFullYear() - birthYear;
const today = useMemo(() => new Date(), []);

return (
  <div>
    <label>
      Birth year:
      <input
        type="number"
        value={birthYear}
        onChange={(e) => setBirthYear(e.target.value)}
      />
    </label>
    {isAdult ? <h1>You are an adult!</h1> : <h1>You are not an adult!</h1>}
  </div>
);
};

```

This is good because `today` will be a reference to the same object every time the component re-renders, and we assume the component will always re-render in the same day.

There's a slight edge case here if the user's clock lapses midnight while they're using this component, but this is a rare edge case that we can ignore for now. Of course, we do better when there's real production code involved.

This example is to facilitate a bigger question: should we wrap `isAdult`'s value in `useMemo`? What happens if we do? The answer is that we shouldn't because `isAdult` is a scalar value: a boolean and it's not expensive to compute. We *do* call `.getFullYear` a bunch of times, but we trust the JavaScript engine and the React runtime to handle the performance for us. It's a simple assignment with no further computation like sorting, filter, or mapping.

In this case, we should not use `useMemo` because it is more likely to slow our app down than speed our app up because of the overhead of `useMemo` itself: including importing it, calling it, passing in the dependencies, and then comparing the dependencies to see if the value should be recomputed. All of this has runtime complexity that can hurt our apps more than help it. Instead, we assign and trust React to intelligently re-render our component when necessary with its own optimizations.

Our applications are now enjoying performance benefits of faster rerenders even in the face of heavy computations—but can we do more? In the next section, let's take a look at shrinking the amount of JavaScript our users have to download using code splitting with React's lazy loading primitives.

Lazy Loading

As our applications grow, we accumulate a lot of JavaScript. Our users then download these massive JavaScript bundles—sometimes going into the double digits on megabytes—only to use a small portion of the code. This is a problem because it slows down our users' initial load time, and it also slows down our users' subsequent page loads because they have to download the entire bundle again.

Shipping too much JavaScript can be problematic for users with limited internet capabilities, as it can lead to slower page load times, increased data usage, and decreased accessibility. Let's explore why shipping too much JavaScript is a problem, how it affects users, and what we can do to mitigate these issues.

One of the main problems with shipping too much JavaScript is that it can slow down page load times. JavaScript files are typically larger than other types of web assets, such as HTML and CSS, and require more processing time to execute. This can lead to longer page load times, especially on slower internet connections or older devices.

For example, consider the following code snippet that loads a large JavaScript file on page load:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
    <script src="https://example.com/large.js"></script>
  </head>
  <body>
    <!-- Page content goes here -->
  </body>
</html>
```

In this example, the `large.js` file is loaded in the `<head>` of the page, which means that it will be executed before any other content on the page. This can lead to slower page load times, especially on slower internet connections or older devices.

Another problem with shipping too much JavaScript is that it can increase data usage. JavaScript files are typically larger than other types of web assets, which means that they require more data to be transferred over the network. This can be a problem for users with limited data plans or slow internet connections, as it can lead to increased costs and slower page load times.

In our previous example, the `large.js` file is loaded in the `<head>` of the page, which means that it will be downloaded and ex-

executed before any other content on the page. This can lead to increased data usage, especially on slower internet connections or limited data plans.

Finally, shipping too much JavaScript can also decrease accessibility. Users with older devices or slower internet connections may not be able to load or execute large JavaScript files, which can lead to broken functionality or incomplete user experiences. Additionally, users with disabilities who rely on assistive technologies may also be affected by excessive JavaScript, as it can interfere with the operation of these tools and make it difficult or impossible for them to access content on the page.

To mitigate these issues, we can take several steps to reduce the amount of JavaScript that is shipped to users. One approach is to use code splitting to load only the JavaScript that is needed for a particular page or feature. This can help reduce page load times and data usage by only loading the necessary code.

For example, consider the following code snippet that uses code splitting to load only the JavaScript that is needed for a particular page:

```
import("./large.js").then((module) => {  
  // Use module here
```

```
});
```

In this example, the `import()` function is used to asynchronously load the `large.js` file only when it is needed. This can help reduce page load times and data usage by only loading the necessary code.

Another approach is to use lazy loading to defer the loading of non-critical JavaScript until after the page has loaded. This can help reduce page load times and data usage by loading non-critical code only when it is needed.

For example, consider the following code snippet that uses lazy loading to defer the loading of non-critical JavaScript:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <!-- Page content goes here -->
    <button id="load-more">Load more content</button>
    <script>
      document.getElementById("load-more").addEventListener("click", () => {
        import("./non-critical.js").then((module) => {
          // Use module here
        });
      });
    </script>
  </body>
</html>
```

```
    });  
  </script>  
</body>  
</html>
```

In this example, the `import()` function is used to asynchronously load the `non-critical.js` file only when the “Load more content” button is clicked. This can help reduce page load times and data usage by loading non-critical code only when it is needed.

Thankfully, React has a solution that makes this even more straightforward: lazy loading using `React.lazy` and `Suspense`. Let’s take a look at how we can use these to improve our application’s performance.

Lazy loading is a technique that allows us to load a component only when it’s needed, like with the dynamic import above. This is useful for large applications that have many components that are not needed on the initial render. For example, if we have a large application with a collapsible sidebar that has a list of links to other pages, we might not want to load the full sidebar if it’s collapsed on first load. Instead, we can load it only when the user toggles the sidebar.

Let’s explore the following code sample:

```

import { Sidebar } from "../Sidebar"; // 22MB to :

const MyComponent = ({ initialState }) => {
  const [showSidebar, setShowSidebar] = useState(false);

  return (
    <div>
      <button onClick={() => setShowSidebar(!showSidebar)}>
        Toggle sidebar
      </button>
      {showSidebar && <Sidebar />}
    </div>
  );
};

```

In this example, let's imagine that `<Sidebar />` is 22MB of JavaScript. This is a lot of JavaScript to download, parse, and execute, and it's not necessary on the initial render if the sidebar is collapsed. Instead, we can use `React.lazy` to lazy load the component, only if `showSidebar` is true. As in, only if we need it as below:

```

import { lazy, Suspense } from "react";

const Sidebar = lazy(() => import("../Sidebar"));

const MyComponent = ({ initialState }) => {

```

```

const [showSidebar, setShowSidebar] = useState(false)

return (
  <div>
    <button onClick={() => setShowSidebar(!showSidebar)}>
      Toggle sidebar
    </button>
    <Suspense>{showSidebar && <Sidebar />}</Suspense>
  </div>
);
};

```

Instead of **statically importing** `./Sidebar`, we **dynamically** import it—that is, we pass a function to `lazy` that returns a promise that resolves to the imported module. A dynamic import returns a promise because the module may not be available immediately. It may need to be downloaded from the server first. React's `lazy` function which triggers the import is never called unless the underlying component (in this case, `Sidebar`) is to be rendered. This way, we avoid shipping the 22MB sidebar until we actually *render* `<Sidebar />`.

You may have also noticed another new import: `Suspense`. What's that doing there? We use `Suspense` to wrap the component in the tree. `Suspense` is a component that allows us to show a fallback component while the promise is resolving (read: as the sidebar is

downloading). In the snippet above, we're not showing a fallback component but we could if we wanted to by adding a `fallback` prop to `Suspense` like so:

```
import { lazy, Suspense } from "react";

const Sidebar = lazy(() => import("./Sidebar"));

const MyComponent = () => {
  const [showSidebar, setShowSidebar] = useState(false);

  return (
    <div>
      <button onClick={() => setShowSidebar(!showSidebar)}>
        Toggle sidebar
      </button>
      <Suspense fallback={<p>Loading...</p>}>
        {showSidebar && <Sidebar />}
      </Suspense>
      <main>
        <p>Hello hello welcome, this is the app's main content</p>
      </main>
    </div>
  );
};
```

Now, when the user clicks the button to toggle the sidebar, they'll see "Loading..." while the sidebar is loaded and rendered. This is a great way to improve our application's performance by only loading the code we need when we need it and still providing immediate feedback to the user.

Greater UI Control with Suspense

React Suspense works like a try/catch block. You know how you can `throw` an exception from literally anywhere in your code, and then catch it with a `catch` block somewhere else—even in a different module? Well, Suspense works the same way. You can place lazy-loaded and asynchronous primitives anywhere in your component tree, and then catch them with a `Suspense` component anywhere above it in the tree, even if your suspense boundary is in a completely different file.

Knowing this, we have the power to choose where we want to show the loading state for our 22MB sidebar. For example, we can hide the entire application while the sidebar is loading—which is a pretty bad idea because we block our entire app's information from the user just for a sidebar—or we can show a loading state for the sidebar only. Let's take a look at how we can do the former (even though we shouldn't) just to understand `Suspense`'s capabilities.

```

import { lazy, Suspense } from "react";

const Sidebar = lazy(() => import("./Sidebar"));

const MyComponent = () => {
  const [showSidebar, setShowSidebar] = useState(false);

  return (
    <Suspense fallback={<p>Loading...</p>}>
      <div>
        <button onClick={() => setShowSidebar(!showSidebar)}>
          Toggle sidebar
        </button>
        {showSidebar && <Sidebar />}
        <main>
          <p>Hello hello welcome, this is the app</p>
        </main>
      </div>
    </Suspense>
  );
};

```

By wrapping the entire component in `Suspense`, we render the `fallback` until all asynchronous children (promises) are resolved. This means that the entire application is hidden until the sidebar is loaded. This can be useful if we want to wait until everything's ready

to reveal the user interface to the user, but in this case might not be the best idea because the user is left wondering what's going on and they can't interact with the application at all.

This is why we should only use `Suspense` to wrap the components that need to be lazy loaded, like this:

```
import { lazy, Suspense } from "react";

const Sidebar = lazy(() => import("./Sidebar"));

const MyComponent = () => {
  const [showSidebar, setShowSidebar] = useState(false);

  return (
    <div>
      <button onClick={() => setShowSidebar(!showSidebar)}>
        Toggle sidebar
      </button>
      <Suspense fallback={<p>Loading...</p>}>
        {showSidebar && <Sidebar />}
      </Suspense>
      <main>
        <p>Hello hello welcome, this is the app's main content</p>
      </main>
    </div>
  );
}
```

```
);  
};
```

The suspense boundary is a very powerful primitive that can remedy layout shift and make user interfaces more responsive and intuitive. It's a great tool to have in your arsenal. Moreover, if high-quality skeleton UI is used in the `fallback`, we can further guide our users to understand what's going on and what to expect while our lazy loaded components load, thereby orienting them to the interface they're about to interact with before it's ready. Taking advantage of all of this is a great way to improve our applications' performance and fluently get the most out of React.

Next, we'll look at another interesting question that many React developers ask: when should we use `useState` vs. `useReducer`?

`useState` vs. `useReducer`

React exposes two hooks for managing state: `useState` and `useReducer`. Both of these hooks are used to manage state in a component. The difference between the two is that `useState` is a hook that is better suited to manage a single piece of state, while `useReducer` is a hook that manages more complex state. Let's

take a look at how we can use `useState` to manage state in a component.

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

const MyComponent = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
    </div>

    );
};
```

In the above example, we're using `useState` to manage a single piece of state: `count`. But what if our state's a little more complex?

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

const MyComponent = () => {
  const [state, setState] = useState({
    count: 0,
    name: "Tejumma",
  });
};
```

```
    age: 30,  
  });  
  
  return (  
    <div>  
      <p>Count: {state.count}</p>  
      <p>Name: {state.name}</p>  
      <p>Age: {state.age}</p>  
      <button onClick={() => setState({ ...state,  
        Increment  
      })}>  
    </button>  
    </div>  
  );  
};
```

Now, we can see that our state is a little more complex. We have a `count`, a `name`, and an `age`. We can increment the `count` by clicking the button, which sets the state to **a new object** that has the same properties as the previous state, but with the `count` incremented by 1. This is a very common pattern in React. The problem with it is that it can raise the possibility of bugs. For example, if we don't carefully spread the old state, we might accidentally overwrite some of the state's properties.

At this point, you should know that `useState` uses `useReducer` internally. You can think of `useState` as a higher-level abstraction

of `useReducer`. In fact, one can reimplement `useState` with `useReducer` if they so wish!

Seriously, you'd just do this:

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

function useState(initialState) {
  const [state, dispatch] = useReducer(
    (state, newValue) => newValue,
    initialState
  );
  return [state, dispatch];
}
```

Let's look at the same example as above, but implemented with `useReducer` instead.

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

const initialState = {
  count: 0,
  name: "Tejumma",
  age: 30,
};

const reducer = (state, action) => {
```

```

const reducer = (state, action) => {
  switch (action.type) {
    case "increment":
      return { ...state, count: state.count + 1 };
    default:
      return state;
  }
};

const MyComponent = () => {
  const [state, dispatch] = useReducer(reducer, {
    count: 0,
    name: "John",
    age: 30
  });

  return (
    <div>
      <p>Count: {state.count}</p>
      <p>Name: {state.name}</p>
      <p>Age: {state.age}</p>
      <button onClick={() => dispatch({ type: "increment" })}>Increment</button>
    </div>
  );
};

```

Now some would say this is a tad more verbose than `useState` and many would agree, but this is to be expected whenever anyone goes a level lower in an abstraction stack: the lower the abstraction, the more verbose the code. After all, abstractions are intended to replace complex logic with syntax sugar in most cases. So since we

can do the same thing with `useState` as we can with `useReducer`, why don't we just always use `useState` since it's simpler?

There are 4 large benefits to using `useReducer` to answer this question:

1. It separates the logic of updating state from the component. Its accompanying `reducer` function can be tested in isolation, and it can be reused in other components. This is a great way to keep our components clean and simple and embrace the **single responsibility principle**.

We can test the reducer like this:

```
describe("reducer", () => {
  test("should increment count when given an
    const initialState = {
      count: 0,
      name: "Tejumma",
      age: 30,
    };
    const action = { type: "increment" };
    const expectedState = {
      count: 1,
      name: "Tejumma",
      age: 30,
    };
  });
```

```
    const actualState = reducer(initialState,
    expect(actualState).toEqual(expectedState
  });

  test("should return the same state object w
    const initialState = {
      count: 0,
      name: "Tejumma",
      age: 30,
    };
    const action = { type: "unknown" };
    const expectedState = initialState;
    const actualState = reducer(initialState,
    expect(actualState).toBe(expectedState);
  });
});
```

In this example, we're testing two different scenarios: one where the increment action is dispatched to the reducer, and one where an unknown action is dispatched.

In the first test, we're creating an initial state object with a count value of 0, and an increment action object. We're then expecting the count value in the resulting state object to be incremented to 1. We use the `toEqual` matcher to compare the expected and actual state objects.

In the second test, we're creating an initial state object with a count value of 0, and an unknown action object. We're then expecting the resulting state object to be the same as the initial state object. We use the `toBe` matcher to compare the expected and actual state objects, since we're testing for reference equality.

By testing our reducer in this way, we can ensure that it behaves correctly and produces the expected output when given different input scenarios.

2. The `dispatch` function returned from `useReducer` is stable and doesn't change between renders. This means that we can safely pass it down to child components without worrying about it changing and triggering expensive rerenders which we learned about in the first half of this chapter around memoization. This can be a great performance win in most cases.
3. Our state and the way it changes is always explicit with `useReducer`, and some would argue that `useState` can obfuscate the overall state update flow of a component through layers of JSX trees.
4. `useReducer` is an *event sourced* model: meaning it can be used to model events that happen in our application which we can then keep track of in some type of audit log. This audit log can be used to replay events in our application to reproduce bugs or to implement **time travel debugging**. It also enables

some powerful patterns like undo/redo, optimistic updates, and analytics tracking of common user actions across our interface.

While `useReducer` is a great tool to have in your arsenal, it's not always necessary. In fact, it's often overkill for most use cases. So when should we use `useState` vs. `useReducer`? The answer is that it depends on the complexity of your state. But hopefully with all of this information, you can make a more informed decision about which one to use in your application.

Whew! What a chapter! Let's wrap things up and summarize what we learned.

Chapter Review

Throughout this chapter, we've discussed various aspects of React, including memoization, lazy loading, reducers, and state management. We've explored the advantages and potential drawbacks of different approaches to these topics and how they can impact the performance and maintainability of React applications.

We started by discussing memoization in React and its benefits for optimizing component rendering. We looked at the `React.memo` function and how it can be used to prevent unnecessary re-renders of components. We also examined some potential issues with memoization, such as stale state and the need to carefully manage dependencies.

Next, we talked about lazy loading in React and how it can be used to defer the loading of certain components or resources until they are actually needed. We looked at the `React.lazy` and `Suspense` components and how they can be used to implement lazy loading in a React application. We also discussed the tradeoffs of lazy loading, such as increased complexity and potential performance issues.

We then moved on to reducers and how they can be used for state management in React. We explored the differences between

`useState` and `useReducer` and discussed the advantages of using a centralized reducer function for managing state updates.

Throughout our conversation, we used code examples from our own implementations to illustrate the concepts we discussed. We explored how these examples work under the hood and how they can impact the performance and maintainability of React applications.

In summary, our conversation covered a range of topics related to React, including memoization, lazy loading, reducers, and state management. We explored the advantages and potential drawbacks of different approaches to these topics and how they can impact the performance and maintainability of React applications. Through the use of code examples and in-depth explanations, we gained a deeper understanding of these topics and how they can be applied in real-world React applications.

Review Questions

Let's ask ourselves a few questions to test our understanding of the concepts we learned in this chapter:

1. What is memoization in React, and how can it be used to optimize component rendering?
2. What are the advantages of using `useReducer` for state management in React, and how does it differ from `useState`?

3. How can lazy loading be implemented in a React application using the `React.lazy` and `Suspense` components?
4. What are some potential issues that can arise when using memoization in React, and how can they be mitigated?
5. How can the `useCallback` hook be used to memoize functions passed as props to components in React?

Up Next

In the next chapter, we'll look at some newer features in React that have been introduced in the past year or so. We'll look at how we can use React's new server components to further optimize our apps and minimize first-load JS—even to 0kB!

Chapter 6. Server-Side React

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@oreilly.com.

React has made significant progress since its inception. Although it started as a client-side library, the demand for server-side rendering has grown over time for reasons we will come to understand in this chapter. Together, we will explore server-side React and understand how it differs from client-only React, and how it can be used to level up our React applications.

As we’ve discussed in earlier chapters, React was initially developed by Meta to address the need for efficient and scalable user interfaces

(UIs). We've looked at how it does this through the virtual DOM back in chapter 3, which enables developers to create and manage UI components with ease. React's client-side approach unlocked fast, responsive user experiences across the web. However as the web continued to evolve, the limitations of client-side rendering became more apparent.

Limitations of Client-Side Rendering

Client-side rendering (CSR) has been the primary approach for building user interfaces with React since it was first open sourced in 2013. 3 years later, React 16 introduced a new rendering mode called "ReactDOMServer" that allowed developers to render React components on the server and send the resulting HTML to the client, instead of rendering the components in the browser. This made it easier to build server-rendered React applications without the need for additional libraries or complex configuration.

React's prior client-only approach allowed us to build applications that provide fast and responsive user experiences, but as web applications have grown more complex, the limitations of client-only rendering (sometimes called "CSR" or client-side rendering) have become more apparent. In this section, we will explore the limitations of

client-side rendering and why server-side rendering (SSR) has become necessary for modern web applications.

SEO and Accessibility

One of the significant limitations of client-side rendering is that search engine crawlers may not correctly index the content, as some of them do not execute JavaScript. This can result in poor search engine optimization (SEO) and accessibility issues for users who rely on screen readers since without JavaScript, client-only React apps are just a blank page, sometimes called an “app shell”.

Consider the following example:

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

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

  useEffect(() => {
    fetch("https://api.example.com/data")
      .then((response) => response.json())
      .then((data) => setData(data));
  }, []);
```

```
return (  
  <div>  
    {data.map((item) => (  
      <div key={item.id}>{item.title}</div>  
    ))}  
  </div>  
);  
};  
  
export default Home;
```

In this example, we are fetching data from an API and rendering it on the client-side. We can tell it's the client side because we are using the `useEffect` hook to fetch the data, and the `useState` hook to store the data in state. These hooks execute inside a browser (a client) only.

A serious limitation with this is that some search engine crawlers will not be able to see this content unless we implement server-side rendering. Instead, they'll see a blank screen or a fallback message which can result in poor SEO.

Performance

Client-side rendering can have performance issues, especially on slower devices and networks. This is because of network waterfalls,

wherein the initial page load is blocked by the amount of JavaScript that needs to be downloaded, parsed, and executed by the browser before the website or web app becomes visible. In cases where network connectivity is a limited resource, this would render a website or application completely unresponsive for significant amounts of time. Consider the following example:

```
import React from "react";

const Home = () => {
  const handleClick = () => {
    alert("Button clicked!");
  };

  return (
    <div>
      <h1>Welcome to my website</h1>
      <p>Click the button below to see an alert</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  );
};

export default Home;
```

In this example, we are using a simple button click event to trigger an alert. However, this code still requires the entire React library to be downloaded and parsed by the browser before the button can be clicked. This can result in a slower initial page load, especially on slower devices and networks.

As of React 17, the bundle size of React and React-DOM is as follows:

- react (development): 152 KB
- react (production, minified): 44 KB
- react-dom (development): 3.3 MB
- react-dom (production, minified): 131 KB

These sizes are for the latest version of React at the time of writing and may vary depending on the version and configuration of React that you are using today. Regardless, it's important to understand from these data that the development version of React DOM is significantly larger than the production version, and the difference between the development and production versions of React itself is also substantial.

This means that even in production environments, our users have to download around 175 KB of JavaScript just for React alone (i.e, React + React DOM), before downloading, parsing, and executing the rest

of our application's code. This can result in a slower initial page load, especially on slower devices and networks, and potentially frustrated users. Moreover, because React essentially owns the DOM and we have no user interface without React in client-only applications, our users have no choice but to wait for React and React DOM to load *first* before the rest of our application does.

In contrast, a server-rendered application would stream rendered HTML to the client before any JavaScript downloads, enabling users to get meaningful content immediately. It would then load relevant JavaScript after the initial page renders, probably while the user is still orienting themselves with a user interface through a process called “hydration”. More on this in the coming sections.

Initially streaming rendered HTML and then hydrating the DOM with JavaScript allows users to interact with the application sooner, resulting in a better user experience: it is immediately available to the user without them having to wait for any extras—that they may or may not even need—to load.

Security

Client-side rendering can also have security issues, especially when dealing with sensitive data. This is because all of the application's code is downloaded to the client's browser, making it vulnerable to

attacks such as cross-site scripting (XSS) and cross-site request forgery (CSRF).

Consider the following example:

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

const Account = () => {
  const [balance, setBalance] = useState(100);

  const handleWithdrawal = (amount) => {
    setBalance(balance - amount);
  };

  return (
    <div>

      <h1>Account Balance: {balance}</h1>
      <button onClick={() => handleWithdrawal(10)}>Withdraw 10</button>
      <button onClick={() => handleWithdrawal(50)}>Withdraw 50</button>
      <button onClick={() => handleWithdrawal(100)}>Withdraw 100</button>
    </div>
  );
};

export default Account;
```

In this example, we have an account component that allows users to withdraw funds. However, if this component is rendered on the client-side, it could be vulnerable to CSRF attacks because the server and client do not have a shared common secret or contract between them. To speak poetically, the client and server don't know each other. This could allow an attacker to steal funds or manipulate the application's data.

If we used server rendering, we could mitigate these security issues by rendering the component on the server with a special secret token generated by the server and then sending HTML containing the secret token to the client. The client would then send this token back to the server that issued it, establishing a secure bidirectional contract. This would allow the server to verify that the request is coming from the correct client that it has pre-authorized and not an unknown one, which could possibly be a malicious attacker. Without server rendering, this becomes very hard if not impossible.

The Rise of Server Rendering

For these reasons, server-side rendering (SSR) has emerged as a critical technique for improving the performance and user experience of web applications. With server rendering, applications can be optimized for speed and accessibility, resulting in faster load times, better

SEO, and improved user engagement. Server-side rendering enables applications to be rendered on the server and sent to the client as fully formed HTML pages.

Benefits of Server Rendering

Let's dive deeper into the benefits of server rendering. First and foremost, server rendering can significantly improve the speed and accessibility of web applications. By rendering applications on the server, developers can ensure that the HTML and CSS are optimized for performance, resulting in faster load times and better overall performance.

Server rendering can also improve the SEO of web applications. Because search engines are primarily concerned with HTML and text content, server-rendered pages are easier for search engines to crawl and index. This can lead to higher rankings and more traffic for web applications; especially for content-rich web applications like blogs.

Finally, server rendering can improve the overall user experience of web applications. Because server-rendered pages are fully formed HTML pages, they are accessible to all users, including those with slow or unreliable internet connections. This can result in higher engagement and retention rates for web applications. Server rendering

has been around for many years, but it has only recently gained widespread adoption among web developers. React, in particular, has been instrumental in popularizing server rendering. There are now several frameworks available for implementing server-rendered React applications, which we will eventually cover in our chapter on frameworks.

With SSR, the initial HTML of a page is generated on the server and sent to the client, allowing the browser to start rendering the page more quickly compared to client-side rendering (CSR), where the HTML is generated in the browser using JavaScript. This results in better perceived performance and faster time to first meaningful paint (FMP).

However, server-rendered HTML is static and lacks interactivity as it does not have any JavaScript initially loaded, and includes no event listeners or other dynamic functionality attached. To enable user interactions and other dynamic features, the static HTML must be “hydrated” with the necessary JavaScript code. Let’s understand the concept of hydration a little better.

Hydration

Hydration is a term used to describe the process of attaching event listeners and other JavaScript functionality to static HTML that is generated on the server and sent to the client. The goal of hydration is to enable a server-rendered application to become fully interactive after being loaded in the browser, providing users with a fast and smooth experience.

In a React application, hydration happens after a client downloads a server-rendered React application. Then, the following steps occur:

- **Loading the client bundle:** While the browser is rendering the static HTML, it also downloads and parses the JavaScript bundle that contains the application's code. This bundle includes the React components and any other code necessary for the application's functionality.
- **Attaching event listeners:** Once the JavaScript bundle is loaded, React "hydrates" the static HTML by attaching event listeners and other dynamic functionality to the DOM elements. This is typically done using the `ReactDOM.hydrate` function, which takes the root React component and the DOM container as arguments. Hydration essentially transforms the static HTML into a fully interactive React application.

After the hydration process is complete, the application is fully interactive and can respond to user input, fetch data, and update the DOM

as necessary.

During hydration, React matches the structure of the DOM elements in the static HTML to the structure defined by the React components via JSX. It is crucial that the structure generated by the React components matches the structure of the static HTML. If there is a mismatch, React will not be able to correctly attach event listeners and will not be aware of what React element directly maps to what DOM element, which would result in the application not behaving as expected.

By combining server-side rendering and hydration, developers can create web applications that load quickly and provide a smooth, interactive user experience.

Adding Server Rendering

If you have an existing client-only React app, you may be wondering how to add server rendering. Fortunately, it's relatively straightforward to add server rendering to an existing React app. One approach is to use a server rendering framework, such as Next.js or Remix. These frameworks provide built-in support for server rendering, allowing you to easily add server rendering to your React app.

The best way to add server rendering to your React app is to use a server rendering framework, since they abstract away a lot of the complexity around implementing server-side rendering while also steering you, the developer, towards a pit of success. Abstractions like this can however leave the more curious of us interested in understanding the underlying mechanisms wanting. If you're a curious person and are interested in how one would add server rendering to a client-only React app manually, or if you're interested in how frameworks do it, read on.

Manually Adding Server Rendering to a Client-Only React App

If you've got a client-only application, this is how you'd add server rendering to it. First, you'd create a `server.js` file in the root of your project. This file will contain the code for your server.

```
// server.js
const express = require("express");
const path = require("path");
const React = require("react");
const ReactDOMServer = require("react-dom/server");

const App = require("../src/App");
```

```
const app = express();

app.use(express.static(path.join(__dirname, "build")));

app.get("*", (req, res) => {
  const html = ReactDOMServer.renderToString(<App />);
  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>My React App</title>
      </head>
      <body>
        <div id="root">${html}</div>
        <script src="/static/js/main.js"></script>
      </body>
    </html>
  `);
});

app.listen(3000, () => {
  console.log("Server listening on port 3000");
});
```

In this example, we're using Express to create a server that serves static files from the `./build` directory and then render our React

app on the server. We're also using `ReactDOMServer` to render our React app to an HTML string and then inject it into the response sent to the client.

In this scenario, we're assuming our client-only React app has some type of `build` script that would output a client-only JavaScript bundle into a directory called `build` that we reference in the above snippet. This is important for hydration. With all these pieces in order, let's go ahead and start our server.

```
node server.js
```

Running this command should start our server on port 3000, and should output `Server listening on port 3000`.

With these steps, we now have a server-rendered React app that can be optimized for speed and accessibility. By taking this “peek under the hood” approach to server rendering, we gain a deeper understanding of how server rendering works and how it can benefit our React applications.

If we open a browser and visit <http://localhost:3000>, we should see a server-rendered application. We can confirm that it is in fact server rendered by viewing the source code of this page, which should reveal actual HTML markup instead of a blank document.

Server Rendering APIs in React

In the previous section, we manually added server rendering to a client-only React app using `Express` and `ReactDOMServer`. Specifically, we used `ReactDOMServer.renderToString()` to render our React app to an HTML string. This is the most basic way to add server rendering to a React app. However, there are other ways to add server rendering to React apps. Let's take a deeper look at server rendering APIs exposed by React and understand when and how to use them.

Let's consider the `renderToString` API in detail, exploring its usage, advantages, disadvantages, and when it is appropriate to use it in a React application. Specifically, let's look into:

- What it is
- How it works, and
- How it fits into our every day usage of React

To start with this, let's talk about what it is.

`renderToString`: What it is

`renderToString` is a server-side rendering API provided by React that enables you to render a React component into an HTML string

on the server. This API is synchronous and returns a fully rendered HTML string, which can then be sent to the client as a response.

`renderToString` is commonly used in server-rendered React applications to improve performance, SEO, and accessibility.

Usage

Using `renderToString` is relatively straightforward. First, you need to import the `renderToString` function from the `react-dom/server` package. Then, you can call the function with a React component as its argument, and it will return the fully rendered HTML as a string. Here's an example of using `renderToString` to render a simple React component:

```
import React from "react";
import { renderToString } from "react-dom/server";

function App() {
  return (
    <div>
      <h1>Hello, world!</h1>
      <p>This is a simple React app.</p>
    </div>
  );
}

const html = renderToString(<App />);
```

```
console.log(html);
```

In this example, we create a simple App component and call `renderToString` with the component as its argument. The function returns the fully rendered HTML, which can be sent to the client.

How it Works

This function traverses the tree of React elements, converts them to a string representation of real DOM elements, and finally outputs a string. It is synchronous and blocking, meaning it cannot be interrupted or paused. If a component tree from the root is many levels deep, it can require quite a bit of processing. Since a server typically services multiple clients, `renderToString` could be called for each client unless there's some type of cache preventing this, and quickly block the event loop and overload the system.

In terms of code, `renderToString` converts this:

```
React.createElement(  
  "section",  
  { id: "list" },  
  React.createElement("h1", {}, "This is my list"),  
  React.createElement(  
    "p",  
    {}  
  )  
)
```

```

    },
    "Isn't my list amazing? It contains amazing t
  ),
  React.createElement(
    "ul",
    {},
    amazingThings.map((t) => React.createElement(
    )
  );

```

to this:

```

<section id="list">
  <h1>This is my list!</h1>
  <p>Isn't my list amazing? It contains amazing t
  <ul>
    <li>Thing 1</li>
    <li>Thing 2</li>
    <li>Thing 3</li>
  </ul>
</section>

```

Because React is declarative and React elements are declarative abstractions, a tree of them can be turned into a tree of anything else—in this case, a tree of React elements is turned into a string-representation of a tree of HTML elements.

Advantages

There are several advantages to using `renderToString` for server-side rendering in a React application:

- **Simplicity:** The `renderToString` API is easy to use and doesn't require any additional setup or configuration. You can simply import the function, call it with a React component, and get the rendered HTML as a string.
- **SEO:** Server-side rendering with `renderToString` can help improve the search engine optimization (SEO) of your application. When search engine crawlers index your site, they can see the fully rendered HTML, making it easier for them to understand the content and structure of your site.
- **Accessibility:** `renderToString` can also improve the accessibility of your application. Users with slow internet connections or devices may have a better experience if they receive fully rendered HTML instead of waiting for client-side JavaScript to load and render the page.
- **Performance:** on the client side, React's rendering is bound by the client's CPU power—which is heavily varied between clients. On the server side, we are able to accurately control the resources of our machines that render React components, thereby giving us more control and predictability.

- **First Meaningful Paint:** By using server-side rendering with `renderToString`, you can send fully rendered HTML to the client, which can result in a faster first meaningful paint. This can lead to a better perceived performance for your users.

Disadvantages

While `renderToString` offers several advantages, it also has some downsides:

- **Performance:** One of the main disadvantages of `renderToString` is that it can be slow for large React applications. Because it is synchronous, it can block the event loop and make the server unresponsive. This can be especially problematic if you have a high-traffic application with many concurrent users.
- **Memory-intensive:** `renderToString` returns a fully rendered HTML string, which can be memory-intensive for large applications. This can lead to increased memory usage on your server and potentially slower response times, or a panic that kills the server process under heavy load.
- **Limited interactivity:** Since `renderToString` generates static HTML, any interactivity in your application must be handled by client-side JavaScript. This means that your users may experience a delay between when the site loads to when the site be-

comes interactive, depending on how long the JavaScript takes to be loaded, parsed, and executed.

- **Lack of streaming support:** `renderToString` does not support streaming, which means that the entire HTML string must be generated before it can be sent to the client. This can result in a slower time to first byte (TTFB) and a longer time for the client to start receiving the HTML. This limitation can be particularly problematic for large applications with lots of content, as the client must wait for the entire HTML string to be generated before any content can be displayed.

Alternatives

For larger applications or situations where the downsides of `renderToString` become problematic, React offers alternative APIs for server-side rendering, such as `renderToPipeableStream`. These APIs return a Node.js stream instead of a fully rendered HTML string, which can provide better performance and support for streaming. We will cover these more in the next section.

When to Use `renderToString`

Considering the advantages and disadvantages of `renderToString`, it can be a good fit for smaller React ap-

plications or in situations where simplicity and ease of use are more important than performance. For example, if you are building a small website or a blog with mostly static content, `renderToString` may be sufficient for your needs.

However, if your application is large and complex, or if you require better performance and streaming capabilities, you should consider using `renderToPipeableStream` instead.

Additionally, it's worth considering using a framework like Next.js or Gatsby for your React application, as they can handle server-side rendering and other optimizations out-of-the-box. These frameworks can make it easier to set up server-side rendering and provide better performance and developer experience compared to using `renderToString` directly.

In summary, `renderToString` is a server-side rendering API in React that allows you to render a React component into an HTML string on the server. It offers several advantages, including simplicity, improved SEO, and better accessibility. However, it also has some downsides, such as potentially slow performance for large applications, memory-intensive operation, and lack of streaming support.

When choosing whether to use `renderToString` or an alternative API like `renderToPipeableStream`, consider factors such as the size of your application, your server environment, and your team's expertise. By understanding the different server-side rendering APIs available in React, you can make an informed decision about which one is best suited for your application.

Ideally by now, we're familiar with what `renderToString` is, how it works, and how it fits into server rendering. Let's now dive into these 3 points of exploration (what it is, how it works, and how it fits) around `renderToString`'s bigger brother `renderToPipeableStream`.

`renderToPipeableStream`

`renderToPipeableStream` is a server-side rendering API introduced in React 18. It provides a more efficient and flexible way to render large React applications to a Node.js stream. It returns a stream that can be piped to a response object.

`renderToPipeableStream` provides more control over how the HTML is rendered and allows for better integration with other Node.js streams.

In addition, it fully supports React's concurrent features including Suspense, which unlocks better handling of asynchronous data fetching during server-side rendering. Because it is a stream, it is also

streamable over the network, where chunks of HTML can be asynchronously and cumulatively sent to clients over the network without blocking. This leads to faster time to first bite (TTFB) measures and generally better performance.

Let's dive deep into `renderToPipeableStream`, discussing its features, advantages, and use cases. We'll also provide code snippets and examples to help you better understand how to implement this API in your React applications.

How it Works

Similar to `renderToString`, `renderToPipeableStream` takes a declaratively-described tree of React elements and—instead of turning them into a string of HTML—turns the tree into a Node.js stream. A Node.js stream is a fundamental concept in the Node.js runtime environment that enables efficient data processing and manipulation. Streams provide a way to handle data incrementally, in chunks, rather than loading the entire data set into memory at once. This approach is particularly useful when dealing with large strings or data streams that cannot fit entirely in memory, or over the network.

Node.js Streams

At its core, a Node.js stream represents a flow of data between a source and a destination. It can be thought of as a pipeline through

which data flows, with various operations applied to transform or process the data along the way.

Node.js streams are categorized into four types based on their nature and direction of data flow:

1. **Readable Streams:** A readable stream represents a source of data from which you can read. It emits events like “data,” “end,” and “error.” Examples of readable streams include reading data from a file, receiving data from an HTTP request, or generating data using a custom generator.
2. **Writable Streams:** A writable stream represents a destination where you can write data. It provides methods like `write()` and `end()` to send data into the stream. Writable streams emit events like “drain” when the destination can handle more data and “error” when an error occurs during writing. Examples of writable streams include writing data to a file, sending data over a network socket, or piping data to another stream.
3. **Duplex Streams:** A duplex stream represents both a readable and writable stream simultaneously. It allows bidirectional data flow, meaning you can both read from and write to the stream. Duplex streams are commonly used for network sockets or communication channels where data needs to flow in both directions.
4. **Transform Streams:** A transform stream is a special type of duplex stream that performs data transformations while data flows

through it. It reads input data, processes it, and provides the processed data as output. Transform streams can be used to perform tasks such as compression, encryption, decompression, or data parsing.

Node.js streams operate using a combination of events and methods. Readable streams emit events such as “data” when new data is available, “end” when the stream has no more data to provide, and “error” when an error occurs during reading. Writable streams provide methods like `write()` to send data into the stream and `end()` to signal the end of writing.

One of the powerful features of Node.js streams is the ability to pipe data between streams. Piping allows you to connect the output of a readable stream directly to the input of a writable stream, creating a seamless flow of data. This greatly simplifies the process of handling data and reduces memory usage.

Streams in Node.js also support backpressure handling. Backpressure is a mechanism that allows streams to control the flow of data when the destination cannot process it as fast as it is being produced. When the writable stream is unable to handle data quickly enough, the readable stream will pause emitting “data” events, preventing data loss. Once the writable stream is ready to consume more data, it

emits a “drain” event, signaling the readable stream to resume emitting data.

Node.js provides a built-in `stream` module that offers a set of classes and utilities for working with streams. In addition to the core module, numerous third-party libraries extend the capabilities of streams or provide specialized stream functionality.

To summarize, Node.js streams are a powerful abstraction for handling data in a scalable and memory-efficient manner. By breaking data into manageable chunks and allowing incremental processing, streams enable efficient handling of large data sets, file I/O operations, network communication, and much more.

In React, the purpose of streaming React components to a stream is to enhance the time-to-first-byte (TTFB) performance of server-rendered applications. Instead of waiting for the entire HTML markup to be generated before sending it to the client, these methods enable the server to start sending chunks of the HTML response as they are ready, thus reducing the overall latency.

The `renderToPipeableStream` function is a part of the React’s experimental server renderer, which is designed to support streaming rendering of a React application to a Node.js stream. It’s a part of the new server renderer architecture called “Fizz”.

Without distracting from our context of server rendering too much, here's a simplified explanation of how it works:

1. **Creating a Request:** The function `renderToPipeableStream` takes as input the React elements to be rendered and an optional options object. It then creates a request object using a `createRequestImpl` function. This request object encapsulates the React elements, resources, response state, and format context.
2. **Starting the Work:** After creating the request, the `startWork` function is called with the request as an argument. This function initiates the rendering process. The rendering process is asynchronous and can be paused and resumed as needed, which is where Suspense comes in. If a component is wrapped in a Suspense boundary and it initiates some asynchronous operation (like data fetching), the rendering of that component (and possibly its siblings) can be “suspended” until the operation finishes.
3. **Returning a Pipeable Stream:** `renderToPipeableStream` then returns an object that includes a `pipe` method and an `abort` method. The `pipe` method is used to pipe the rendered output to a writable stream (like an HTTP response object in Node.js). The `abort` method can be used to cancel any pending I/O and put anything remaining into client-rendered mode.
4. **Piping to a Destination:** When the `pipe` method is called with a destination stream, it checks if the data has already started

flowing. If not, it sets `hasStartedFlowing` to `true` and calls the `startFlowing` function with the request and the destination. It also sets up handlers for the 'drain', 'error', and 'close' events of the destination stream.

5. **Handling Stream Events:** The 'drain' event handler calls `startFlowing` again to resume the flow of data when the destination stream is ready to receive more data. The 'error' and 'close' event handlers call the `abort` function to stop the rendering process if an error occurs in the destination stream or if the stream is closed prematurely.
6. **Aborting the Rendering:** The `abort` method on the returned object can be called with a reason to stop the rendering process. It calls the `abort` function from the 'react-server' module with the request and the reason.

The actual implementation of these functions involves more complex logic to handle things like progressive rendering, error handling, and integration with the rest of the React server renderer. The code for these functions can be found in the 'react-server' and 'react-dom' packages of the React source code.

Features of `renderToPipeableStream`

- **Streaming:** `renderToPipeableStream` returns a pipeable Node.js stream, which can be piped to a response object. This

allows the server to start sending the HTML to the client before the entire page is rendered, providing a faster user experience and better performance for large applications.

- **Flexibility:** `renderToPipeableStream` offers more control over how the HTML is rendered. It can be easily integrated with other Node.js streams, allowing developers to customize the rendering pipeline and create more efficient server-side rendering solutions.
- **Suspense support:** `renderToPipeableStream` fully supports React's concurrent features, including Suspense. This allows developers to manage asynchronous data fetching more effectively during server-side rendering, ensuring that data-dependent components are only rendered once the necessary data is available.
- **Advanced API:** Although `renderToPipeableStream` is a more advanced API compared to `renderToString`, it enables developers to create more efficient and powerful server-side rendering solutions, particularly for large and complex React applications.

How it Fits

Let's take a look at some code that illustrates the benefits of this API. We have an application that displays a list of dog breeds. The list is

populated by fetching data from an API endpoint. The application is rendered on the server using `renderToPipeableStream` and then sent to the client. Let's start by looking at our dog list component:

```
// ./src/DogBreeds.jsx

const dogResource = createResource(
  fetch("https://dog.ceo/api/breeds/list/all")
    .then((r) => r.json())
    .then((r) => Object.keys(r.message))
);

function DogBreeds() {
  return (
    <ul>

      <Suspense fallback="Loading...">
        {dogResource.read().map((profile) => (
          <li key={profile}>{profile}</li>
        ))}
      </Suspense>
    </ul>
  );
}

export default DogBreeds;
```

Now, let's look at our overall `App` that contains the `DogBreeds` component:

```
// src/App.js
import React, { Suspense } from "react";

const UserProfile = React.lazy(() => import("./DogBreeds"));

function App() {
  return (
    <div>
      <h1>Dog Breeds</h1>
      <Suspense fallback=<div>Loading Dog Breeds</div>>
        <UserProfile />
      </Suspense>
    </div>
  );
}

export default App;
```

Notice, we're using `React.lazy` here as mentioned in prior chapters, just so we have another suspense boundary to demonstrate how `renderToPipeableStream` handles `Suspense`. Okay, let's tie this all together with an Express server.

```

// server.js
import express from "express";
import React from "react";
import { renderToPipeableStream } from "react-dom";
import App from "./App.jsx";

const app = express();

app.use(express.static("build"));

app.get("/", async (req, res) => {
  const htmlStart = `
    <!DOCTYPE html>
    <html lang="en">
      <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>React Suspense with renderToPipeableStream</title>
      </head>
      <body>
        <div id="root">
  `;

  res.write(htmlStart);

  const { pipe } = renderToPipeableStream(<App />
    onShellReady: () => {
      pipe(res);
    }
  );
});

```

```
        pipe(res),
      },
    });
  });

  app.listen(3000, () => {
    console.log("Server is listening on port 3000");
  });
```

What we're doing in the above code snippet is responding to a request with a stream of HTML. We're using `renderToPipeableStream` to render our `App` component to a stream, and then piping that stream to our response object. We're also using the `onShellReady` option to pipe the stream to the response object once the shell is ready. The shell is the HTML that is rendered before the React application is hydrated, and before `Suspense` is resolved. In our case, the shell is the HTML that is rendered before the dog breeds are fetched from the API. Let's take a look at what happens when we run this code.

If we visit `http://localhost:3000`, we get a page with a heading "Dog Breeds", and our suspense fallback "Loading Dog Breeds...". This is the shell that is rendered before the dog breeds are fetched from the API. The really cool thing is—even if we don't include React on the client side in our HTML and hydrate the page, the `Suspense` fallback is replaced with the actual dog breeds once they are fetched

from the API. This swapping of DOM when data becomes available happens entirely from the server side, without client-side React!

Let's understand how this works in a bit more detail.

Warning: *we are about to dive deep into React implementation details here that are quite likely to change over time. The point of this exercise (and this book) is not to obsess over single implementation details, but instead of understand the underlying mechanism so we can learn and reason about React better. This isn't required to use React, but understanding the mechanism can give us hints and practical tools to use in our day-to-day working with React. With that, let's move forward.*

When we visit `http://localhost:3000`, the server responds with the shell HTML, which includes the heading “Dog Breeds” and the suspense fallback “Loading Dog Breeds...”. This HTML looks like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-v
    <title>React Suspense with renderToPipeableSt
  </head>
  <body>
```

body-

```
<div id="root">
  <div>
    <h1>User Profiles</h1>
    <!--$?--><template id="B:0"></template>
    <div>Loading user profiles...</div>
    <!--/$?-->
  </div>
  <div hidden id="S:0">
    <ul>
      <!--$-->
      <li>affenpinscher</li>
      <li>african</li>
      <li>airedale</li>
      [...]
      <!--/$-->
    </ul>
  </div>
  <script>
    function $RC(a, b) {
      a = document.getElementById(a);
      b = document.getElementById(b);
      b.parentNode.removeChild(b);
      if (a) {
        a = a.previousSibling;
        var f = a.parentNode,
            c = a.nextSibling,
            e = 0;
        do {
```

```

        if (c && 8 === c.nodeType) {
            var d = c.data;

            if ("/$" === d)
                if (0 === e) break;
                else e--;
            else ("$" !== d && "$?" !== d &&
        }
        d = c.nextSibling;
        f.removeChild(c);
        c = d;
    } while (c);
    for (; b.firstChild; ) f.insertBefore
    a.data = "$";
    a._reactRetry && a._reactRetry();
    }
    }
    $RC("B:0", "S:0");
</script>
</div>
</body>
</html>

```

What we see here is quite interesting:

1. There's a `<template>` element with a generated ID (`B:0`) in this case and some HTML comments. The HTML comments are

used to mark the start and end of the shell. These are markers or “holes” where resolved data will go once Suspense is resolved.

2. There’s also a `<script>` element. This `<script>` tag contains a function called `$RC` that is used to replace the shell with the actual content. The `$RC` function takes two arguments: the ID of the `<template>` element that contains the marker, and the ID of the `<div>` element that contains the fallback. The function then fills the marker with rendered UI after data is available, while removing the fallback.

It’s pretty unfortunate that this function is minified even in development mode, but let’s try to unminify it and understand what it does. If we do, this is what we observe:

```
function reactComponentCleanup(reactMarkerId, siblingId) {
  let reactMarker = document.getElementById(reactMarkerId);
  let sibling = document.getElementById(siblingId);
  sibling.parentNode.removeChild(sibling);

  if (reactMarker) {
    reactMarker = reactMarker.previousSibling;
    let parentNode = reactMarker.parentNode,
        nextSibling = reactMarker.nextSibling,
        nestedLevel = 0;

    do {
      if (nextSibling && 8 === nextSibling.nodeType) {
```

```

        let nodeData = nextSibling.data;
        if ("/$" === nodeData) {
            if (0 === nestedLevel) {
                break;
            } else {
                nestedLevel--;
            }
        } else if ("$" !== nodeData && "$?" !== nodeData) {
            nestedLevel++;
        }
    }
    let nextNode = nextSibling.nextSibling;
    parentNode.removeChild(nextSibling);
    nextSibling = nextNode;
} while (nextSibling);

while (sibling.firstChild) {
    parentNode.insertBefore(sibling.firstChild, sibling);
}

reactMarker.data = "$";
reactMarker._reactRetry && reactMarker._react
}
}

reactComponentCleanup("B:0", "S:0");

```

Let's break this down further.

The function takes two arguments: `reactMarkerId` and `siblingId`. Effectively, the marker is hole where data will go once its available, and the sibling is the Suspense fallback.

The function then removes the sibling element (the fallback) from the DOM using the `removeChild` method on its parent node when data is available.

If the `reactMarker` element exists, the function runs. It sets the `reactMarker` variable to the previous sibling of the current `reactMarker` element. The function also initializes variables `parentNode`, `nextSibling`, and `nestedLevel`.

A `do...while` loop is used to traverse the DOM tree, starting with the `nextSibling` element. The loop continues as long as the `nextSibling` element exists. Inside the loop, the function checks whether the `nextSibling` element is a comment node (indicated by a `nodeType` value of `8`).

a. If the `nextSibling` element is a comment node, the function inspects its data (i.e., the text content of the comment). It checks whether the data is equal to `"/$"`, which signifies the end of a nested structure. If the `nestedLevel` value is `0`, the loop breaks, indicating that the desired end of the structure has been reached. If the

nestedLevel value is not 0 , it means that the current `"/$"` comment node is part of a nested structure, and the `nestedLevel` value is decremented.

b. If the comment node data is not equal to `"/$"` , the function checks whether it is equal to `"$"` , `"$?"` , or `"$!"` . These values indicate the beginning of a new nested structure. If any of these values are encountered, the `nestedLevel` value is incremented.

During each iteration of the loop, the `nextSibling` element is removed from the DOM using the `removeChild` method on its parent node. The loop continues with the next sibling element in the DOM tree.

Once the loop has completed, the function moves all child elements of the sibling element to the location immediately before the `nextSibling` element in the DOM tree using the `insertBefore` method. This process effectively restructures the DOM around the `reactMarker` element.

The function then sets the data of the `reactMarker` element to `"$"` , which is likely used to mark the component for future processing or reference. If a `reactRetry` property exists on the `reactMarker` element and it is a function, the function invokes this method.

In summary, this function appears to perform cleanup and restructuring of the DOM elements related to a React component. It uses comment nodes with specific data values to determine the structure of the component and manipulates the DOM accordingly. Since this is inlined in our HTML from the server, we can stream data like this using `renderToPipeableStream` and have the browser render the UI as it becomes available without even including React in the browser bundle or hydrating.

Thus, `renderToPipeableStream` gives us quite a bit more control and power compared to `renderToString` when server rendering.

Don't Roll Your Own

Creating a custom server rendering implementation for a React application can be a challenging and time-consuming task. While React does provide some APIs for server rendering, building a custom solution from scratch can lead to various issues and inefficiencies. In this section, we'll explore the reasons why it's better to rely on established frameworks like Next.js and Remix, rather than building your own server rendering solution.

- **Handling edge cases and complexities:** React applications can become quite complex, and implementing server rendering requires addressing various edge cases and complexities. These can include handling asynchronous data fetching, code splitting, and managing various React lifecycle events. By using a framework like Next.js or Remix, you can avoid the need to handle these complexities yourself, as these frameworks have built-in solutions for many common edge cases.

One such edge case is security. As the server processes numerous client requests, it's crucial to ensure that sensitive data from one client doesn't inadvertently leak to another. This is where frameworks like Next.js, Remix, and Gatsby can provide invaluable assistance in handling these concerns. Imagine a scenario where client A accesses the server, and their data is cached by the server. If the server accidentally serves this cached data to client B, sensitive information could be exposed.

Consider the following example:

```
// server.js
const express = require("express");
const app = express();
const cachedUserData = null;

app.get("/user/:userId", (req, res) => {
  const { userId } = req.params;
```

```
    if (cachedUserData) {  
        return res.json(cachedUserData);  
    }  
  
    // Fetch user data from a database or another source  
  
    const userData = fetchUserData(userId);  
  
    dataCache = userData;  
    res.json(userData);  
});  
  
app.listen(3000, () => {  
    console.log("Server listening on port 3000")  
});
```

In this example, the server caches user data in the `dataCache` object. If a race condition occurs, there is a risk that one client's data could be served to another client. This issue can lead to unauthorized access to sensitive information and serious security breaches.

If we roll our own, the risk of human error is ever present. If we lean on frameworks built by large communities, this risk is mitigated. These frameworks are designed with security in mind and ensure that sensitive data is handled properly. They prevent po-

tential data leakage scenarios by using secure and isolated data fetching methods

- **Performance optimizations:** Frameworks like Next.js and Remix come with numerous performance optimizations out of the box. These optimizations can include automatic code splitting, server rendering, and caching. Building a custom server rendering solution might not include these optimizations by default, and implementing them can be a challenging and time-consuming task.

```
// Example of automatic code splitting with Next.js
import dynamic from "next/dynamic";

const DynamicComponent = dynamic(() =>
  import("../components/DynamicComponent")
);

function Page() {
  return (
    <div>
      <DynamicComponent />
    </div>
  );
}

export default Page;
```

- **Developer experience and productivity:** Building a custom server rendering implementation can be a complex and time-consuming endeavor. By using a framework like Next.js or Remix, developers can focus on building features and functionality for their application instead of worrying about the underlying server rendering infrastructure. This can lead to increased productivity and a better overall developer experience.

```
// Example of simplified file-based routing with  
// File: pages/blog/[slug].js
```

```
function BlogPost({ post }) {  
  return (  
    <div>  
      <h1>{post.title}</h1>  
      <div>{post.content}</div>  
    </div>  
  );  
}
```

```
export async function getStaticProps({ params })  
  const post = await getPostBySlug(params.slug);  
  return { props: { post } };  
}
```

```
export async function getStaticPaths() {  
  const slugs = await getAllPostSlugs();
```

```
return {  
  paths: slugs.map((slug) => ({ params: { slug  
    fallback: false,  
  };  
});  
}  
  
export default BlogPost;
```

- **Community support and ecosystem:** Established frameworks like Next.js and Remix have large and active communities, which can provide invaluable support and resources. By using a well-supported framework, you can leverage the collective knowledge and experience of the community, access a vast array of plugins and integrations, and benefit from regular updates and improvements.
- **Best practices and conventions:** Using a framework like Next.js or Remix can help enforce best practices and conventions in your project. These frameworks have been designed with best practices in mind, and by following their conventions, you can ensure that your application is built on a solid foundation.

```
// Example of best practices with Remix  
// File: routes/posts/$postId.tsx  
  
import { useParams } from "react-router-dom";
```

```
import { useLoaderData } from "@remix-run/react";

export function loader({ params }) {
  return fetchPost(params.postId);
}

function Post() {
  const { postId } = useParams();
  const post = useLoaderData();

  return (
    <div>
      <h1>{post.title}</h1>
      <div>{post.content}</div>
    </div>
  );
}

export default Post;
```

Considering the benefits and optimizations provided by established frameworks like Next.js and Remix, it becomes evident that building a custom server rendering solution for a React application is not an ideal approach. By leveraging these frameworks, you can save development time, ensure best practices are followed, and benefit from the

ongoing improvements and support provided by their respective communities.

Some would say these frameworks are “opinionated”, as in that they adhere to a specific set of conventions or best practices. These conventions are usually prescribed by the framework’s creators or the broader community, and they guide developers in structuring their projects, writing code, and handling various aspects of application development.

Opinionated frameworks can save developers time and effort by providing a well-defined structure, established patterns, and sensible defaults. They often include a set of tools and libraries that work well together, promoting consistency and enabling developers to quickly get up and running with their projects.

Conventions in Frameworks

Conventions are the rules and guidelines that dictate the organization, structure, and coding practices within a framework. They are designed to promote consistency, improve code readability, and make it easier for developers to understand and maintain their projects.

Some common conventions found in opinionated frameworks include:

- **Directory Structure:** Opinionated frameworks often prescribe a specific directory structure for organizing your application's files and folders. This structure makes it easier for developers to locate specific components, assets, and other files, and it ensures that the application's organization is consistent across projects using the same framework.

An example of this is the `./pages` or `./app` directory in Next.js. This directory is used to store all of the application's pages, and they are then automatically mapped to routes on the server. By following this convention, developers can quickly and easily add new pages to their application without having to manually configure routing.

- **Component Naming and Structure:** Conventions around component naming and structure encourage developers to write modular, reusable code that adheres to best practices. For example, a framework might dictate that components should be named using PascalCase and organized in a specific way to promote code readability and maintainability.
- **Coding Style:** Opinionated frameworks typically recommend a specific coding style or provide built-in linting and formatting tools to ensure consistency across the project. By adhering to a common coding style, developers can more easily read and understand each other's code, making collaboration and maintenance more efficient.

- **Routing and Navigation:** Opinionated frameworks often include built-in support for client-side routing and navigation, making it easier to create single-page applications (SPAs) with complex routing requirements. By following the framework's conventions, developers can implement routing and navigation more quickly and with fewer potential pitfalls.

Benefits of Opinionated Frameworks

There are several benefits to using an opinionated framework for your projects, including:

- **Faster Development:** With predefined conventions and best practices in place, developers can spend less time setting up their projects and more time focusing on their application's core functionality. This streamlined development process can help teams get their applications up and running more quickly.
- **Consistency:** By adhering to a specific set of conventions, developers can create more consistent and maintainable codebases. This consistency makes it easier for new developers to join a project and quickly understand its structure and coding practices.
- **Improved Code Quality:** Opinionated frameworks often include tools and features designed to enforce best practices and improve code quality. By following the framework's conventions,

developers can create more reliable, maintainable, and efficient applications.

- **Easier Collaboration:** When all developers on a team follow the same set of conventions, it becomes easier to collaborate on projects and share code. This consistency can help reduce the risk of bugs and other issues caused by miscommunication or differences in coding styles.

This also makes contributing to open source more approachable.

- **Less Decision Fatigue:** With an opinionated framework, many decisions about project structure, coding practices, and tooling are already made for you. This can help reduce decision fatigue and allow developers

Conclusion

In conclusion, server-side rendering (SSR) and hydration are powerful techniques that can significantly improve the performance, user experience, and SEO of web applications. React provides a rich set of APIs for server rendering, such as `renderToString` and `renderToPipeableStream`, each with its own strengths and trade-offs. By understanding these APIs and selecting the right one based on factors such as application size, server environment, and developer experience, you can optimize your React application for both server and client-side performance.

As we've seen throughout this chapter, `renderToString` is a simple and straightforward API for server rendering that is suitable for smaller applications. However, it may not be the most efficient option for larger applications due to its synchronous nature and potential to block the event loop. On the other hand, `renderToPipeableStream` is a more advanced and flexible API that allows for better control over the rendering process and improved integration with other Node.js streams, making it a more suitable choice for larger applications.

Chapter Review

Now that you've gained a solid understanding of server-side rendering and hydration in React, it's time to test your knowledge with some review questions. If you can confidently answer these, it's a good sign that you've got a solid understanding of mechanism in React and can comfortably move forward. If you cannot, we'd suggest reading through things a little more, although this will not hurt your experience as you continue through the book.

1. What is the main advantage of using server-side rendering in a React application?
2. How does hydration work in React, and why is it important?
3. What are the key differences between `renderToString` and `renderToPipeableStream` APIs in React?
4. How can you choose the right server rendering API for your React application?

Up Next

Once you've mastered server-side rendering and hydration, you're ready to explore even more advanced topics in React development. In the next chapter, we'll dive into "Asynchronous React." As web applications become more complex, handling asynchronous actions be-

comes increasingly important for creating smooth user experiences. We'll explore techniques like `Suspense`, `React.lazy`, and concurrent mode, which allow you to handle asynchronous data fetching, code splitting, and rendering more efficiently.

By learning how to leverage asynchronous React, you'll be able to create highly performant, scalable, and user-friendly applications that can handle complex data interactions with ease. So, stay tuned and get ready to level up your React skills as we continue our journey into the world of Asynchronous React!

About the Authors

Tejas Kumar has been writing React code since 2014 and has given multiple conference talks, workshops, and guest lectures on the topic. With his wealth of experience across the technical stack of multiple startups, Tejas has developed a deep understanding of React's core concepts and enjoys using it to encourage, equip, and empower others to write React apps fluently.