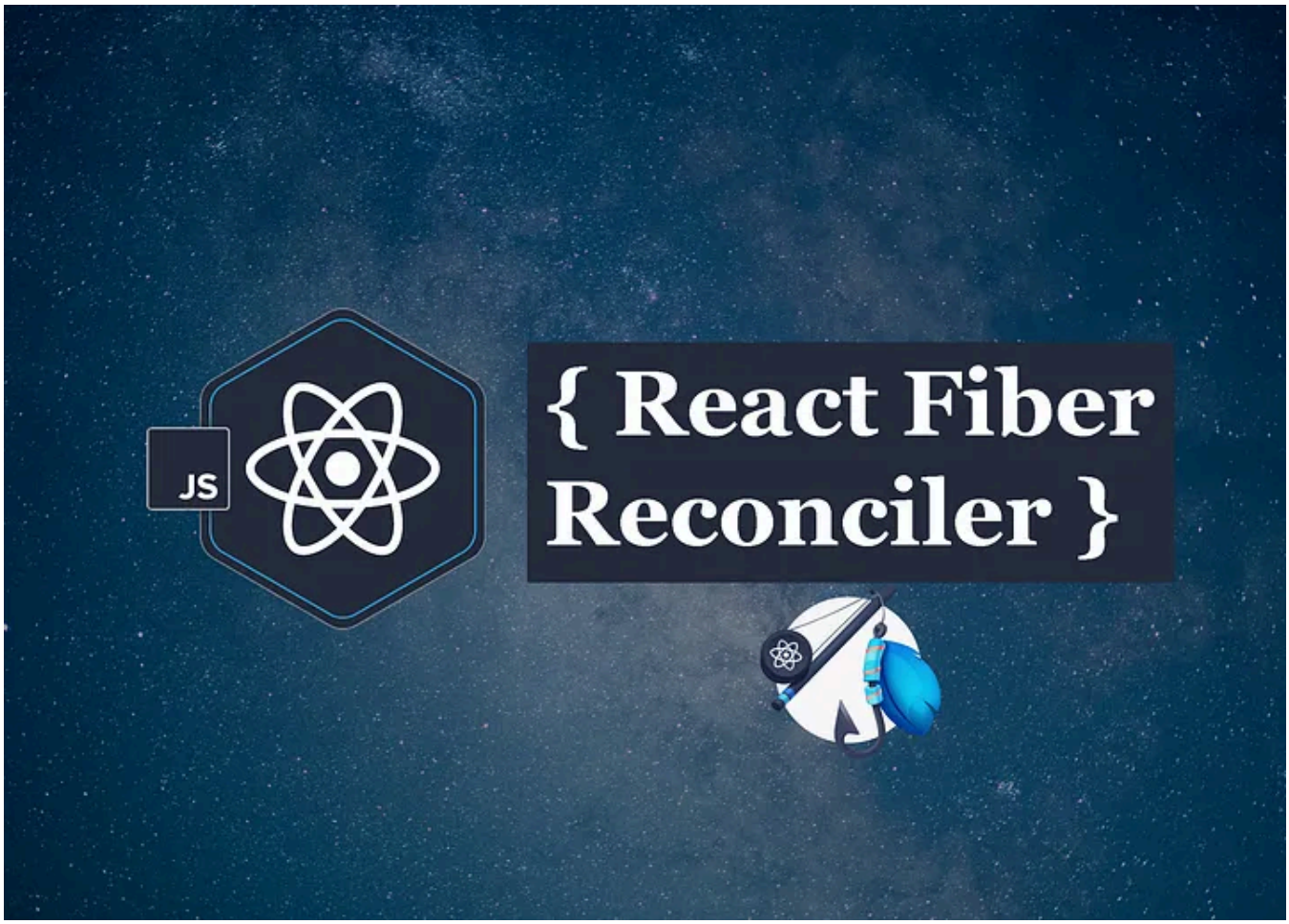
How does the React fiber reconciler work?

Max Shahdoost · Follow
7 min read · Jun 16, 2024



React Fiber Reconciler

In this article, I am going to deep dive into the React world and what is the core value proposition of React, what is React reconciler, how it worked before version 16, and how it works today!

1- What the heck is reconciliation?

First of all let's see what reconciliation means, according to the translation it means :

"The action of making one view or belief compatible with another"

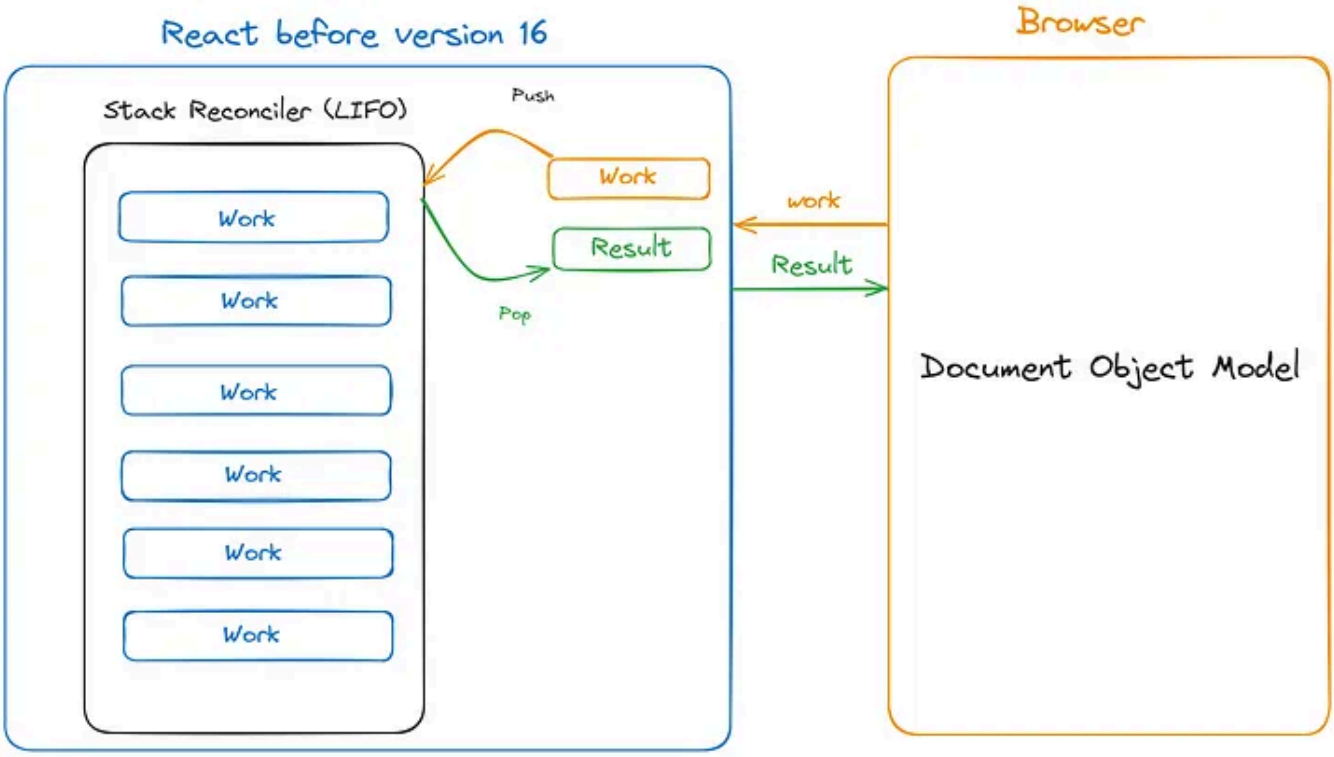
It represents the main purpose of React itself in single-page applications. Remember previously before SPAs we had trouble because of slow routing and the way browsers handled page transitions, to improve the speed of page transition in the browsers, React has introduced a new way of handling routing in web applications and it was the Virtual DOM.

Instead of using the native browser routing and navigation, we keep all of that in the JavaScript memory and use JavaScript to handle pages and routing for us, and as a result, it will become a lot faster.

To achieve that goal, the React team introduced the Virtual DOM and the stack reconciler to handle the virtual DOM tree in the memory and then apply the changes to the real DOM in a declarative way.

2- Stack Reconciler and the History lesson!

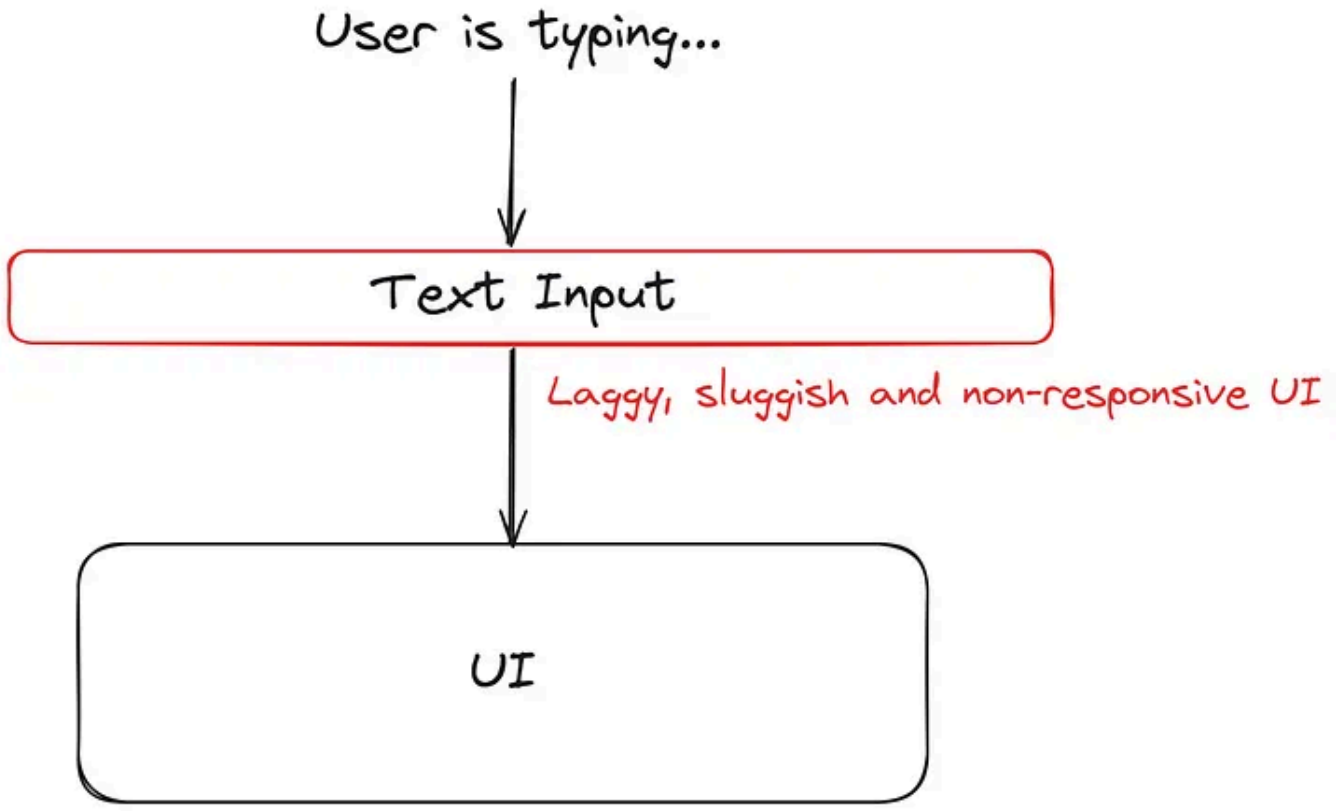
Alright, let's go back to the previous years before we had React Hooks introduced and see how React was building DOM trees at that time.
Below is a picture of a simplified workflow of React 15 and previous versions of React, in those versions, we had a reconciler called stack reconciler which was a LIFO data structure responsible for picking a work and returning the results just like how the JavaScript call stack works.



React Stack Reconciler before version 16

This approach was a breakthrough already but it had a lot of problems with itself. The main problem was that the stack reconciler was synchronous and sequential meaning there was no chance for it to handle multiple units of work at the same time in parallel or concurrently.

For example, assume the below interaction with the UI by a user:



The problem of non-responsive UI in stack reconciler

The problem of non-responsive UI in stack reconciler

As you can see, with the synchronous and sequential order of the stack reconciler, if a user wants to type something in a text input, the response to the user will be laggy and non-responsive because it is a high priority in the rendering order but there is no way we can tell the stack to handle this with higher priority. Another problem is that if any error happens in the middle of this process there is no way for us to find out where it happened and what is the stack trace which can be painful.

Now assume a large application with too many states and works to handle, it could become chaos and the user experience could be destroyed.

3- Fiber Reconciler, that's what we were looking for!

In React version 16 and above, the React team has introduced a new way of handling the units of work and virtual DOM tree using the new meta called Fiber Reconciler to tackle two main challenges:

1- Synchronous way of processing the units of work
2- Prioritizing and concurrency of the units of work

The current React Fiber Reconciler consists of many fiber nodes which are plain JavaScript objects with a lot of properties to handle their work.

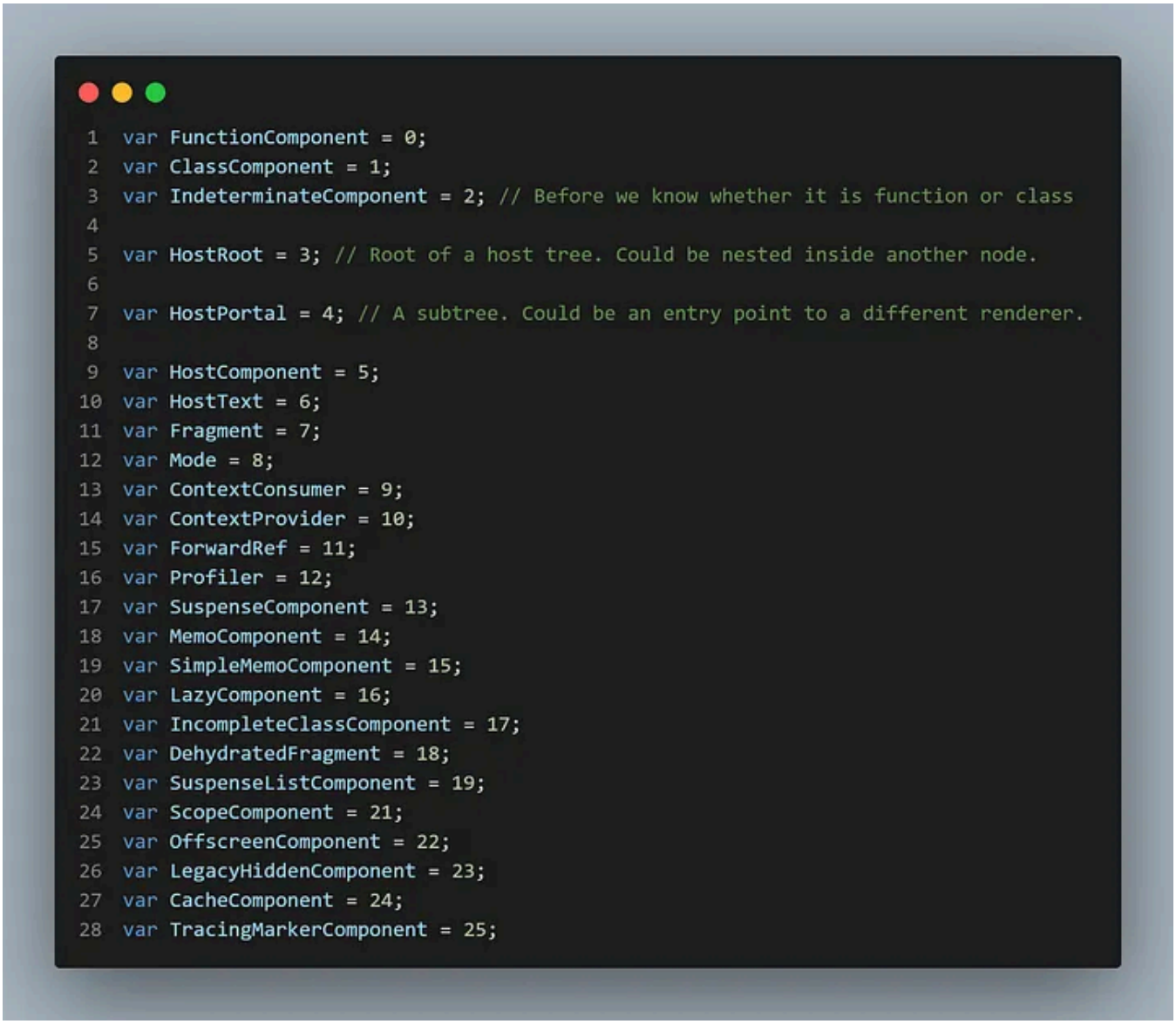
Fiber = { a JavaScript object with many properties OR unit of work }.

Fiber Reconciler = The current React reconciler based on Fiber objects or units of work.

What is a unit of work?

A unit of work in React can be a change in props or state or DOM updates, anything that can change the output for the screen.

The fiber has a relation of 1 to 1 with something whether a component instance, DOM node, etc. The type of something is stored in a tag in the fiber object. The possibilities of the types are:



```
var FunctionComponent = 0;
var ClassComponent = 1;
var IndeterminateComponent = 2; // Before we know whether it is function or class

var HostRoot = 3; // Root of a host tree. Could be nested inside another node.

var HostPortal = 4; // A subtree. Could be an entry point to a different renderer.

var HostComponent = 5;
var HostText = 6;
var Fragment = 7;
var Mode = 8;
var ContextConsumer = 9;
var ContextProvider = 10;
var ForwardRef = 11;
var Profiler = 12;
var SuspenseComponent = 13;
var MemoComponent = 14;
var SimpleMemoComponent = 15;
var LazyComponent = 16;
var IncompleteClassComponent = 17;
var DehydratedFragment = 18;
var SuspenseListComponent = 19;
var ScopeComponent = 21;
var OffscreenComponent = 22;
var LegacyHiddenComponent = 23;
var CacheComponent = 24;
var TracingMarkerComponent = 25;
```

Possibility of the type of “something” in a possible 1 to 1 relation with a fiber object.

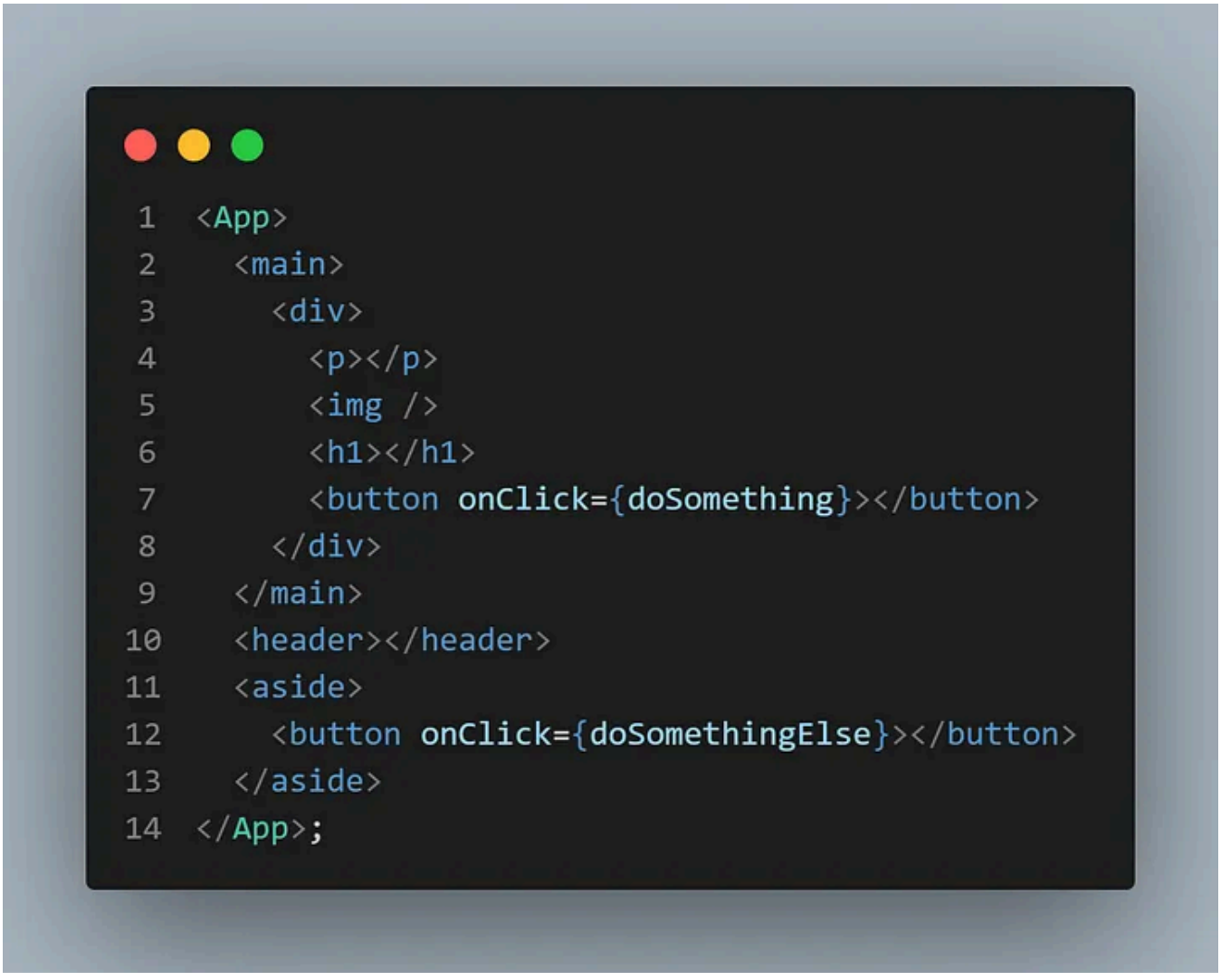
In the source code of the React-DOM library, you can find functions that are named like:

createFiberFromtText();
createFiberFromElement();
createFiberFromPortal();

It shows that Fibers can be created from almost a lot of options in the DOM and the React ecosystem. Now let’s find out how the new React Fiber Reconciler works!

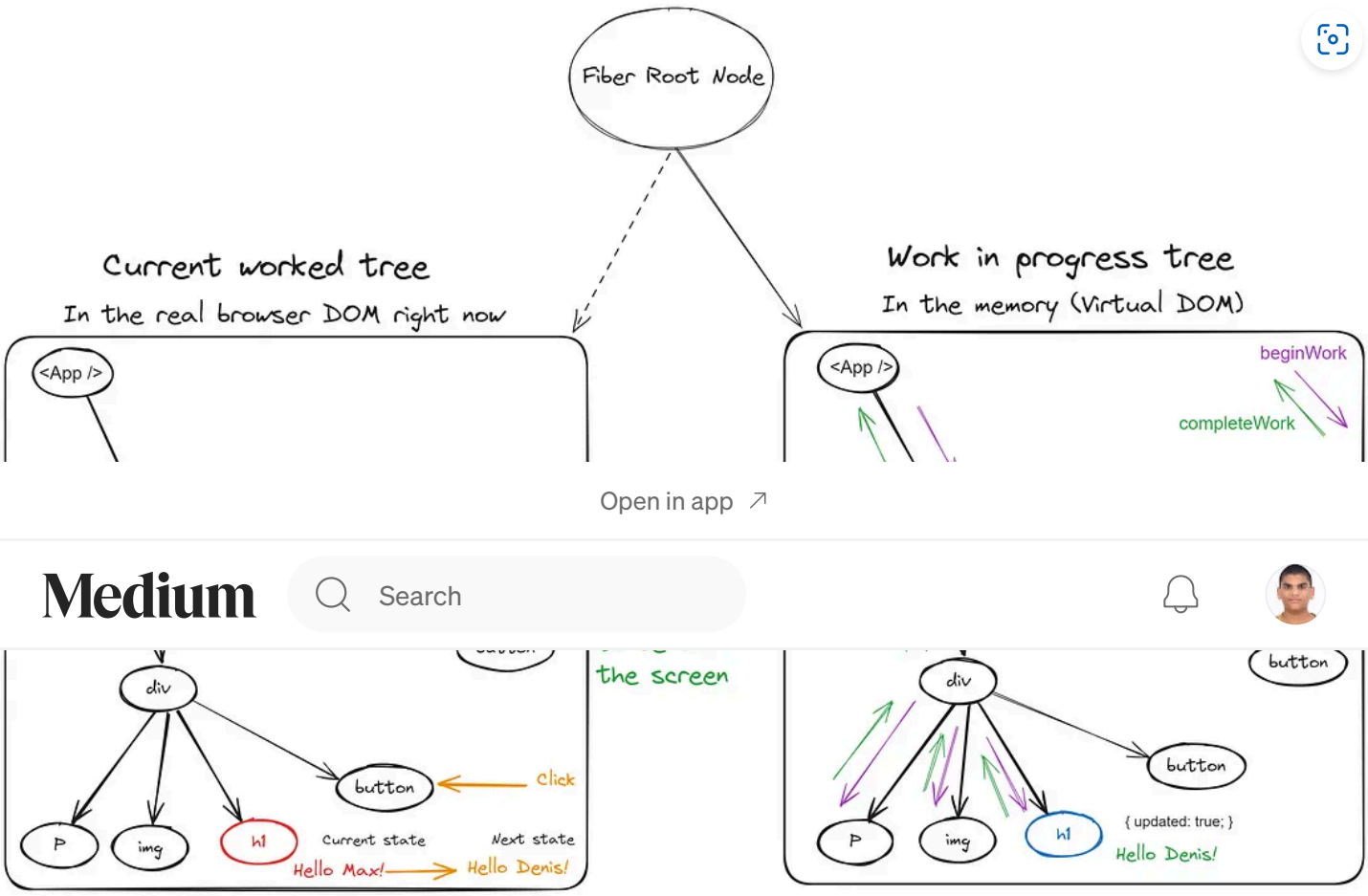
React is a declarative way of handling DOM manipulation, it means that we tell it what we want to be shown on the screen and it will do the heavy-duty work for us under the hood so that we can focus on the business logic and more importantly stuff that we need for our business.

The below is a simple example of a page:



Sample DOM tree that we would want to be shown on a screen

Now let's see how it is handled by the React fiber reconcile:



React Fiber Reconciliation Process in a Simplified Flow

As you can see in the above picture, React Fiber Reconciler constructs a tree of DOM elements in the memory (Virtual DOM) and holds a blueprint of the real DOM as well to work on the real DOM and render the updates on the Virtual DOM and then apply those changes to the real DOM.

This process has two steps:

1- Render phase (Asynchronous)
2- Commit phase (Syncronous)

When an update happens in your application, the Fiber reconciler starts the work with a function called beginWork which takes three parameters, the current tree which is on the real DOM, the work in progress tree, and the rendering lanes.

The process is a lot more complex but the overview of what is happening under the hood in the React reconciler is as the blow in a simple click on a button and change in the state example:

1- When the user clicks on the button the process will start by calling beginWork(currentTree, workInProgressTree, lanes) which will recursively start checking the tree from top to bottom going down on siblings and children nodes.

2- While there is work to be done meaning changes happened in the props or the states, it marks the nodes using an updated flag that the node needs to be updated.

3- When it is finished working on a fiber unit of work it marks the updates and finishes the work by calling the completeWork(currentTree, workInProgressTree, lanes) function which is responsible for going up in the working tree.

4- Complete work also constructs HTML elements tree to be shown in the real DOM based on updates off-screen in the memory.

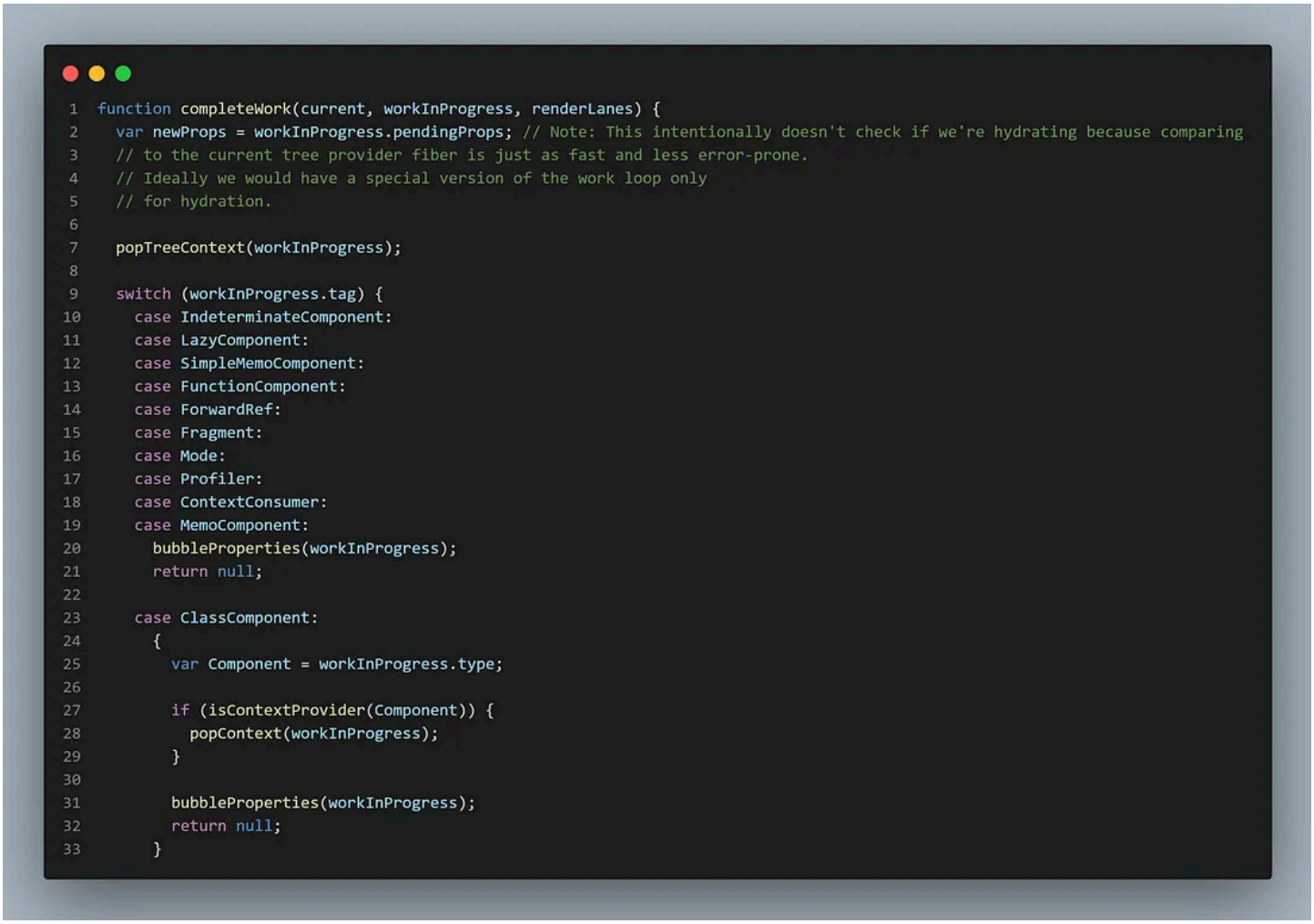
5- When everything is done the fiber has finished work and there is no more work to be done, it will commit all the newly constructed DOM trees to the real DOM and reflect the changes to be shown on the screen.

```
function beginWork(current, workInProgress, renderLanes) {
  {
    if (workInProgress._debugNeedsRemount && current !== null) {
      // This will restart the begin phase with a new fiber.
      return remountFiber(current, workInProgress, createFiberFromTypeAndProps(workInProgress.type, workInProgress.key, workInProgress.pendingProps, workInProgress._debugOwner || null, workInProgress.mode, workInProgress.lanes));
    }
  }

  if (current !== null) {
    var oldProps = current.memoizedProps;
    var newProps = workInProgress.pendingProps;

    if (oldProps !== newProps || hasContextChanged() || ( // Force a re-render if the implementation changed due to hot reload:
     workInProgress.type !== current.type )) {
      // If props or context changed, mark the fiber as having performed work.
      // This may be unset if the props are determined to be equal later (memo).
      didReceiveUpdate = true;
    } else {
```

The beginWork function in the React-DOM library



```
function completeWork(current, workInProgress, renderLanes) {
  var newProps = workInProgress.pendingProps; // Note: This intentionally doesn't check if we're hydrating because comparing
  // to the current tree provider fiber is just as fast and less error-prone.
  // Ideally we would have a special version of the work loop only
  // for hydration.

  popTreeContext(workInProgress);

  switch (workInProgress.tag) {
    case IndeterminateComponent:
    case LazyComponent:
    case SimpleMemoComponent:
    case FunctionComponent:
    case ForwardRef:
    case Fragment:
    case Mode:
    case Profiler:
    case ContextConsumer:
    case MemoComponent:
      bubbleProperties(workInProgress);
      return null;

    case ClassComponent:
      {
        var Component = workInProgress.type;

        if (isContextProvider(Component)) {
          popContext(workInProgress);
        }

        bubbleProperties(workInProgress);
        return null;
      }
```

The completeWork function in the React-DOM library

The rendering phase and working on the DOM tree in the memory is completely off-screen and asynchronous so if any update or interrupt comes up in the middle of a process, the process can wait or even a bail-out can happen to drop the process and start working on another process again, it can be prioritized, it can be delayed, it can be canceled, it can be parallelized to do multiple works concurrently at the same time. This is exactly the solution we were looking for to solve the problems of the stack reconciler previously.

By using the Fiber reconciler we can now have concurrent rendering, suspense features, and error boundaries to catch rendering phase errors and show a fallback to the user without letting the whole app crash at once!

How do effects work and apply in this process?

A very good question, the result of a fiber tree is not dependent only on itself, we also have a list of effects that may happen for example a network request, a

mutation in the real DOM, calling lifecycle methods, or anything happening outside of the React ecosystem that needs to be synced with React state.

In the commit phase, React will go through all the effects and apply them to component instances, the results are visible to the user and React does all that in a single pass.

```
function markLayoutEffectsStarted(lanes) {
  {
    if (injectedProfilingHooks !== null && typeof injectedProfilingHooks.markLayoutEffectsStarted === 'function') {
      injectedProfilingHooks.markLayoutEffectsStarted(lanes);
    }
  }
}
function markLayoutEffectsStopped() {
  {
    if (injectedProfilingHooks !== null && typeof injectedProfilingHooks.markLayoutEffectsStopped === 'function') {
      injectedProfilingHooks.markLayoutEffectsStopped();
    }
  }
}
function markPassiveEffectsStarted(lanes) {
  {
    if (injectedProfilingHooks !== null && typeof injectedProfilingHooks.markPassiveEffectsStarted === 'function') {
      injectedProfilingHooks.markPassiveEffectsStarted(lanes);
    }
  }
}
function markPassiveEffectsStopped() {
  {
    if (injectedProfilingHooks !== null && typeof injectedProfilingHooks.markPassiveEffectsStopped === 'function') {
      injectedProfilingHooks.markPassiveEffectsStopped();
    }
  }
}
```

React side effects in the reconciliation process

We are all over it now! wasn't that cool? knowing what is happening under the hood when you are using React empowers you to know the React ecosystem and how it handles the process and DOM behind the scenes.

I hope you have enjoyed it, if so please follow me for more articles in the future, peace, and happy coding.

Resources:

https://www.youtube.com/watch?v=0ympFIwQFJw

https://www.youtube.com/watch?v=rKk4XJYzSQA&t=9s

https://www.youtube.com/watch?v=Zan16X8VvGM

React Reactjs Reconciliation Dom JavaScript

Follow

Written by Max Shahdoost

314 Followers · 53 Following

A highly passionate and motivated Frontend Engineer with a good taste for re-usability, security, and developer experience.

Responses (1)

Kovidh Meduri

What are your thoughts?

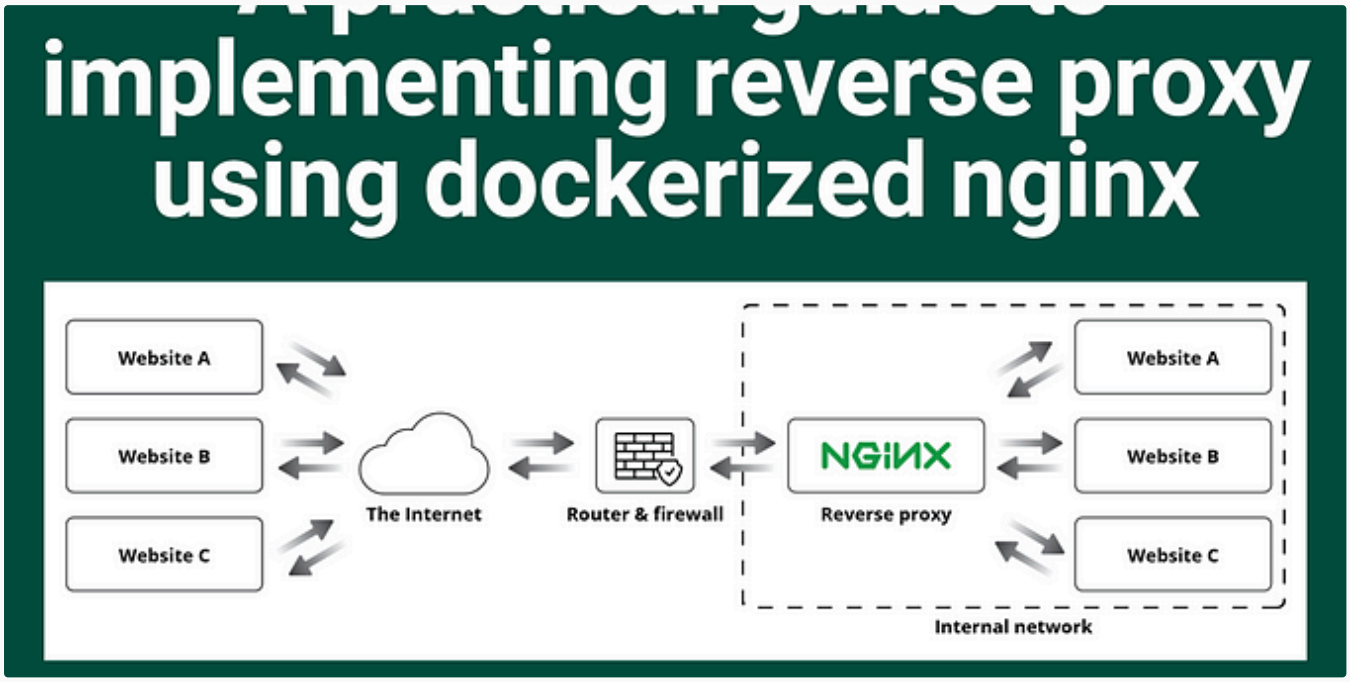
Anashesham
Jun 28, 2024

> What the heck is reconciliation?

nice titile

5 Reply

More from Max Shahdoost

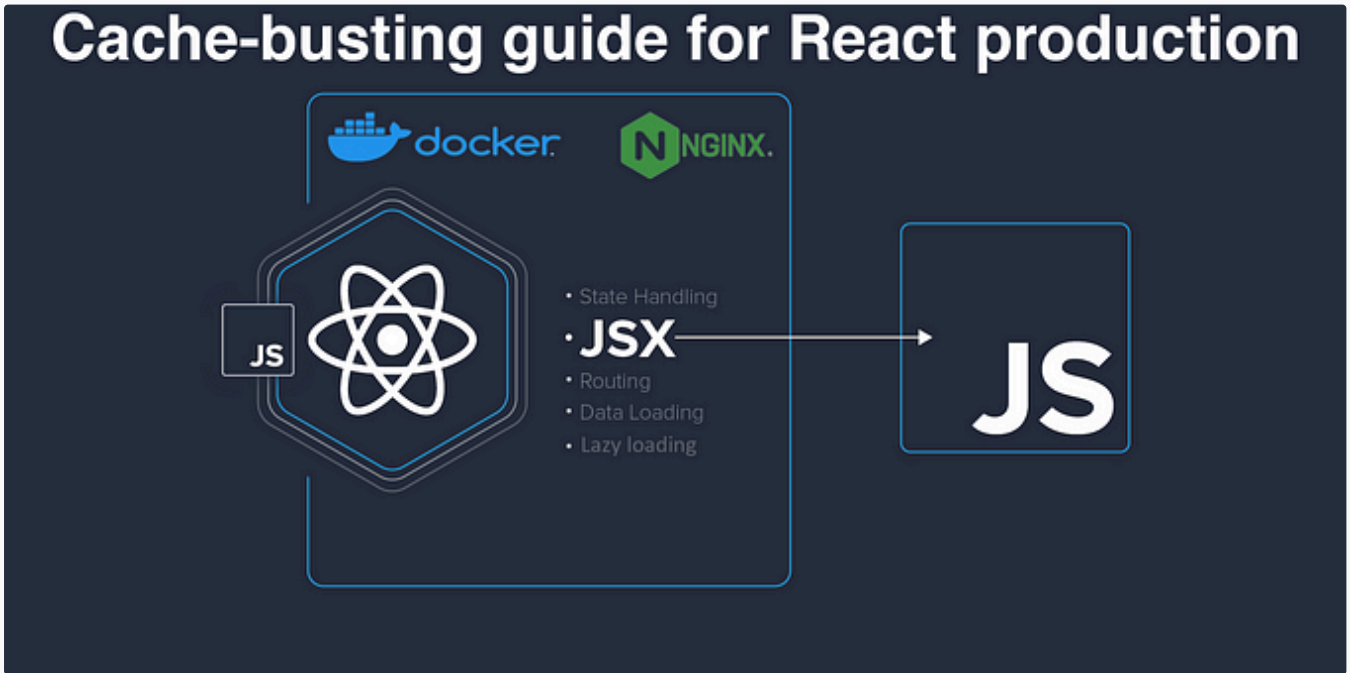


Max Shahdoost

A practical guide to implementing reverse proxy using dockerized nginx with multiple apps

In this article, we are going to cover a very common practice architecture used in many services called reverse proxy and we are going...

Jun 10, 2023 74 1



Max Shahdoost

The ultimate guide to cache busting for React production applications

NOTE: We are NOT going through route-based lazy loading since it needs a whole other topic for itself so if you are not familiar with...

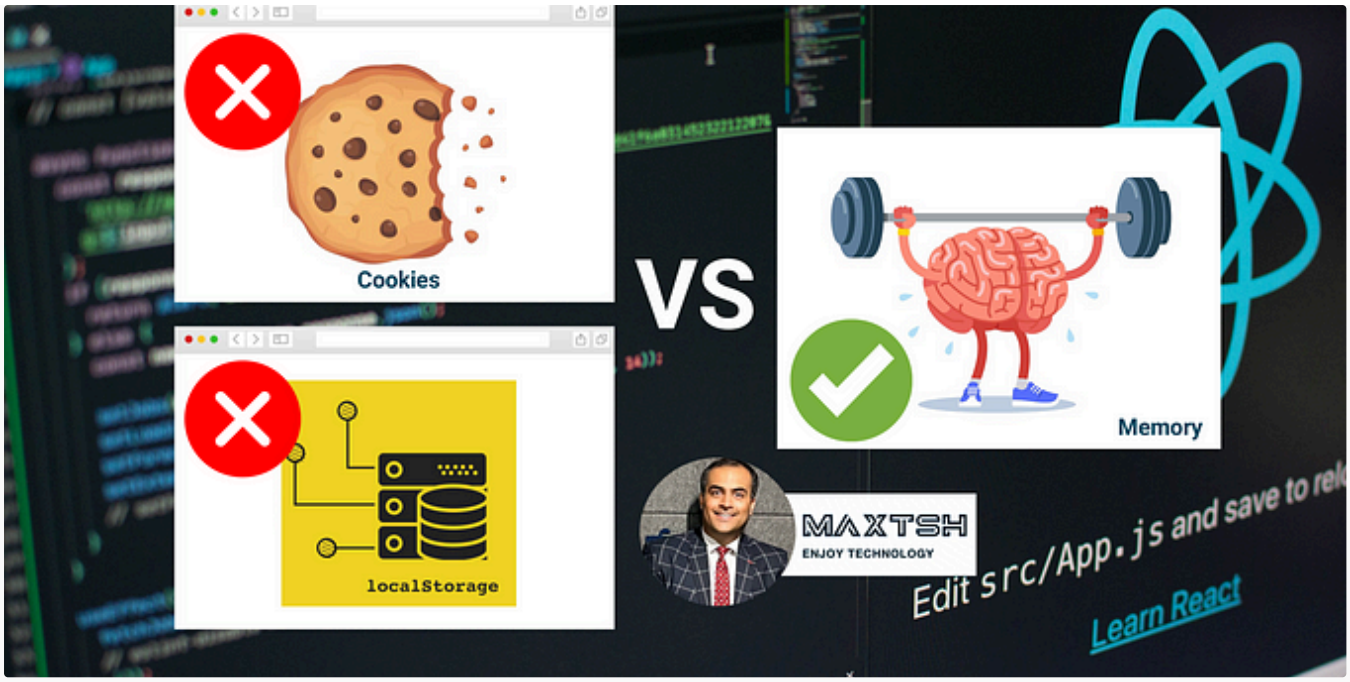
Jan 21, 2024 103 2

Max Shahdoost

Nginx reverse proxy security and performance best practices

About a month ago I published an article about a practical way to implement nginx reverse proxy but I've got feedback from some dudes that...

Jul 28, 2023 24 1



Max Shahdoost

Production-ready comprehensive anti-CSRF/XSS Reactjs client-side authentication using access and...

There are multiple ways of doing authentication in React.js Client-Side Rendering AKA CSR applications all over the internet and courses...

Jul 4, 2023 110 1

See all from Max Shahdoost

Recommended from Medium

Vinod Pal

How I Review Code As a Senior Developer For Better Results

I have been doing code reviews for quite some time and have become better at it. From my experience here I have compiled a list of...

Jan 25 1.4K 37

Mate Marschalko

18 Advanced React Techniques Every Senior Dev Needs to Know

As React applications grow more complex, the patterns that were “just fine” when you were starting out might start to feel limiting. Maybe...

Jan 6 926 19

Lists

Stories to Help You Grow as a Software Developer

19 stories · 1619 saves

General Coding Knowledge

20 stories · 1937 saves

Medium's Huge List of Publications Accepting Submissions

414 stories · 4652 saves

Generative AI Recommended Reading

52 stories · 1685 saves

Dzmitry Ihnatovich

How React Hooks Work Under the Hood: Recreating useState, useEffect, and useRef from Scratch

React's hooks API fundamentally changed the way developers interact with state and side effects in React components. But how do these hooks...

Nov 14, 2024 21 1

Ravi Sharma

Top 30 JavaScript Interview Questions and Answers for 2025

Prepare for your next JavaScript interview with confidence!

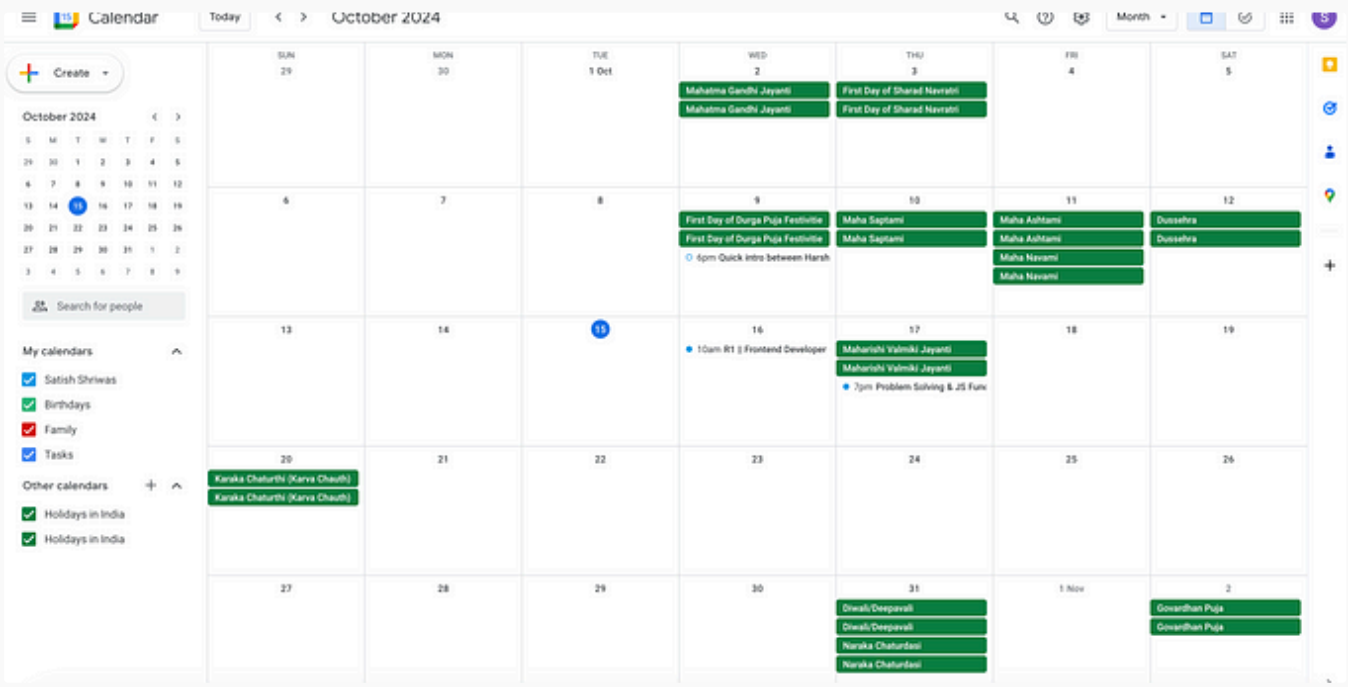
Jan 1, 2024 2K 25

Nikola Perišić

Don’t use React imports like this. Use Wrapper Pattern instead

While working on a real-life project, I encountered an inefficient React.js import strategy. Learn how to import the right way!

Feb 21 535 18



In Stackademic by Satish Shriwas

System Design: Google Calendar

Creating a calendar application like Google Calendar involves careful planning, architecture, and optimization. Let’s walk through the...

Oct 15, 2024 25 1

See more recommendations