Agenda

- Local State
- Props Drilling and Managing State with Context API
- Optimization Techniques in react

Local State

Any component is ReactJS majorly depends on its props and state to manage data. A component's state is private to it and is responsible for governir its behavior throughout its life. A state is nothing but a structure that records any data changes in a react application. It can be used for storing value form inputs, data from an API, etc. This local state-managed within a component can not be affected by other components.

Keep in mind that using local state in the context of React requires you to create your components using the ES6 classes which come with a construct function to instantiate the initial requirements of the component. Additionally, you have the option of using the useState Hook while creating function components.

In a component built with ES6 classes, whenever the state changes (only available through setState function), React triggers a re-render which essential for updating the state of the application. Here is an example:

```
import React from 'react';

Class FlowerShop extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      roses: 100
    }
    this.buyRose = this.buyRose.bind(this);
  }

  buyRose() {
    this.setState({
      roses: this.state.roses + 1
    })
  }

  render() {
    return (
      <div>
        <button
          onClick={ this.buyRose }>
          Buy Rose
        </button>
        { this.state.roses }
      </div>
    )
  }

}
```

Imagine that the above component acts like a flower shop that has its internal tracking system to see how many roses the store has at a given time. Th can work properly if the FlowerShop component is the only entity that should have access to this state. But imagine if this store decides to open a secor branch. In that case, the second flower shop will need access to the number of available roses as well (AKA this.state). Something that is not possib with the usage of local state.

So now we have realized a big disadvantage of the local state which is the shared state. On the other hand, if we want to keep track of an isolated sta of the component that will not be shared with other parts of the outside world (like UI state), then the local state will be a perfect tool for that use case.

Props Drilling

Dealing with state management in React applications can be a tricky thing, especially when data needs to be passed from a root component down deeply-nested components.

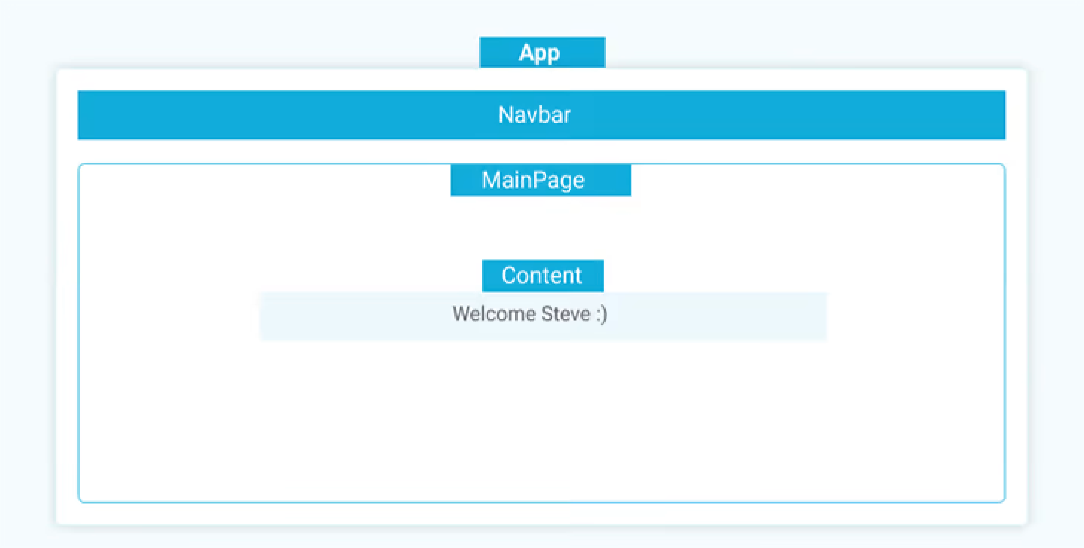
Prop drilling is the unofficial term for passing data through several nested children components, in a bid to deliver this data to a deeply-neste component. The problem with this approach is that most of the components through which this data is passed have no actual need for this data. They a simply used as mediums for transporting this data to its destination component.

This is where the term “drilling” comes in, as these components are forced to take in unrelated data and pass it to the next component, which in tu passes it, and so on, until it reaches its destination. This can cause major issues with component reusability and app performance, which we’ll expla later on.

For now, let’s look at an example set of circumstances that could lead to prop drilling.

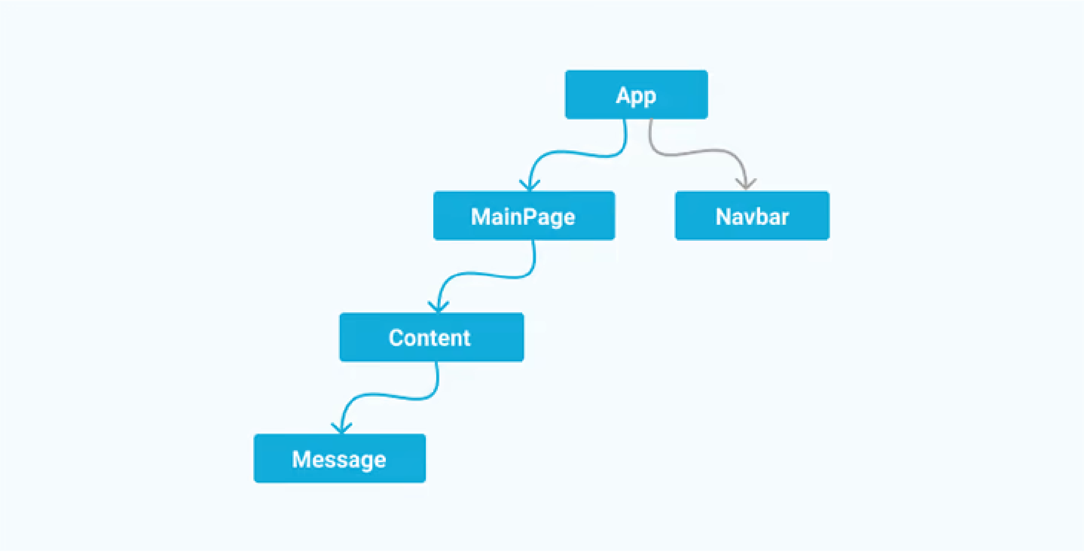
Building a deeply-nested app for prop drilling

Imagine for a second that we are building an app that welcomes a user by name when they log in. Below is the visual representation of the demo ap we’ll be looking at.



We won’t be covering the styling to keep our code minimal; this is just to provide a solid idea of what our app would look like.

Now, let’s look at the component hierarchy to understand the relationship between the components



As you can probably see now, the problem we have is that the `user` object that holds the user’s name is only available at the root component lev (**App**), whereas the component rendering the welcome message is nested deep within our app (**Message**). This means we somehow have to pas this `user` object down to the component that renders the welcome message.

The blue arrows represent the actual `user` object prop as it’s drilled down from the root **App** component, through several nested components, to tl actual **Message** component in need of it. It then finally renders the welcome message with the logged-in user’s name.

This is a typical case of prop drilling. This is where developers often resort to the Context API as a means of bypassing this supposed problem, witho giving much thought to the potential problems created therein.

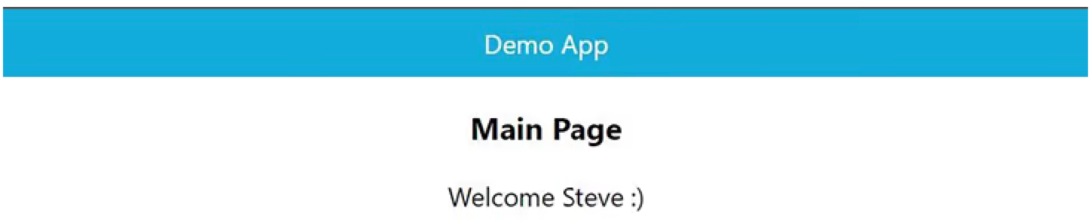
Now that we have a visual map of the project, let’s get our hands dirty with actual code.

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

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

// Navbar Component
function Navbar() {
  return <nav style={{ background: "#10ADDE", color: "#fff" }}>Demo App</nav>;
}

//MainPage Component
function MainPage({ user }) {
  return (
    <div>
      <h3>Main Page</h3>
      <Content user={user} />
    </div>
  );
}

// Content Component
function Content({ user }) {
  return (
    <div>
      <Message user={user} />
    </div>
  );
}

//Message Component
function Message({ user }) {
  return <p>Welcome {user.name}</p>;
}
```

Notice that, rather than splitting our components into different files and then importing each individual component, we put them all in the same file as the own, individual function components. We can now use them without any external imports.

Our resulting output would be:



Now that we have a basic working app, let's compare this solution to prop drilling by solving it once more, this time using the Context API.

Solving prop drilling by using the Context API

The Context API can be used to share data with multiple components, without having to pass data through props manually. For example, some us cases the Context API is ideal for: theming, user language, authentication, etc.

createContext

To start with the Context API, the first thing we need to do is create a context using the `createContext` function from React.

```
const UserContext = createContext(initialValue);
```

The `createContext` function accepts an initial value, but this initial value is not required.

After creating your context, that context now has two React components that are going to be used: `Provider` and `Consumer` .

Provider

The `Provider` component is going to be used to wrap the components that are going to have access to our context.

```
<UserContext.Provider value={{ user: "Steve" }}>
 ....
</UserContext.Provider>
```

The `Provider` component receives a prop called `value` , which can be accessed from all the components that are wrapped inside `Provider` , ar it will be responsible to grant access to the context data.

Consumer

After you wrap all the components that are going to need access to the context with the `Provider` component, you need to tell which component going to consume that data.

The `Consumer` component allows a React component to subscribe to the context changes. The component makes the data available using a rend prop.

```
<UserContext.Consumer>
 {value => /* render something based on the context value */}
</UserContext.Consumer>
```

useContext

You might have been using React Hooks for some time now, but if you don't know yet what React Hooks are and how they work, let me very brief explain them to you:

React Hooks allow us to manage state data inside functional components; now we don't need to create class components just to manage state data.

The `useContext` hook allows us to connect and consume a context. The `useContext` hook receives a single argument, which is the context that yc want to have access to.

```
const { user } = useContext(UserContext);
```

The `useContext` is way better and cleaner than the `Consumer` component—we can easily understand what's going on and increase tl maintainability of our application.

Let's now create an example with the Context API and the hook to see how it applies in a real-world application. We're going to use the same applicatic above:

Let's create a context and pass the `user` object to the context provider. We'll then go ahead and wrap our desired components with the conte provider, and access the state it holds inside the specific component that needs it.

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

//Creating a context
const UserContext = createContext();

function App() {
  return (
    <div>
      <Navbar />
      <UserContext.Provider value={{ user: "Steve" }}>
        <MainPage />
      </UserContext.Provider>
    </div>
  );
}
export default App;
```

```
function Navbar() {
  return <nav style={{ background: "#10ADDE", color: "#fff" }}>Demo App</nav>;
}

function MainPage() {
  return (
    <div>
      <h3>Main Page</h3>
      <Content />
    </div>
  );
}

function Content() {
  return (
    <div>
      <Message />
    </div>
  );
}

function Message() {
// Getting access to the state provided by the context provider wrapper
  const { user } = useContext(UserContext);
  return <p>Welcome {user} :)</p>;
}
```

We start by importing a `createContext` Hook, which is used for creating a context, and a `useContext` Hook, which will extract the state provided b a context provider.

We then call the `createContext` Hook function, which returns a context object with an empty value. That is then stored in a variab called `UserContext`.

Moving forward, we proceed to wrap the `MainPage` component with the `Context.Provider` and pass the `user` object to it, which provides it every component nested within the `MainPage` component.

Lastly, we extract this user in the `Message` component nested within the `MainPage` component, using the `useContext` Hook and a bit destructuring.

We have completely nullified the need to pass down the user prop through the intermediary components. As a result, we've solved the issue of pro drilling.

Our rendered output remains the same, but the code underneath is a bit leaner and cleaner.

The Context API can be really helpful in some use cases, such as authentication when you need to check if the user is authenticated in a few unrelate components.

Optimizing React Apps

1.The useMemo Hook

`useMemo()` is a built-in React hook that accepts 2 arguments — a function `compute` that computes a result and the `depedencies` array:

```
const memoizedResult = useMemo(compute, dependencies);
```

The useMemo hook is used in the functional component of React to return a memoized value.

In computer science, memoization is a concept used in general when we can save re-compilation time by returning the cached result. Now, you must b wondering what memoization is in React-Hooks.

> Memoization refers to the concept of not recompiling a function with the same argument again for the next run because it returns the cached result the next time that it is called.

During initial rendering, `useMemo(compute, dependencies)` invokes `compute`, memoizes the calculation result, and returns it to the component.

If during next renderings the dependencies don't change, then `useMemo()` *doesn't invoke* `compute` but returns the memoized value.

But if dependencies change during re-rendering, then `useMemo()` *invokes* `compute`, memoizes the new value, and returns it.

That's the essence of `useMemo()` hook.

If your computation callback uses props or state values, then be sure to indicate these values as dependencies:

```
const memoizedResult = useMemo(() => {
  return expensiveFunction(propA, propB);
}, [propA, propB]);
```

Now let's see how `useMemo()` works in an example.

A component `<CalculateFactorial />` calculates the factorial of a number introduced into an input field.

Here's a possible implementation of `<CalculateFactorial />` component:

```
import { useState } from 'react';
export function CalculateFactorial() {
  const [number, setNumber] = useState(1);
  const [inc, setInc] = useState(0);
  const factorial = factorialOf(number);
  const onChange = event => {
    setNumber(Number(event.target.value));
  };
  const onClick = () => setInc(i => i + 1);

  return (
    <div>
      Factorial of
      <input type="number" value={number} onChange={onChange} />
      is {factorial}
      <button onClick={onClick}>Re-render</button>
    </div>
  );
}
function factorialOf(n) {
  console.log('factorialOf(n) called!');
  return n <= 0 ? 1 : n * factorialOf(n - 1);
}
```

Every time you change the input value, the factorial is calculated `factorialOf(n)` and `'factorialOf(n) called!'` is logged to console.

On the other side, each time you click *Re-render* button, `inc` state value is updated. Updating `inc` state value triggers `<CalculateFactoria />` re-rendering. But, as a secondary effect, during re-rendering the factorial is recalculated again — `'factorialOf(n) called!'` is logged console.

How can you memoize the factorial calculation when the component re-renders? Welcome `useMemo()` hook!

By using `useMemo(() => factorialOf(number), [number])` instead of simple `factorialOf(number)` , React memoizes the factorial calculatio

Let's improve `<CalculateFactorial />` and memoize the factorial calculation:

```
import { useState, useMemo } from 'react';
export function CalculateFactorial() {
  const [number, setNumber] = useState(1);
  const [inc, setInc] = useState(0);
  const factorial = useMemo(() => factorialOf(number), [number]);
  const onChange = event => {
    setNumber(Number(event.target.value));
  };
  const onClick = () => setInc(i => i + 1);

  return (
    <div>
      Factorial of
      <input type="number" value={number} onChange={onChange} />
      is {factorial}
      <button onClick={onClick}>Re-render</button>
    </div>
  );
}
```

```
function factorialOf(n) {
  console.log('factorialOf(n) called!');
  return n <= 0 ? 1 : n * factorialOf(n - 1);
}
```

Every time you change the value of the number, `'factorialOf(n) called!'` is logged to console. That's expected.

However, if you click *Re-render* button, `'factorialOf(n) called!'` isn't logged to console because `useMemo(() => factorialOf(number), [number])` returns the memoized factorial calculation. Great!

2.The useCallback Hook

Improving performance In React applications includes preventing unnecessary renders and reducing the time a render takes propagate. `useCallback()` can help us prevent some unnecessary renders and therefore gain a performance boost.

1. Functions equality

Before diving into `useCallback()` , let's have a quick refresher about the concept of referential equality and function equality.

In JavaScript, functions can be treated just like any other variable. a function can be passed as an argument to other functions, returned by anoth function, assigned as a value to a variable, compared, and so on. In short, it can do anything that an object can do.

Let's implement a function called *`sumFunctionFactory()`* , which returns another function that sums numbers. Then let's use that function to crea two functions *`function1`* and *`function2`*

```
// factory function
function sumFunctionFactory() {
  return (a, b) => a + b;
}

const function1 = sumFunctionFactory();
const function2 = sumFunctionFactory();

function1(2, 3);
// expected output: 5
function2(2, 3);
// expected output: 5

console.log(function1 === function2);
// expected output: false
```

The functions *`function1`* and *`function2`* share the same code source, but they are distinct separate function objects, meaning they refer different instances, thus Comparing them evaluates to false and that's just how JavaScript works.

2. The useCallback Hook

Going back to React, when a component re-renders, every function inside of the component is recreated and therefore these functions' reference change between renders.

`useCallback(callback, dependencies)` will return a memoized instance of the callback that only changes if one of the dependencies has change This means that instead of recreating the function object on every re-render, we can use the same function object between renders.

```
const memoized = useCallback(() => {
   // the callback function to be memoized
 },
  // dependencies array
[])
```

Let's create an example so we can understand more easily how this hook works. We're going to create a component called `Notes` , which will be o parent component. This component will have a state called `notes` , which will be all our notes, and a function called `addNote` that will add a rando note every time we click a button.

```
const Notes = () => {
 const [notes, setNotes] = useState([]);
 const addNote = () => {
   const newNote = "random";
   setNotes(n => [...n, newNote]);
 };
 return (
```

```
    <div>
    <h1>Button:</h1>
    {notes.map((note, index) => (
      <p key={index}>{note}</p>
    ))}
    </div>
  );
};
```

Now, let's create our `Button` component. We're going to create a simple button and pass a prop called `addNote` that will add a note every time w click it. We put a console.log inside our `Button` component, so every time our component re-renders it'll console it.

```
const Button = ({ addNote }) => {
 console.log("Button re-rendered :( ");
 return (
   <div>
   <button onClick={addNote}>Add</button>
   </div>
 );
};
```

Let's import our `Button` component and pass our `addNote` function as a prop and try to add a note. We can see that we can add a note successfull but also our `Button` component re-renders every time, and it shouldn't. The only thing that's changing in our app is the `notes` state, n the `Button` .

This is a totally unnecessary re-render in our application, and this is what the `useCallback` hook can help us avoid. So, in this case, how we could us the `useCallback` hook to avoid an unnecessary re-render in our component?

We can wrap the `addNote` function with the `useCallback` hook, and pass as a dependency the `setNotes` updater function, because the on thing that's a dependency of our `Button` component is the `setNotes` .

```
const addNote = useCallback(() => {
 const newNote = "random";
 setNotes(n => [...n, newNote]);
}, [setNotes]);
```

But if we look at the console, we can see that the `Button` component is still re-rendering.

We know that React will re-render every component by default unless we use something that can prevent this. In this case, we can us the `React.memo` to prevent re-rendering our `Button` component unless a prop has changed—in our case, the `addNote` prop.

> `React.memo()` is a higher-order component that we can use to wrap components that we do not want to re-render unless props within them change

But, since we're using the `useCallback` hook, it'll never change, so our `Button` component will never be re-rendered. This is how our `Button` w look:

```
const Button = React.memo(({ addNote }) => {
 console.log("Button re-rendered :( ");
return (
   <div>
   <button onClick={addNote}>Add</button>
   </div>
 );
});
```

Now we have a very performative and effective component, avoiding unnecessary re-renders in our components. The `useCallback` hook is pret simple at first, but you must pay attention to where and when to use this hook, otherwise it won't help you at all.

Difference Between useMemo And useCallback

In both useMemo and useCallback, the hook accepts a function and an array of dependencies. The major difference between useCallback ar useMemo is that useCallback will memory the returned value, whereas useMemo will memory the function. Essentially, the only difference between thes hooks is that one caches a value type, and the other caches a function.

Let's take an example; if the computationally expensive code accepts arguments and returns a value, you would need to use useMemo so you cou keep referencing that value between renders without re-running the computationally expensive code.

On the other hand, in order to keep a function instance persistent for multiple renders, useCallback is needed. This is like placing a function outside tl scope of your react component to keep it intact.

3.Throttling and Debouncing Event Action in JavaScript

Debouncing

enforces that a function won't be called again until a certain amount of time has passed without it being called.

Throttling

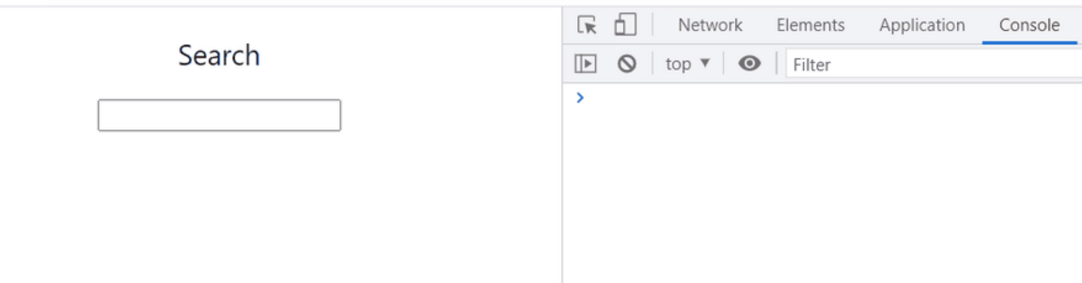
enforces a maximum number of times a function can be called over time.

Debouncing

Before dive deep into debouncing, let's see a simple and normal example that implement a search box that allows users to search something witho clicking any button.

```
function App() {

  const handleChange = e => {
    console.log('api call...')
  }

  return (
    <div className="App">
      <header className="App-header">
        <p> Search  </p>
        <input type='text' onChange={handleChange} />
      </header>
    </div>
  );
}
```

The issue is that `handleChange` is very expensive, and this is bad to the server because it will receive many HTTP requests in the same time.



To solve the problem, we need to use a `debounce function` .

A debounce function is called after a specific amount of time passes since its last call.

```
function debounce(fn, delay) {
    let timer
    return function (...args) {
      clearTimeout(timer)
      timer = setTimeout(()=>fn(...args), delay)
    }
}
```

The idea is to define a high-order function called debounce takes as arguments a callback function and a delay in ms, then returns a new function th sets the timer to execute the callback after the timer done.

The secret here is that every call of the function returned from the debounce function will cancel the previous timer using cleartimeout(timer)

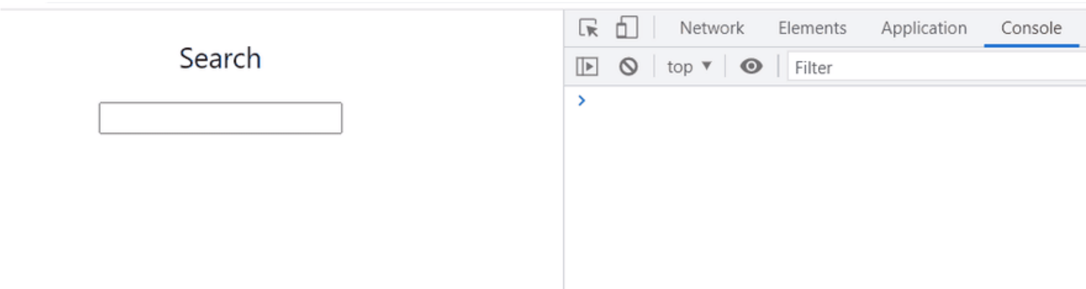
and start a new timer.

With this approach, we can be sure that the callback function will be executed just once after the time that we passed as an argument in the last call.

```
<div className="App">
    <header className="App-header">
```

```
        <p> Search  </p>
        <input type='text' onChange={debounce(handleChange, 500)} />
    </header>
</div>
```

Result:

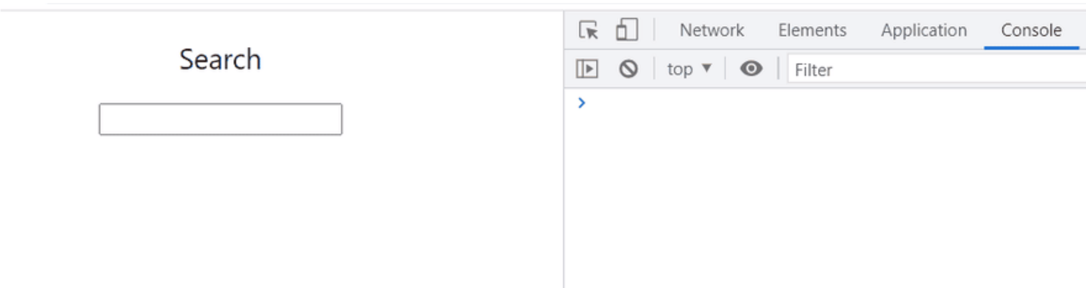


Throttling

Let's assume that we have an event listener in our app to track the movement of the user mouse, then send data to a backend server to do som operations based on the location of the mouse.

```
const handleMouseMove = e => {
      //everytime the mouse moved this function will be invoked
      console.log('api call to do some operations...')
  }
 //event listener to track the movement of the mouse
window.addEventListener('mousemove',handleMouseMove)
```

If we stick with this solution, we will end up with a down backend server because it will receive a hundred of requests in short duration.



1600 API calls in few seconds is very very bad ????????????.

To fix this issue, we need to limit the number of API calls, and this kind of problems can be solved using a throttle function.

A throttle function is a mechanism to limit the number of calls of another function in a specific interval, any additional calls within the specified tim interval will be ignored.

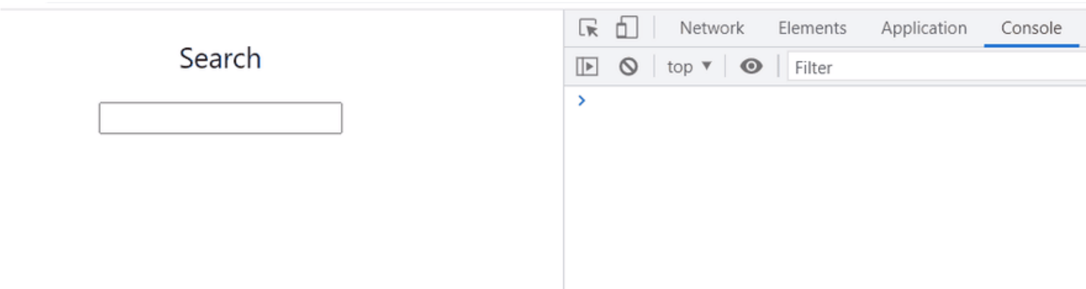
```
function throttle(fn, delay) {
    let run = false
    return function (...args) {
      if (!run) {
        fn(...args)
        run = true
        setTimeout( () => run = false, delay)
      }
    }
  }
```

The throttle function accepts two arguments: fn, which is a function to throttle, and delay in ms of the throttling interval and returns a throttled function.

```
const handleMouseMove = e => {
      //everytime the mouse moved this function will be invoked
      console.log('api call to do some operations...')
  }
```

```
    //event listener to track the movement of the mouse
    window.addEventListener('mousemove', throttle(handleMouseMove, 1000))
  //1000ms => 1 second
```

Result:



4.Avoid using Index as Key for map

You often see indexes being used as a key when rendering a list.

```
{
    comments.map((comment, index) => {
        <Comment
            {..comment}
            key={index} />
    })
}
```

But using the key as the index can show your app incorrect data as it is being used to identify DOM elements. When you push or remove an item fro the list, if the key is the same as before, React assumes that the DOM element represents the same component.

It's always advisable to use a unique property as a key, or if your data doesn't have any unique attributes, then you can think of using the `shorti` `module` which generates a unique key.

```
import shortid from  "shortid";
{
    comments.map((comment, index) => {
        <Comment
            {..comment}
            key={shortid.generate()} />
    })
}
```

However, if the data has a unique property, such as an ID, then it's better to use that property.

```
{
    comments.map((comment, index) => {
        <Comment
            {..comment}
            key={comment.id} />
    })
}
```

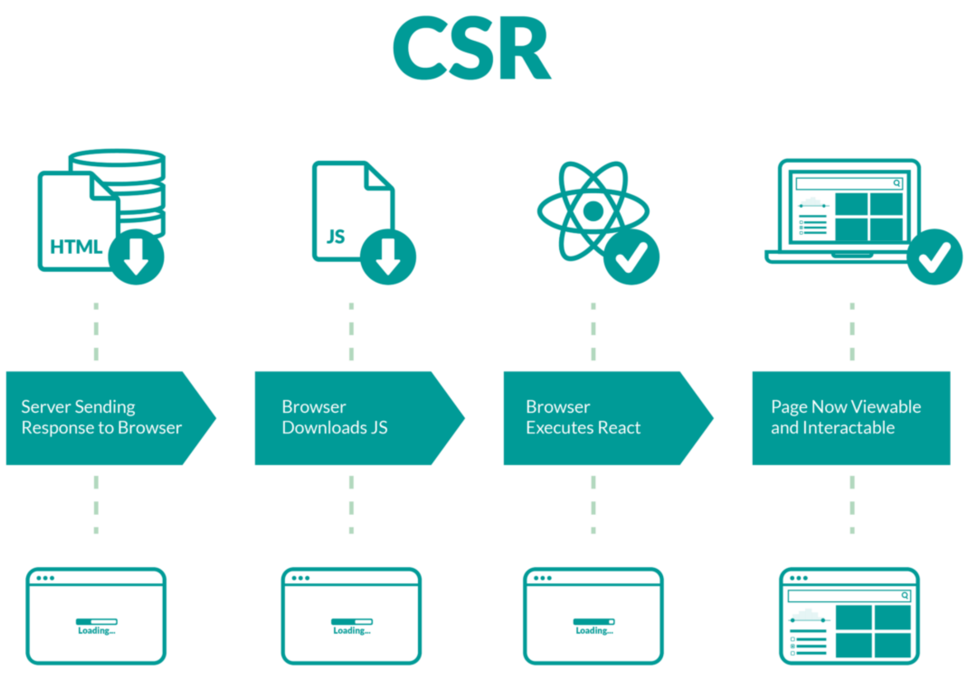
In certain cases, it's completely okay to use the index as the key, but only if below condition holds:

- The list and items are static
- The items in the list don't have IDs and the list is never going to be reordered or filtered
- List is immutable

5.Code Splitting using React.Lazy & Suspense

The more code a project has, the slower the browser will load. Therefore, you often have to balance the size of your dependencies with the performanc you expect out of your JavaScript. Code splitting is a useful way to strike this balance.

What is code splitting?



Many JavaScript frameworks bundle all dependencies into one single large file. This makes it easy to add your JavaScript to an HTML web page. Th bundle requires only one link tag with fewer calls needed to set up the page since all the JavaScript is in one place. In theory, bundling JavaScript in th manner should speed up page loads and lower the amount of traffic that page needs to handle.

At a certain point, however, a bundle grows to a certain size at which the overhead of interpreting and executing the code slows the page load dow instead of speeding it up. This critical point is different for every page, and you should test your pages to figure out where this is. There isn't a gener guideline — it all relies on the dependencies which is being loaded.

The key to code splitting is figuring out which parts of a page need to use different JavaScript dependencies. Code splitting allows you to strategical remove certain dependencies from bundles, then insert them only where they are needed. Instead of sending all the JavaScript that makes up th application as soon as the first page is loaded, splitting the JavaScript into multiple chunks improves page performance by a huge margin.

Code splitting is a common practice in large React applications, and the increase in speed it provides can determine whether a user continues using web application or leaves. Many studies have shown that pages have less than three seconds to make an impression with users, so saving off eve fractions of a second could be significant. Therefore, aiming for three seconds or less of load time is ideal.

How does code splitting work in React?

Different bundlers work in different ways, but React has multiple methods to customize bundling regardless of the bundler used.

Dynamic imports

Perhaps the simplest way to split code in React is with the dynamic "import" syntax. Some bundlers can parse dynamic import statements natively, whi others require some configuration. The dynamic import syntax works for both static site generation and server-side rendering.

Dynamic imports use the `then` function to import only the code that is needed. Any call to the imported code must be inside that function.

```
import("./parseText").then(parseText => {
  console.log(parseText.count("This is a text string", "text"));
});
```

The single bundle used in the application can be split into multiple separate chunks:

- One responsible for the code that makes up our initial route
- Other chunks that contains our unused code

With the use of dynamic imports, a secondary chunk can be *lazy loaded,* or loaded on demand. For eg, the code that makes up the chunk can be loade only when the user presses the button or on execution of certain condition.

Using React.lazy

React.lazy allows for lazy loading of imports in many contexts. The React.lazy function allows you to dynamically import a dependency and render th dependency as a component in a single line of code. The Lazy component should then be rendered inside Suspense Component which helps to refle some fallback content meanwhile the lazy component loads.

```
import React, { Suspense } from 'react';
const LazyComponent = React.lazy(() => import('./LazyComponent'));
function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}
```

The fallback prop can accept any element of React which will be rendered while waiting for the loading of the Component. The Suspense Compone can be placed anywhere above the lazy component. Moreover, multiple lazy components can be wrapped with a single Suspense Component.

```
import React, { Suspense } from 'react';
const ComponentOne = React.lazy(() => import('./ComponentOne'));
const ComponentTwo = React.lazy(() => import('./ComponentTwo'));
function MyComponent() {
   return (
      <div>
        <Suspense fallback={<div>Loading...</div>}>
         <ComponentOne />
         <ComponentTwo />
        <Suspense>
      </div>
   );
}
```

Route based code splitting: It can be difficult to implement code-splitting in code, the bundles can be split evenly, which will improve the experience f the user.

```
import React from 'react';
import Suspense from 'react';
import lazy from 'react';
import {Route, Switch, BrowserRouter } from 'react-router-dom';
const HomeComponent = lazy(() => import('./routes/HomeComponent'));
const BlogComponent = lazy(() => import('./routes/BlogComponent'));
const App = () => (
  <Suspense fallback={<div>Loading...</div>}>
    <BrowserRouter>
      <Switch>
         <Route path={"/home"}>
            <HomeComponent />
         </Route>
         <Route path={"/blog"}>
            <BlogComponent />
         </Route>
         <Route path="/">
            <Redirect to={"/home"} />
         </Route>
      </Switch>
    </BrowserRouter>
  <Suspense/>
```

Conclusion

So in this module we understood:

- What is local state and props drilling
- Managing Application state using Context API
- Different Optimization techniques for React

In the next module we will learn how to manage Application State/Global State using Redux Library

---

Thank You !