Performance Optimization

# Performance Optimization Tools

## Prevent Wasted Renders

* We can **memoize** components with **memo**
* We can use **useMemo** and **useCallback**
* Pass elements as children or regular prop

## Improve App Speed/ Responsiveness

* **useMemo**
* **useCallback**
* **useTransition**

## Reduce bundle Size

* Use fewer 3rd-party packages
* Code Splitting and Lazy Loading

# Prevent Wasted Renders

## When does a component instance re-renders?

* When state changes
* The Context changes
* When parent component is re-rendered

Re-rendering does **not** mean that the DOM is updatded

Re-rendering means that the component function get’s called

* **It can be an EXPENSIVE operation**
* Diffing / Reconciliation

**All these re-renders are called WASTED RENDERS**

* They do not produce any change in the DOM
* Only a problem when it happens **to frequently** or **component is very slow**

# The Profiler Tool

* We can analyze renders, re-renders and how much the re-render took
* On the settings of the profiler, Check – Record why each component….
  + This will provide one of the 3 reasons we mentioned ABOVE

A screenshot of a computer

Description automatically generated

### The Flamegraph

* Another way to present the COMPONENT TREE
* Colors
  + Grey
    - Component is not rerenderd
  + The Brighter the color
    - The more it took to rerender

A screenshot of a computer

Description automatically generated

* Hovering over a component, we can see the **EXACT REASON** why it rerendered

### The Ranked Tab

* We can see a rank of how fast and slow the component rendered

# Trick 1. Pass component as children

* If we pass a component as Children, React will not re-render that component even if the Parent Component is Re-rendered
  + This happens if the children component if not dependent on the State

A screenshot of a computer

Description automatically generated- we can see here that the PostProvider has rerendered, so basically all other Components inside should have rendered

- But, since the components are passed as **CHILDRED** into the PostProvider, will rerender only if they are actually using **(consume)** the Context and are affected by the state change

- We can see that **Main and Footer** did **NOT** Rerender.

# Memoization

* The operation that executes a function and stores the result in **cache**
* If we call the function again with **same arguments**, the result will be took from **cache** instead of running the function again

A diagram of a store result

Description automatically generated

* In **React**
  + Memoize **components** with **memo**
  + Memoize **objects** with **useMemo**
  + Memoize  **functions** with **useCallback**
* Memozation in React
  + Prevent Wasted Renders
  + Improve App speed/ responsiveness

## The Memo Function

* Used to create a component that will **NOT** re-render when the parent re-renders as long as the **props stay the same between renders**
* Only affects **PROPS**
  + Will still re-render when it’s **OWN state** changes
  + Will still re-render when the **Context** the is **consumed changes**
* Should we memo all our components???? 🡺 **NO**
  + Used when component is
    - Heavy
    - Re-renders Often with same props

### Memo practice

A screenshot of a computer

Description automatically generated

* We can see here **HOW LONG** the  **Archive** component is taking to re-render
* All other components are not even on the graph
* Based on the 3 steps, we can see the HUGE difference between when the Archive was open and when it was not open

### To use memo

* we just WRAP the whole function into the memo() function, which is imported from react
* we store the result into a variable with the same nameA screen shot of a computer program

  Description automatically generated

### Memo Results

A screenshot of a computer

Description automatically generated

* as we can see, there was an increase in Time when the Archive 1st opened
* After that, all other operations were **FAST** because archive was not **re-rendered**

A screenshot of a computer

Description automatically generated

# useMemo

* An object or a function will always be different, even if they have the same structure.
* This means that if Objects or Functions are passed as props
  + Will always bee seen as **NEW PROPS ON EACH RE-RENDER**
* This means that PROPS will be different in each Re-render, so **MEMO will not work**

**useMemo to the rescue**

* We use **useMemo 🡺** the momoize values
* We use **useCallback 🡺** to memoize functions
* Values passed into useMemo and useCallback will be stored in memory (‘cached’)
  + As long as the dependencies (‘inputs’) are the same

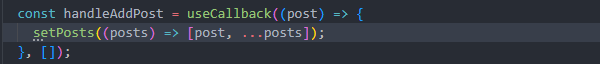
## useMemo structure

* useMemo(()=>{ callback function },[ dependency array ])
  + what should be calculated the 1st time this runs and stored into the memo
  + dependency array 🡺 determines when the callback function needs to be recalled

A computer screen shot of a computer screen

Description automatically generated

# useCallback

* memoize functions
* 

# Context Re-Render Optimization

## 3 things to check before Context Optimization

* Constant change of state in context
* Context has many consumers
* App is slow and laggy

### Only if all of THESE ARE TRUE, we optimize the Context

### Optimization

* Pass Consumers as **children**  to the Context
* A screen shot of a computer screen

  Description automatically generated
* Memoize the value of the context Provider by using **useMemo inside the Context**

A screen shot of a computer program

Description automatically generated

# WorldWise App Optimization

* We walk through the ENTIRE App to find **PERFORMANCE BOTTLENECKS**
  + Using the **Profiler tool**
* To **Identify** really bad performing components, we use the **Ranked tab**

A screenshot of a computer

Description automatically generated

* We can see that the Component that took the longest time is the **Routes**
* We cycle through all Renders to find slow components
* 

### Using the Profiler, we can see that there are NONE slow components

# Optimizing Bundle Size

## What is a Bundle?

* When the use navigates to our app, they visit a website that is hosted on a server
* The server sends a HUGE JS file that contains the entire code
  + This is the **BUNDLE**
* **Bundle Size**
  + The amount of the code and the size of the file that the user needs to download to **start**  the application

## Code Splitting 🡺 Lazy Loading

* The most used technique
* To split the code at the **Routes** level (pages)

### Steps

1. Identify the pages
   1. Product, pricing, Login, AppLayout, Page not Found
2. Use React **lazy functionality**
   1. Const Homepage = lazy(()=>import(‘./pages/homepage’))
   2. A screen shot of a computer code

      Description automatically generated
3. We use **Suspense API**
   1. Is a React functionality that allows certain components to ‘suspense’, to wait for something to happen
   2. These lazy components will wait until they are loaded
   3. We are building the ‘**Suspense’ fallback** above our Routes tree

A computer screen shot of a computer code

Description automatically generated

* 1. With this **Suspense** we will display the **Spinner fallback** option until a PAGE IS LOADED

# Do’s and Don’t’s for Optimization

## Do

* Find actual performance bottlenecks using the Profiler
* Find performance issues by eye ( laggy UI)
* Fix those real performance Issues
* Memoize expensive Re-renders
* Optimize Context if has lots’ of consumers
* Memoize Context Value + child components
* Divide Context into multiple contexts
* Implement code splitting + lazy loading in SPA routes

## Don’t

* Don’t Optimize Prematurely
* Don’t optimize if there is nothing to optimize
* Don’t wrap all components in memo().
* Don’t wrap all values into useMemo().
* Don’t wrap all function into useCallback().
* Don’t optimize context if it’s not slow and doesn’t have many cosumers.