Performance in React(Advanced)

Once a React application grows, maintenance becomes a priority. To prepare for this eventuality, we'll cover performance optimization, type safety, testing, and project structure. Each can strengthen your app to take on more functionality without losing quality.

Performance optimization prevents applications from slowing down by assuring efficient use of available resource. Typed programming languages like TypeScript detect bugs earlier in the feedback loop. Testing gives us more explicit feedback than typed programming, and provides a way to understand which actions can break the application. Lastly, project structure supports the organized management of assets into folders and files, which is especially useful in scenarios where team members work in different domains.

Performance in React

This section is just here for the sake of learning about performance improvements in React. We wouldn't need optimizations in most React applications, as React is fast out of the box. While more sophisticated tools exist for performance measurements in JavaScript and React, we will stick to a simple `console.log()` and our browser's developer tools for the logging output.

Don't run on first render

Previously we covered React's useEffect Hook, which is used for side-effects. It runs the first time a component renders (mounting), and then every re-render (updating). By passing an empty dependency array to it as a second argument, we can tell the hook to run on the first render only. Out of the box, there is no way to tell the hook to run only on every re-render (update) and not on the first render (mount). For instance, examine this custom hook for state management with React's `useStat`e Hook and its semi persistent state with local storage using React's `useEffect` Hook:

local storage using React's `useEffect` Hook:

```
const useSemiPersistentState = (key, initialState) => {
  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {
    console.log('A');

    localStorage.setItem(key, value);
  }, [value, key]);

  return [value, setValue];
};
```

src/App.js

With a closer look at the developer's tools, we can see the log for a first time render of the component using this custom hook. It doesn't make sense to run the side-effect for the initial rendering of the component, because there is nothing to store in the local storage except the initial value. It's a redundant function invocation, and should only run for every update (re-rendering) of the component.

As mentioned, there is no React Hook that runs on every re-render, and there is no way to tell the `useEffect` hook in a React idiomatic way to call its function only on every re-render. However, by using React's useRef Hook which keeps its `ref.current` property intact over re-renders, we can keep a *made up state* (without re-rendering the component on state updates) of our component's lifecycle:

```
const useSemiPersistentState = (key, initialState) => {

  const isMounted = React.useRef(false);


  const [value, setValue] = React.useState(
    localStorage.getItem(key) || initialState
  );

  React.useEffect(() => {

    if (!isMounted.current) {
      isMounted.current = true;
    } else {

      console.log('A');
      localStorage.setItem(key, value);
    }

  }, [value, key]);

  return [value, setValue];
};
```

src/App.js

We are exploiting the `ref` and its mutable `current` property for imperative state management that doesn't trigger a re-render. Once the hook is called from its component for the first time (component render), the ref's `current` is initialized with a `false` boolean called `isMounted`. As a result, the side-effect function in `useEffect` isn't called; only the boolean flag for `isMounted` is toggled to `true` in the side-effect. Whenever the hook runs again (component re-render), the boolean flag is evaluated in the side-effect. Since it's `true`, the side-effect function runs. Over the lifetime of the component, the `isMounted` boolean will remain `true`. It was there to avoid calling the side-effect function for the first time render that uses our custom hook.

The above was only about preventing the invocation of one simple function for a component rendering for the first time. But imagine you have an expensive computation in your side-effect, or the custom hook is used frequently in the application. It's more practical to deploy this technique to avoid unnecessary function invocations.

> *Note: This technique isn't only used for performance optimizations, but for the sake of having a side-effect run only when a component re-renders. I used it several times, and I suspect you'll stumble on one or the other use case for it eventually.*

Don't re-render if not needed

Earlier, we explored React's re-rendering mechanism. We'll repeat this exercise for the App and List components. For both components, add a logging statement.

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

  console.log('B:App');
};

const List = ({ list, onRemoveItem }) =>

  console.log('B:List') ||

  list.map(item => (
    <Item
      key={item.objectID}
      item={item}
      onRemoveItem={onRemoveItem}
    />
  ));
```

src/App.js

Because the List component has no function body, and developers are lazy folks who don't want to refactor the component for a simple logging statement, the List component uses the `||` operator instead. This is a neat trick for adding a logging statement to a function component without a function body. Since the `console.log()` on the left hand side of the operator always evaluates to false, the right hand side of the operator gets always executed.

```
function getTheTruth() {
  if (console.log('B:List')) {
    return true;
  } else {
    return false;
  }
}

console.log(getTheTruth());
// B:List
// false
```

src/App.js

Let's focus on the actual logging in the browser's developer tools. You should see a similar output. First the App component renders, followed by its child components (e.g. List component).

```
B:App
B:List
B:App
B:App
B:List
```

Since a side-effect triggers data fetching after the first render, only the App component renders, because the List component is replaced by a loading indicator in a conditional rendering. Once the data arrives, both components render again.

```
// initial render
B:App
B:List

// data fetching with loading
B:App

// re-rendering with data
B:App
B:List
```

So far, this behavior is acceptable, since everything renders on time. Now we'll take this experiment a step further, by typing into the SearchForm component's input field. You should see the changes with every character entered into the element:

```
B:App
B:List
```

But the List component shouldn't re-render. The search feature isn't executed via its button, so the `list` passed to the List component should remain the same. This is React's default behavior, which surprises many people.

If a parent component re-renders, its child components re-render as well. React does this by default, because preventing a re-render of child components could lead to bugs, and the re-rendering mechanism of React is still fast.

Sometimes we want to prevent re-rendering, however. For instance, huge data sets displayed in a table shouldn't re-render if they are not affected by an update. It's more efficient to perform an equality check if something changed for the component. Therefore, we can use React's memo API to make this equality check for the props:

```
const List = React.memo(
  ({ list, onRemoveItem }) =>
    console.log('B:List') ||
    list.map(item => (
      <Item
        key={item.objectID}
        item={item}
        onRemoveItem={onRemoveItem}
      />
    ))

);
```

src/App.js

However, the output stays the same when typing into the SearchForm's input field:

```
B:App
B:List
```

The `list` passed to the List component is the same, but the `onRemoveItem` callback handler isn't. If the App component re-renders, it always creates a new version of this callback handler. Earlier, we used React's useCallacbk Hook to prevent this behavior, by creating a function only on a re-render (if one of its dependencies has changed).

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


  const handleRemoveStory = React.useCallback(item => {

    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });

  }, []);


  ...

  console.log('B:App');

  return (... );
};
```

src/App.js

Since the callback handler gets the `item` passed as an argument in its function signature, it doesn't have any dependencies and is declared only once when the App component initially renders. None of the props passed to the List component should change now. Try it with the combination of `memo` and `useCallback`, to search via the SearchForm's input field. The "B:List" output disappears, and only the App component re-renders with its "B:App" output.

While all props passed to a component stay the same, the component renders again if its parent component is forced to re-render. That's React's default behavior, which works most of the time because the re-rendering mechanism is fast enough. However, if re-rendering decreases the performance of a React application, `memo` helps prevent re-rendering.

Sometimes `memo` alone doesn't help, though. Callback handlers are re-defined each time in the parent component and passed as *changed* props to the component, which causes another re-render. In that case, `useCallback` is used for making the callback handler only change when its dependencies change.

Don't rerun expensive computations

Sometimes we'll have performance-intensive computations in our React components – between a component's function signature and return block – which run on every render. For this scenario, we must create a use case in our current application first.

```
const getSumComments = stories => {
  console.log('C');

  return stories.data.reduce(
    (result, value) => result + value.num_comments,
    0
  );
};


const App = () => {
  ...

  const sumComments = getSumComments(stories);

  return (
    <div>

      <h1>My Hacker Stories with {sumComments} comments.</h1>
      ...
    </div>
  );
};
```

src/App.js

If all arguments are passed to a function, it's acceptable to have it outside the component. It prevents creating the function on every render, so the `useCallback` hook becomes unnecessary. The function still computes the value of summed comments on every render, which becomes a problem for more expensive computations.

Each time text is typed in the input field of the SearchForm component, this computation runs again with an output of "C". This may be fine for a non-heavy computation like this one, but imagine this computation would take more than 500ms. It would give the re-rendering a delay, because everything in the component has to wait for this computation. We can tell React to only run a function if one of its dependencies has changed. If no dependency changed, the result of the function stays the same. React's useMemo Hook helps us here:

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

  const sumComments = React.useMemo(() => getSumComments(stories), [
    stories,
  ]);


  return ( ... );
};
```

src/App.js

For every time someone types in the SearchForm, the computation shouldn't run again. It only runs if the dependency array, here `stories`, has changed. After all, this should only be used for cost expensive computations which could lead to a delay of a (re-)rendering of a component.

Now, after we went through these scenarios for `useMemo`, `useCallback`, and `memo`, remember that these shouldn't necessarily be used by default. Apply these performance optimization only if you run into a performance bottlenecks. Most of the time this shouldn't happen, because React's rendering

mechanism is pretty efficient by default. Sometimes the check for utilities like `memo` can be more expensive than the re-rendering itself.

```
  □ □ □□  □   ã□  F   □□  □   □  )□     □   9□  5□  @@  □   °□  n□  □PNG
□
   IHDR   □   □□□   (-□S   äPLTE""""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU
□
   IHDR   □   □□□   ×©ÍÊ  □ePLTE""""""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□iqÎ□+1°□}Â□5
□
   IHDR       □□   D¤□Æ  □APLTE   """""""""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0
□
   IHDR   @   @□□   □·□ì  □:PLTE   """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?i÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□
©¬¢Z\TsOR*□(□  ¯©□J□□□□u□X/□4J□9□¡5·DEµ4kÇ4□&i¥V4Ú□¡®Ð□□¯□vsf:àg,□¢èBC»î$¶□ºÍùî□□á□@□
-ê>Û□º«¢XÕ¢î}ß¨ëÛÑ;□ÃöN´□ØvÅý□Î¸ÿ1 □ë×ÄO@&v/Äþ_□ö\ô□Ç\í.□□½+0□□;□□□!□fÊ□¦´Ó%Â JY·O□Â□'
```

Exercises:

- Confirm the changes from the last section.
- Read more about React's memo API.
- Read more about React's useCallback Hook.
- Download *React Developer Tools* as an extension for your browser. Open it for your application in the browser via the browser's developer tools and try its various features. For instance, you can use it to visualize React's component tree and its updating components.
- Does the SearchForm re-render when removing an item from the List with the "Dismiss" button? If it's the case, apply performance optimization techniques to prevent re-rendering.
- Does each Item re-render when removing an item from the List with the "Dismiss" button? If it's the case, apply performance optimization techniques to prevent re-rendering.
- Remove all performance optimizations to keep the application simple. Our current application doesn't suffer from any performance bottlenecks. Try to avoid premature optimizations. Use this section as reference, in case you run into performance problems.