The ultimate guide to optimize React Flow project performance

synergy codes

Table of contents

Why is React Flow prone to performance issues?

React Flow is a library vulnerable to a diagram's performance pitfalls. Especially when a developer isn't cautious while coding.

Even one non-optimized line of code can cause unnecessary re-rendering of the diagram's elements on every state change (mainly when the positions of nodes are updated). As a result, an application works slowly.

To understand how easily performance issues can arise, let's analyze the following example. When you drag a node on the diagram, the application acts as follows:

1. Every node's position change causes node to re-render.
2. A node's state change causes a refresh of ReactFlow's internal state.
3. ReactFlow's internal state refreshing leads to the `<ReactFlow>` component refreshing, which relies, among other things, on the nodes array. Consequently, any changes to a node or other diagram element cause the re-rendering of this main component.

Re-rendering of a single component usually doesn't impact an application's performance.

However, if we unintentionally make the states of nodes dependent on each other or place heavy components as children within the main ReactFlow component, the application may lose its smoothness and fail to meet performance requirements.

In addition, there is a general rule in the React ecosystem not to optimize code too early unless there are performance issues.

As for React Flow, if such issues occur later in a development process, they can be difficult to overcome without significant changes in code logic.

So, the sooner you work on software performance optimization, the better.

Now, I will share six hints on avoiding performance pitfalls in React Flow applications.

The data I present in this article comes from a project consisting of 100 nodes. Every node had two Handles and rendered one of two things:

- In default mode, one text input from MaterialUI.
- In "heavy" mode, one DataGrid from MaterialUI that had nine rows and five columns.

The base FPS (frames per second) with optimal performance in the project on my computer is 60 FPS.

#1.<ReactFlow> component optimization

The basic usage of the `<ReactFlow>` component is as follows:

```
<ReactFlow
  nodes={nodes}
  edges={edges}
  nodeTypes={nodeTypes}
  onNodesChange={onNodesChange}
  onEdgesChange={onEdgesChange}
  onConnect={onConnect}
/>
```

The React Flow library's documentation strongly recommends passing memoized references to this component, whether the props are functions or objects. To ensure optimal performance, follow these two rules:

- **Objects memoization:** Objects passed to the `<ReactFlow>` component should be memoized using `useMemo` or defined outside of the component.
- **Functions memoization:** All functions passed as props should be memoized using `useCallback`.

Additionally, memoized objects and functions must have stable dependencies to ensure consistent behavior.

If you include elements with frequently changing references (e.g., functions not wrapped in `useCallback`) in the `dependencyArray` of `useMemo` or `useCallback`, the memoization will not yield the expected results.

Benchmark

Let’s consider the following code modification that illustrates this issue:

```
<ReactFlow
  nodes={nodes}
  edges={edges}
  nodeTypes={nodeTypes}
  onNodesChange={onNodesChange}
  onEdgesChange={onEdgesChange}
  onConnect={onConnect}
  onNodeClick={() => {}} //added anonymous function
/>
```

Introducing an anonymous function to `onNodeClick` prop forces React to assign a new reference in every render.

The results (number of FPS) for dragging operation:

100 default nodes	Decrease to 10 FPS	100 "heavy" nodes	Decrease to 2 FPS

The cause

This change caused the re-render of all the diagram's nodes whenever a node's state was updated. It means that with every dragging operation, not only the main `<ReactFlow>` component and dragged node are being re-rendered but also 99 nodes remaining.

The conclusion

You must remember to properly memoize props, i.e., using `useCallback` and `useMemo`, when working with `<ReactFlow>` component.

#2. Dependencies on Node and Edge Arrays

Uncontrolled dependencies of components and hooks on node and edge arrays are among the main threats to the performance of applications using ReactFlow.

The state of nodes and edges can change even with minor updates to individual properties of any diagram element. This often results in unnecessary re-renders of components dependent on this state.

The example

Let's assume that you want to display selected nodes' IDs. A quick but not optimal way is:

```tsx
export const Node: React.FC<NodeProps<BaseNode>> = () => {
  const selectedNodes = useStore(
    (state) => state.nodes.filter((node) => node.selected)
  );

  return (
    <div className="react-flow__node-default">
      <Handle type="target" position={Position.Top} />
      <div className={styles.nodeContainer}>
        <DefaultContent />
        {selectedNodes.map((node) => node.id)}
      </div>
      <Handle type="source" position={Position.Bottom} id="a" />
    </div>
  );
};
```

We fetch the complete node array from the store, filter the selected objects, and display their IDs inside the nodes.

The results (number of FPS) for dragging operation:

100 default nodes	Decrease to 12 FPS	100 "heavy" nodes	Decrease to 2 FPS

The cause

The main issue arises from the behavior of the `useStore` hook: the `selectedNodes` reference changes with every update of `state.nodes` (e.g., during every tick of a dragging operation). As a result, all nodes on the diagram defined by this component will re-render with every state update, regardless of whether they are being dragged or not.

The solution

To avoid unnecessary rendering, you can define a field in store where you can keep selected objects. Thanks to that a Node component won't be directly dependent on nodes array.

```
//Store

const useStore = ....
....
selectedNodes: [],
setSelectedNodes: (selectedNodes: Node[]) => {
    set({ selectedNodes });
  },
...
}));
//

// Diagram
const [
    nodes,
    edges,
    onNodesChange,
    onEdgesChange,
    onConnect,
    setSelectedNodes,
  ] = useStore(
    useShallow((state) => [
      state.nodes,
      state.edges,
      state.onNodesChange,
      state.onEdgesChange,
      state.onConnect,
      state.setSelectedNodes,
    ])
  );

  const onSelectionChange = useCallback(
    (event: OnSelectionChangeParams) => {
      setSelectedNodes(event.nodes);
    },
    [setSelectedNodes]
  );
```

```
  return (
    <div className={styles['diagram']}>
      <ReactFlow
        nodes={nodes}
        edges={edges}
        nodeTypes={nodeTypes}
        onNodesChange={onNodesChange}
        onEdgesChange={onEdgesChange}
        onConnect={onConnect}
        onSelectionChange={onSelectionChange}
      />
    </div>
  );
//

// Node component
export const Node: React.FC<NodeProps<BaseNode>> = () => {
  const selectedNodes = useStore((state) => state.selectedNodes);

  return (
    <div className="react-flow__node-default"
>
      <Handle type="target" position={Position.Top} />
      <div className={styles.nodeContainer}>
        <DefaultContent />
        {selectedNodes.map((node) => node.id)}
      </div>
      <Handle type="source" position={Position.Bottom} id="a" />
    </div>
  );
};
```

In this approach, the `selectedNodes` variable is refreshed if and only if the selection actually changes, avoiding unnecessary re-renders of the components that depend on it.

An alternative approach

When we want to extract an array of primitive types from a collection of nodes or edges, as shown in the example above, there is also a straightforward solution that uses `shallow` comparison provided by Zustand.

Zustand offers methods for memoizing selectors either by using the `useShallow` hook or by creating a store with `createWithEqualityFn` and passing the `shallow` parameter. In this example, we use a solution based on store configuration with createWithEqualityFn (see more in Zustand chapter).

```
const selectedNodes = useStore(
  (state) => state.nodes.filter((node) => node.selected).map((node) => node.id)
);
```

Thanks to this approach, the array of primitive types is memoized. Even if its reference changes, the selector will still return the previous reference as long as none of the elements in the array have changed.

The conclusions

Using state from the ReactFlow store in components should be carefully considered, especially when there are dependencies on dynamically changing arrays of nodes or edges.

It's easy to notice issues with suboptimal state usage in a custom node component. However, similar issues can also arise in other scenarios:

- **"Heavy" UI component's dependency on nodes and edges array:** Let's take an example. Suppose you have a sidebar with MaterialUI forms that depend on the nodes array. In that case, this sidebar will render every time any diagram's object changes, decreasing performance.

- **Hook dependency on nodes and edges array:** For instance, a hook that depends on the nodes array and whose returned value is included in the `dependencyArray` of a function passed to `<ReactFlow>`. In this scenario, simply using `useCallback` won't solve the issue because the hook still changes its reference based on dynamically changing data. As a result, the returned values will be updated just as frequently.

#3. Custom nodes and edges templates

One of the most effective ways for keeping a ReactFlow app's performance is wrapping custom nodes and edges in `React.memo`.

Why use `React.memo`?

Thanks to wrapping nodes and edges in a `React.memo` component, even if listeners on the main ReactFlow component are not used optimally, smaller diagrams are unlikely to experience significant performance issues. This is because the contents of nodes and edges won't re-render during i.e. dragging operation.

Benchmark

Let's test how wrapping in `memo` impacts performance. To do that, let's restore an anonymous function in the main `<ReactFlow>` component:

```
<ReactFlow
  nodes={nodes}
  edges={edges}
  nodeTypes={nodeTypes}
  onNodesChange={onNodesChange}
  onEdgesChange={onEdgesChange}
  onConnect={onConnect}
  onSelectionChange={onSelectionChange}
  onNodeClick={() => {}}
/>
```

And let’s wrap Node component in `React.memo`:

```tsx
export const Node: React.FC<NodeProps<BaseNode>> = memo(() => {
  const selectedNodes = useStore((state) => state.selectedNodes);

  return (
    <div className="react-flow__node-default">
      <Handle type="target" position={Position.Top} />
      <div className={styles.nodeContainer}>
        <DefaultContent />
        {selectedNodes.map((node) => node.id)}
      </div>
      <Handle type="source" position={Position.Bottom} id="a" />
    </div>
  );
});
```

The results (number of FPS) for dragging operation:

100 default nodes

In the first second of operation, the number decrease to **50 FPS**

then becomes stable at **60 FPS**

100 "heavy" nodes

Decrease to **30 FPS**

When compared to the first chapter's test's result without using `memo` (decrease to 10 FPS for default nodes and to 2 FPS for "heavy" nodes), it is clearly visible that wrapping components in `memo` significantly improves performance.

It is worth noting that I conducted these tests with developer tools turned off in my browser. These tools can slow down an application and negatively impact the FPS count.

The conclusions

Every custom template should be wrapped in a `React.memo`, which significantly decreases the burden during operations such as dragging.

“Heavy” nodes

Tests prove that “heavy” nodes (those with more complex components, such as DataGrid from MaterialUI) have a bigger impact on a diagram’s performance.

Having implemented all above-mentioned optimization,
the results (number of FPS) for dragging operation:

100
"heavy" nodes

Decrease to
35-40 FPS

The solution

To minimize the impact that Node’s contents have on performance, you can wrap the inside elements in `React.memo` component. Thanks to that, their rendering will be limited only to these cases, when their props will actually change.

```tsx
export const Node: React.FC<NodeProps<BaseNode>> = memo(() => {
  const selectedNodes = useStore((state) => state.selectedNodes);

  return (
    <div className="react-flow__node-default">
      <Handle type="target" position={Position.Top} />
      <div className={styles.nodeContainer}>
        <HeavyContent />
        {selectedNodes.map((node) => node.id)}
      </div>
      <Handle type="source" position={Position.Bottom} id="a" />
    </div>
  );
});

// using React.memo for heavy content
const HeavyContent = memo(() => (
  <DataGrid
    rows={rows}
    columns={columns}
    initialState={{
      pagination: {
        paginationModel: {
          pageSize: 5,
        },
      },
    }}
    pageSizeOptions={[5]}
    checkboxSelection
    disableRowSelectionOnClick
  />
));
```

The results (number of FPS) for dragging operation:

100
default nodes

In the first second of operation, the number decreases to

35-40 FPS

then becomes stable at

60 FPS

The conclusions

If a node has heavy content, such as DataGrid from MaterialUI, the content should be wrapped in `React.memo`. This will avoid unnecessary re-rendering of its contents during state's updates.

#4. Access to Zustand store

Zustand is a small, fast and flexible state management library for React applications, used internally by ReactFlow.

While getting data from the Zustand store, a natural approach would be writing such a code:

```
const [selectedNodes, someOtherProperty] = useStore((state) => [
  state.selectedNodes,
  state.someOtherProperty,
]);
```

Although this code seems intuitive, without a proper store configuration, it can lead to performance issues and, in some cases, errors such as `Maximum update depth exceeded`.

This happens because the array returned by `useStore` is recreated from scratch every time the state changes. Since the resulting array has a different reference on each update, it causes component to re-render even if individual values within the array remain unchanged. A similar situation occurs if an object is returned instead of an array—every state change generates a new reference to the object.

Memoization with `useShallow`

Zustand provides the `useShallow` hook, which memoizes the returned reference if the contents of an array or object have not changed, helping reduce unnecessary re-renders.

```
const [selectedNodes, someOtherProperty] = useStore(useShallow((state) => [
  state.selectedNodes,
  state.someOtherProperty,
]));
```

The drawback of this method is that useShallow requires remembering to use it every time you get more than one field from the store.

Function createWithEqualityFn

The alternative method is to create a store using the function createWithEquityFn (from “Zustand/traditional” package) with a `shallow` parameter. As a result, every selector uses memoization with `shallow` comparison by default.

```
const useStore = createWithEqualityFn<AppState>(
  (set, get) => ({
    ...
  }),
  shallow
);
```

Thanks to that, you can use the original approach to `useStore` hooks.

```
const [selectedNodes, someOtherProperty] = useStore((state) => [
  state.selectedNodes,
  state.someOtherProperty,
]);
```

#5. UI components

Regular React components, apart from diagram objects, can also negatively impact performance if they are not properly optimized.

In such cases, you should follow the hint I explained in the previous chapter and a chapter dedicated to dependencies on node and edge arrays.

Zustand store dependency

Let's look at a sidebar with a form that displays IDs of selected nodes.

```
export const Sidebar = memo(() => {
  const selectedNodes = useStore((state) =>
    state.nodes.filter((node) => node.selected)
  );

  return (
    <div className={styles.container}>
      <div>Selected object key: {selectedNodes.map((node) => node.id)}</div>
      <TextField label="Attribute 1" defaultValue="Value 1" />
      <TextField label="Attribute 2" defaultValue={'Value 2'} />
  ...
```

In this example, a Sidebar will re-render every time there's a change in the nodes array. For instance, at every dragging operation, because `selectedNodes` reference changes at every `state.nodes` update.

You can optimize this component using methods from previous chapters of this article. Assuming that the store was created with `createWithEqualityFn` with `shallow` parameter, you can memoize selectors' results.

```jsx
export const Sidebar = memo(() => {
  // The selector extracts an array of IDs of selected nodes, which is memoized by Zustand
  // thanks to the use of createWithEqualityFn when creating the store
  const selectedNodeIds = useStore((state) =>
    state.nodes.filter((node) => node.selected).map((node) => node.id)
  );

  return (
    <div className={styles.container}>
      <div>Selected object keys: {selectedNodeIds}</div>
      <TextField label="Attribute 1" defaultValue="Value 1" />
      <TextField label="Attribute 2" defaultValue={'Value 2'} />
  ...
```

Zustand memoizes the selector's result, and `selectedNodeIds` will change its reference only if the selection actually changes, which Zustand's `shallow` comparison will detect.

Previous selector result wasn't memoized because it extracted nodes' references from the state, which changes much more often than selected IDs.

`<ReactFlow>` child components

Nested components placed as children of the main `<ReactFlow>` component will be automatically re-rendered with every state change in the diagram because `<ReactFlow>` itself re-renders that often. To prevent this, they should be wrapped in `React.memo`.

```
...
return (
    <div className={styles['diagram']}>
      <ReactFlow
        nodes={nodes}
        edges={edges}
        nodeTypes={nodeTypes}
        onNodesChange={onNodesChange}
        onEdgesChange={onEdgesChange}
        onConnect={onConnect}
        onSelectionChange={onSelectionChange}
      >
        <ChildComponent />
      </ReactFlow>
    </div>
  );
}

// child component wrapped in React.memo
const ChildComponent = memo(() => {
   ...
});
```

#6. Application's architecture

Let's focus on what you should consider for your application's architecture to avoid some of the performance issues while working with React Flow and to make it easier to identify potential performance bottlenecks.

Presentational and Container Components pattern

This pattern helps to isolate performance issues and makes them easier to identify.

Presentational components:

- They are responsible only for displaying data passed by props.
- If they have a state, it is only local and independent from the global application's state.
- They are light and easy to unit test.

Container components:

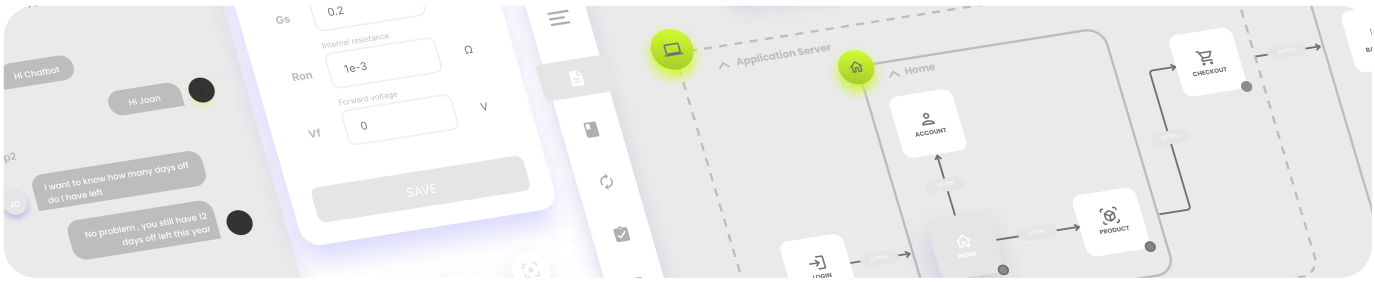
- Have access to an app's state (e.g., to Zustand store).
- Define business logic and interactions (e.g., data processing, functions calls, communication with backend).
- Use hooks.
- Pass data and functions as props to presentational components.

By the above definition, performance issues are most often found in Container Components, which means you usually don't need to review Presentational Components when looking for the source of the problem.

Application's state

In ReactFlow applications, due to the high volatility of nodes' and edges' states, I recommend avoiding storing the diagram state in `useState`, `useReducer`, or the Context API. Instead, it's best to move the state to a Zustand store.

Zustand allows precise subscription to selected parts of the state and performance optimization through selector memoization.

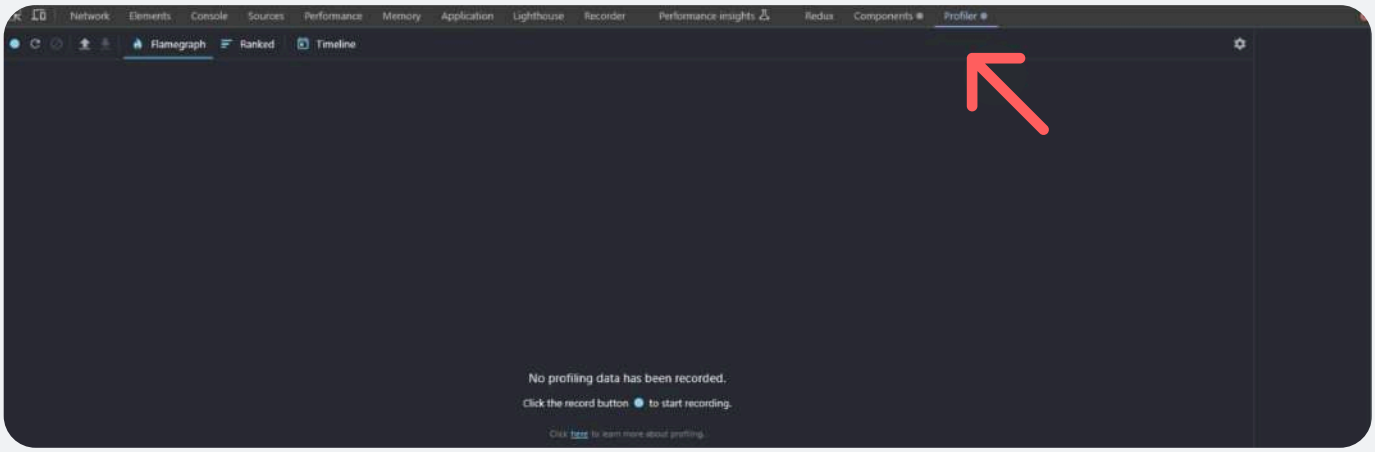


Bonus: How to do debugging?

You already know some hints on how to impact React Flow apps' performance. Now, let's see how to check if your app has any performance issues.

Performance analysis

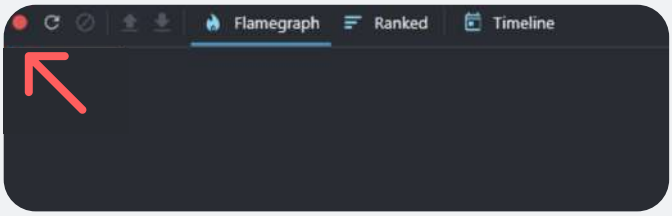
I usually use React Developer Tools—React, especially the Profiler module, to analyze an app's performance. You can find it (after installation) in your browser in Developer Tools.



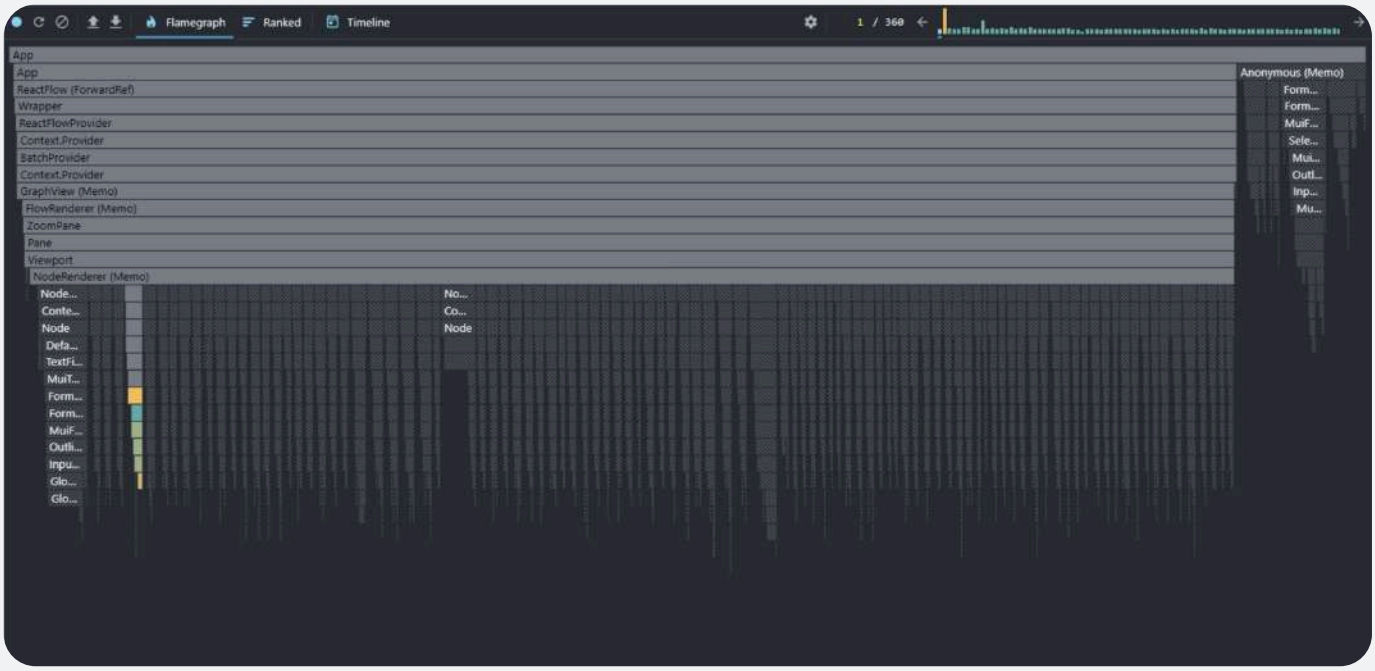
My workflow of analyzing with Profiler looks as follows:

1. Start an application and add several dozen nodes on a diagram (e.g., 30).
2. Turn on recording in Profiler.

3. Choose one node and drag it for a few seconds.
4. Stop the recording in Profiler.



5. Go to the Flamegraph section.



How to interpret results on Flamegraph:

- Dark-grey elements are components that haven't been rendered.
- Vertical, narrowing down "teeth" under the `NodeRenederer` group represent node components. If you hover over any node, it highlights it in an application, making it easy to identify.
- An optimized application will highlight only one node on a graph (the dragged one), while a not optimized one will display many elements in the same frame. You can change frames in Profiler's right upper corner—on a colorful bar chart.

Below, you can see a Flamegraph of an application that wasn't optimized. In a rendering frame, you can observe many more highlighted nodes. This means that while one of them was dragged, the rest also re-rendered.

In a similar way, you can look for not optimized UI components.

The main goal is to have a case in which, while moving only one element, only this element and its edges are re-rendered (unless business requirements dictate different behavior).

Identifying bottlenecks

There's no one perfect way to identify bottlenecks, but my approach is as follows:

1. Analyzing communication with a store.
   a. Check if selectors are memoized and if a state is effectively managed (e.g., with a Zustand).
2. Analyzing references passed to `<ReactFlow>`.
   a. Make sure that you use `useCallback` and `useMemo` and check their dependencies in dependencyArray.
3. Nodes and edges memoization.
   a. Check if nodes and edges are wrapped in `React.memo`.

4. In less complex projects: Review of using state in application.
    a. Check manually all state usage according to good practices.
5. In complex projects:
    a. Perform a binary search for `<ReactFlow>`.
        I. Comment out half of the props passed to `<ReactFlow>` and check if it improved performance.
        II. If so, uncomment half of the props and search further.
        III. If not, comment out half of the remaining props and search further.
    b. Nodes and edges contents:
        I. Comment out their contents to check if the source of performance issue lies in rendered elements' complexity.
        II. Use binary search, as I mentioned above.
        III. Check the usage of 3rd party libraries in components.
    c. The application's main panels:
        I. Comment out everything besides a diagram (and eventually search for an issue with a binary search).

Sometimes, a bottleneck hides in one place, and one of the above points will help. However, there might be also several bottlenecks and then you have to combine some of those techniques.

Want to discuss how to optimize your React Flow project? Reach out to me on LinkedIn.

Łukasz Jaźwa
CTO at Synergy Codes

Contact me →

synergy codes

synergycodes.com