Using Generators in React Components

Tomasz Gil

Aug 18, 2021 • 9 min read

Most of the problems that you encounter when building user interface components are relatively simple. Some state management, some DOM manipulation, some side effect handling. Well-established libraries or frameworks come with tools or

abstractions that let you handle these. But some problems seem to be more complex than that and require a bit more work.

I recently came across one problem in UI which required a specific approach to state updates React doesn't support out of the box. One way it could be solved was with JavaScript generators.

> *If you need a refresher or an introduction to iterators and generators in JavaScript, I recently published a blog post on this topic.*

Our main character - breadcrumbs

I suppose you know what the breadcrumb component is, or if not, you for sure came across one. It typically displays the current page name, with some part of or the entire hierarchy of pages it belongs to. In general, it's used to represent a path to any nested object.

We're gonna build a simplified version of that today! And we will use a generator function. Sounds cool?

As you can see in the example, we want our breadcrumbs to render a list of elements. Additionally, if the entire path is longer than the available space, we want to display only the last items that can fit plus an ellipsis indicating that the path is longer. Just like so.

... / Breadcrumbs / **Default**

Implementing the component

First, let's decide on the API that the component will have. I went with two components, one acting as a container and the other representing a breadcrumb item.

COPY

```
<Breadcrumbs>
  <BreadcrumbItem key="home">Home</BreadcrumbItem>
  <BreadcrumbItem key="docs">Documentation</BreadcrumbItem>
  <BreadcrumbItem key="crumbs">Breadcrumbs</BreadcrumbItem>
  <BreadcrumbItem key="default">Default</BreadcrumbItem>
</Breadcrumbs>
```

It's worth noting already that we will need these `key` properties to properly rerender these breadcrumbs dynamically in the future, whenever `children` change.

Let's then go to the implementation, shall we? Since breadcrumbs is a navigation component, the `BreadcrumbItem` component can simply be a link tag.

COPY

```
export function BreadcrumbItem({ children, ...props }) {
  return <a {...props}>{children}</a>;
}
```

The `Breadcrumbs` component will be where the fun stuff happens. Let's define the component then.

We know that we will get our items as `children`. Depending on how much space there is, we might not render all of them, so let's have a variable that will store the number of visible items. Since changing this should trigger a rerender, let's use state, starting with the initial children count passed. We will handle updating it based on calculating the overflow later.

COPY

```
export function Breadcrumbs({ children }) {
  const childrenCount = React.Children.count(children);
  const [visibleItemsCount, setVisibleItemsCount] = useState(childrenCou
}
```

What's left is the return statement. We know how many elements should be visible - `visibleItemsCount`. Also, when the number of items visible is lower than the number of items passed, we will render an ellipsis.

COPY

```
export function Breadcrumbs({ children }) {
  const childrenCount = React.Children.count(children);
  const [visibleItemsCount, setVisibleItemsCount] = useState(childrenCou

  const shouldRenderStack = childrenCount > visibleItemsCount;
  const visibleChildren = shouldRenderStack
    ? children.slice(-visibleItemsCount)
    : children;

  return (
    <nav>
      <ol className="list">
        {shouldRenderStack && <li className="crumb">...</li>}

        {React.Children.map(visibleChildren, (item) => (
          <li key={item.key} className="crumb">
            {item}
```

```
          </li>
        ))}
      </ol>
    </nav>
  );
}
```

Awesome! Now let's think about how we can approach the overflow.

Handling overflowing elements

When we get new `children` we know that we should update the overflowing elements in our component. We want to do this as soon as we render the component. For taking measurements React gives us a hook - `useLayoutEffect`. It's similar to `useEffect`, although it runs a bit earlier, right after React has performed all DOM mutations.

COPY

```
export function Breadcrumbs({ children }) {
  const childrenCount = React.Children.count(children);
  const [visibleItemsCount, setVisibleItemsCount] = useState(childrenCou

  const updateOverflow = useCallback(() => {}, []);

  useLayoutEffect(updateOverflow, [children, updateOverflow]);

  const shouldRenderStack = childrenCount > visibleItemsCount;
  const visibleChildren = shouldRenderStack
    ? children.slice(-visibleItemsCount)
    : children;

  return (
    ...
```

```
    );
  }
```

Now, the `updateOverflow` function has to do two things:

- measure the elements so that we know how many items we can fit
- update the `visibleItemsCount` based on that information

The problem here is that we can only measure elements that are actually present in the DOM. If some items are already folded, we can't measure them because they're not rendered. We could make an invisible copy of the entire component, just for measuring, but let's assume we don't want to use this technique.

This way, we can't perform the update in a single run without making assumptions on how wide elements are (which we don't want to make either, since that would make the component implementation depend on styles). So how about we do the following.

1. Render all elements.
2. Measure how many will fit in the available space.
3. Render the elements that fit.

We would do this for each change that happens to our breadcrumbs' children. In our case, rendering certain items means updating the number of visible items. And as you see, this requires making two sequential state updates, one after another.

Making sequential state updates

This approach to the breadcrumbs component is based on an open-source UI library called React Spectrum coming from Adobe (here you can see the breadcrumbs source code).

Let's come back to our `updateOverflow` implementation. Following what we've established before, let's have a `computeVisibleItems` that takes currently visible items count and returns the number of items that fit in the available space. The implementation is not key to understanding the whole concept, so I'll skip it (the full implementation is available in this gist).

COPY

```
export function Breadcrumbs({ children }) {
  const childrenCount = React.Children.count(children);
  const [visibleItemsCount, setVisibleItemsCount] = useState(childrenCou

  const updateOverflow = useCallback(() => {
    function computeVisibleItems(currentVisibleItemsCount) {
      let newItemsCount = 0;
      // calculations...
      return newItemsCount;
    }
  }, []);

  useLayoutEffect(updateOverflow, [children, updateOverflow]);

  ...
}
```

With this, we can perform the updates. Like we said, let's first set the visible items to all passed children, then calculate the items and set visible items again.

COPY

```
export function Breadcrumbs({ children }) {
  const childrenCount = React.Children.count(children);
  const [visibleItemsCount, setVisibleItemsCount] = useState(childrenCou
```

```
  const updateOverflow = useCallback(() => {
    function computeVisibleItems(currentVisibleItemsCount) {
      let newItemsCount = 0;
      // calculations...
      return newItemsCount;
    }

    setVisibleItemsCount(childrenCount);
    const newVisibleItems = computeVisibleItems(childrenCount);
    setVisibleItemsCount(newVisibleItems);
  }, [setVisibleItemsCount, childrenCount]);

  useLayoutEffect(updateOverflow, [children, updateOverflow]);

  ...
}
```

There's one problem with this. It might work in some cases, but there's no guarantee it will work every time. The reason is that we don't wait for the first update to be applied and cause a rerender before we run `computeVisibleItems` and take measurements.

To be sure that the previous update is reflected in the DOM, we would need to perform the consecutive updates as consecutive layout effects.

Ok, so let's change the approach a bit. Let's declare our updating algorithm as a generator function. Let's also change the `useState` to `useValueEffect` which we will implement in a minute.

COPY

```
export function Breadcrumbs({ children }) {
  const childrenCount = React.Children.count(children);
  const [visibleItemsCount, setVisibleItemsCount] = useValueEffect(
    childrenCount
  );
```

```
  const updateOverflow = useCallback(() => {
    function computeVisibleItems(currentVisibleItemsCount) {
      let newItemsCount = 0;
      // calculations...
      return newItemsCount;
    }

    setVisibleItemsCount(function* () {
      yield childrenCount;
      const newVisibleItems = computeVisibleItems(childrenCount);
      yield newVisibleItems;
    });
  }, [setVisibleItemsCount, childrenCount]);

  useLayoutEffect(updateOverflow, [children, updateOverflow]);

  ...
}
```

Extracting to a custom hook

The idea behind `useValueEffect` is that it will work similar to `useState` - it will expose the state and a way to change it. But instead of direct state changes, we will allow the user to declare a sequence of state changes - in a form of a generator.

Why a generator you might ask? It allows the client to declare the sequence of state values (using `yield` keyword). On the other end, we have full control over how we progress through these values. And this control is exactly what we need - the hook internally will make sure that each state change happens only after a layout effect.

Let's declare the hook then. We still need a regular state inside to store the value. Additionally, we'll return a function that will accept a generator function, execute it

and store the generator object as our effect (in a `ref` since changing it should not trigger a rerender).

COPY

```
function useValueEffect(defaultValue) {
  const [value, setValue] = useState(defaultValue);
  const effect = useRef(null);

  const queue = useCallback(
    (fn) => {
      effect.current = fn();
    },
    [effect]
  );

  return [value, queue];
}
```

Awesome, let's now implement a mechanism that will move the generator. We will implement this as a function stored in another `ref` so we can easily call it and make sure it has access to the correct value.

The mechanism itself is fairly straightforward. First, get the next value. Reset the effect if we're done. If we got a value and it's different than the current one, update the state (or if the value is the same, just move on).

COPY

```
function useValueEffect(defaultValue) {
  const [value, setValue] = useState(defaultValue);
  const effect = useRef(null);

  const nextRef = useRef(null);

  nextRef.current = () => {
```

```
    const newValue = effect.current.next();

    if (newValue.done) {
      effect.current = null;
      return;
    }

    if (value === newValue.value) {
      nextRef.current();
    } else {
      setValue(newValue.value);
    }
  };

  const queue = useCallback(
    (fn) => {
      effect.current = fn();
    },
    [effect]
  );

  return [value, queue];
}
```

Cool, we need two more things. One is staring our sequence of updates and the second one is moving our generator to consecutive yields as a layout effect which this hook is all about. Let's do this!

COPY

```
function useValueEffect(defaultValue) {
  const [value, setValue] = useState(defaultValue);
  const effect = useRef(null);

  const nextRef = useRef(null);
```

```
    nextRef.current = () => {
      ...
    };

    useLayoutEffect(() => {
      if (effect.current) {
        nextRef.current();
      }
    });

    const queue = useCallback(
      (fn) => {
        effect.current = fn();
        nextRef.current();
      },
      [effect, nextRef]
    );

    return [value, queue];
  }
```

Performing one more calculation

There's one edge case we have yet to cover. Let's look at the following example.

1. We have 300 px of available space, and 4 items, each of them 100 px wide.
2. On the first run, we render all 4 elements, calculate how many of these we can fit and we get that we can fit only 3 (since `3 * 100px = 300px`).
3. We update the visible items count to 3.

But now, since we aren't rendering all children, we also render the ellipsis, which we didn't include in the calculation because it wasn't rendered!

We need to add one more step to our algorithm. Since we have an easy way of declaring that using a generator, extending it becomes trivial.

COPY

```
export function Breadcrumbs({ children }) {
  ...

  const updateOverflow = useCallback(() => {
    function computeVisibleItems(currentVisibleItemsCount) {
      let newItemsCount = 0;
      // calculations...
      return newItemsCount;
    }

    setVisibleItemsCount(function* () {
      yield childrenCount;
      const newVisibleItems = computeVisibleItems(childrenCount);
      yield newVisibleItems;

      if (newVisibleItems < childrenCount) {
        yield computeVisibleItems(newVisibleItems);
      }
    });
  }, [setVisibleItemsCount, childrenCount]);

  useLayoutEffect(updateOverflow, [children, updateOverflow]);

  ...
}
```

And we're done! Full code from this example is available in this gist and this sandbox, including the function for calculating the number of items visible.

Conclusion

Our component could be further improved, making the folded items available to select or updating the overflow also when resize happens. Regardless, the core mechanism stays the same.

This mechanism could be also implemented in different ways, e.g. using a `ref` for storing the step of our overflow algorithm we've reached. Having it defined as one function though is arguably cleaner, far easier to reuse and extend.

Hope this article gives you an idea of how you can approach sequential state updates in React component using generators!

Further reading and references

Full code available in this gist and this sandbox

Photo by Austin Ban on Unsplash

JavaScript React UI state ReactHooks

WRITTEN BY

Tomasz Gil

+ Follow

I'm a software engineer focused on problem solving, lifelong learner based in Poznań, Poland. Currently working at Rvvup as Senior Frontend Engineer.

MORE ARTICLES

Tomasz Gil

7 Lessons From The Software Craftsman

After many decades of software engineering that produced all the advanced tools and methodologies, m...

Tomasz Gil

Wordle in Remix: Part 10 – Error Handling

This is the tenth and last article in a series where we create a Wordle clone in Remix! 💿 We went s...

Tomasz Gil

Wordle in Remix: Part 9 – Validation

This is the ninth article in a series where we create a Wordle clone in Remix! 💿 We go step by step...

Comments (2)

+ Write a comment

gadi tzkhori

Oct 3, 2021

Wow such a creative solution! I had a similar issue but a bit different I think, Tag components overflowing parent container. Bailed out with intersectionObserver. Please revise my package and tell me what you think react-truncate-jsx

Reply 1 1 1

Tomasz Gil

Oct 4, 2021

gadi tzkhori yup, that looks like a similar problem! Didn't want to deal with intersection observer directly though, so I used a hook for that.

Generally, found a lot of help from this library from Adobe, I really liked the idea.

Like