React documentation on the importance of keys in reconciliation: Keys

share|improve this answer
edited Jun 4 at 18:16
answered Feb 4 '15 at 19:16

jmingov
7,16822332

- 5
  I am running into the exact same error. Was this resolved after the chat? If so, can you please post an update to this question. – Deke Feb 26 '16 at 23:04
- 1
  The answer works, Deke. Just make sure that `key` value for the prop is unique for each item and you're putting the `key` prop to the component which is closest to array boundaries. For example, in React Native, first I was trying to put `key` prop to `<Text>` component. However I had to put it to `<View>` component which is parent of `<Text>`. if Array == [(View > Text), (View > Text)] you need to put it to View. Not Text. – scaryguy Jul 28 '16 at 0:52
- 92
  Why is is so hard for React to generate unique keys itself? – Davor Lucic Aug 9 '16 at 13:16
- 7
  @DavorLucic, here is a discussion: github.com/facebook/react/issues/1342#issuecomment-39230939 – koddo Oct 6 '16 at 11:40
- 3
  This is pretty much the official word on a note made in the issue chat linked to above: keys are about identity of a member of a set and auto-generation of keys for items emerging out of an arbitrary iterator probably has performance implications within the React library. – sameers Jan 11 '17 at 22:05

| show **11** more comments

up vote 250 down vote
+500

Be careful when iterating over arrays!!

It is a common misconception that using the index of the element in the array is an acceptable way of suppressing the error you are probably familiar with:

```
Each child in an array should have a unique "key" prop.
```

However, in many cases it is not! This is **anti-pattern** that can in *some* situations lead to **unwanted behavior**.

Understanding the `key` prop

React uses the `key` prop to understand the component-to-DOM Element relation, which is then used for the reconciliation process. It is therefore very important that the key *always* remains *unique*, otherwise there is a good chance React will mix up the elements and mutate the incorrect one. It is also important that these keys *remain static* throughout all re-renders in order to maintain best performance.

That being said, one does not *always* need to apply the above, provided it is *known* that the array is completely static. However, applying best practices is encouraged whenever possible.

A React developer said in this GitHub issue:

> - key is not really about performance, it's more about identity (which in turn leads to better performance). randomly assigned and changing values are not identity
> - We can't realistically provide keys [automatically] without knowing how your data is modeled. I would suggest maybe using some sort of hashing function if you don't have ids
> - We already have internal keys when we use arrays, but they are the index in the array. When you insert a new element, those keys are wrong.

In short, a `key` should be:

- **Unique** - A key cannot be identical to that of a sibling component.
- **Static** - A key should not ever change between renders.

Using the `key` prop

As per the explanation above, carefully study the following samples and try to implement, when possible, the recommended approach.

Bad (Potentially)

```
<tbody>
    {rows.map((row, i) => {
        return <ObjectRow key={i} />;
    })}
</tbody>
```

This is arguably the most common mistake seen when iterating over an array in React. This approach isn't technically *"wrong"*, it's just... *"dangerous"* if you don't know what you are doing. If you are iterating through a static array then this is a perfectly valid approach (e.g. an array of links in your navigation menu). However, if you are adding, removing, reordering or filtering items, then you need to be careful. Take a look at this detailed explanation in the official documentation.

```
class MyApp extends React.Component {
  constructor() {
    super();
    this.state = {
      arr: ["Item 1"]
    }
  }

  click = () => {
    this.setState({
      arr: ['Item ' + (this.state.arr.length+1)].concat(this.state.arr),
    });
  }

  render() {
    return(
      <div>
        <button onClick={this.click}>Add</button>
        <ul>
          {this.state.arr.map(
            (item, i) => <Item key={i} text={"Item " + i}>{item + " "}</Item>
          )}
        </ul>
      </div>
    );
  }
}
```

```
const Item = (props) => {
  return (
    <li>
      <label>{props.children}</label>
      <input value={props.text} />
    </li>
  );
}

ReactDOM.render(<MyApp />, document.getElementById("app"));
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.1.0/react.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.1.0/react-dom.min.js"></script>
<div id="app"></div>
```

In this snippet we are using a non-static array and we are not restricting ourselves to using it as a stack. This is an unsafe approach (you'll see why). Note how as we add items to the beginning of the array (basically unshift), the value for each `<input>` remains in place. Why? Because the `key` doesn't uniquely identify each item.

In other words, at first `Item 1` has `key={0}`. When we add the second item, the top item becomes `Item 2`, followed by `Item 1` as the second item. However, now `Item 1` has `key={1}` and not `key={0}` anymore. Instead, `Item 2` now has `key={0}`!!

As such, React thinks the `<input>` elements have not changed, because the `Item` with key `0` is always at the top!

So why is this approach only *sometimes* bad?

This approach is only risky if the array is somehow filtered, rearranged, or items are added/removed. If it is always static, then it's perfectly safe to use. For example, a navigation menu like `["Home", "Products", "Contact us"]` can safely be iterated through with this method because you'll probably never add new links or rearrange them.

In short, here's when you can **safely** use the index as `key`:

- The array is static and will never change.
- The array is never filtered (display a subset of the array).
- The array is never reordered.
- The array is used as a stack or LIFO (last in, first out). In other words, adding can only be done at the end of the array (i.e push), and only the last item can ever be removed (i.e pop).

Had we instead, in the snippet above, **pushed** the added item to the end of the array, the order for each existing item would always be correct.

Very bad

```
<tbody>
    {rows.map((row) => {
        return <ObjectRow key={Math.random()} />;
    })}
</tbody>
```

While this approach will probably guarantee uniqueness of the keys, it will *always* force react to re-render each item in the list, even when this is not required. This a very bad solution as it greatly impacts performance. Not to mention that one cannot exclude the possibility of a key collision in the event that `Math.random()` produces the same number twice.

> Unstable keys (like those produced by `Math.random()`) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

Very good

```
<tbody>
    {rows.map((row) => {
        return <ObjectRow key={row.uniqueId} />;
    })}
</tbody>
```

This is arguably the best approach because it uses a property that is unique for each item in the dataset. For example, if `rows` contains data fetched from a database, one could use the table's Primary Key (*which typically is an auto-incrementing number*).

> The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys

---

Good

```
componentWillMount() {
  let rows = this.props.rows.map(item => {
    return {uid: SomeLibrary.generateUniqueID(), value: item};
  });
}

...

<tbody>
    {rows.map((row) => {
        return <ObjectRow key={row.uid} />;
    })}
</tbody>
```

This is also a good approach. If your dataset does not contain any data that guarantees uniqueness (*e.g. an array of arbitrary numbers*), there is a chance of a key collision. In such cases, it is best to manually generate a unique identifier for each item in the dataset before iterating over it. Preferably when mounting the component or when the dataset is received (*e.g. from `props` or from an async API call*), in order to do this only *once*, and not each time the component re-renders. There are already a handful of libraries out there that can provide you such keys. Here is one example: react-key-index.

share|improve this answer
edited Apr 27 at 11:51
answered May 10 '17 at 12:44

Chris
26.7k125375

- 32
  much much better than official docs!!!! – TechTurtle May 31 '17 at 21:08
- In the official docs, they use `toString()` to convert to string instead of leaving as number. Is this important to remember? – skube Aug 3 '17 at 13:10
- @skube, no, you can use integers as `key` as well. Not sure why they are converting it. – Chris Aug 3 '17 at 13:15
- 1
  I guess you *can* use integers but *should* you? As per their docs they state "...best way to pick a key is to use a **string** that uniquely identifies..." (emphasis mine) – skube Aug 3 '17 at 15:37
- 1
  @skube, yes that is perfectly acceptable. As stated in the examples above, you can use the item's index of the iterated array (and that's an integer). Even the docs state: *"As a last resort, you can pass item's index in the array as a key"*. What happens though is that the `key` always ends up being a String anyway. – Chris Aug 3 '17 at 19:10

| show **4** more comments
up vote 2 down vote