Lists & Keys

Rendering a list with map(), what the key prop is, why it matters, and the most common mistake -- using the index as the key -- with simple examples.

Beginner 10 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/lists-and-keys

---

Lists & Keys

So far, every example has shown just one thing -- a single counter, a single user. But real apps usually show a LIST: a list of products, a list of comments, a list of tasks. This lesson covers rendering lists in React and why the `key` prop matters.

Rendering a List with map()

To turn an array into JSX elements on screen, you use JavaScript's `map()` function:

```
function MapRenderListExample() {
  const fruits = ["Apple", "Pear", "Banana"];

  // map() converts each item in an array into a JSX element; the result
  // is again an array (this time made up of JSX elements).
  return (
    <ul>
      {fruits.map((fruit) => (
        <li key={fruit}>{fruit}</li>
      ))}
    </ul>
  );
}
```

`map()` turns every element in an array into something else and returns a new array -- here, we're turning each fruit string into an `<li>` element. The result is an array of JSX elements, which React can render directly.

What Is the key Prop?

When rendering a list with `map()`, you need to give each element a `key` prop:

```
function KeyPropExample() {
  const users = [
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" },
    { id: 3, name: "Clara" },
  ];

  // The key should, whenever possible, be a PERMANENT and UNIQUE identifier
  // specific to that item -- not its name or index -- such as an id
  // coming from the database, as here.
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

`key` is a string or number that UNIQUELY identifies that item within the list. Whenever possible, use a stable identifier like an `id` from your database, not the item's name or its index.

Why Keys Matter

`key` tells React which list item on the next render is the SAME as which item from the previous render:

```
function WhyKeysMatterExample() {
  const tasks = [
    { id: 1, text: "Buy milk" },
    { id: 2, text: "Buy bread" },
  ];

  // The key tells React "which item from the previous render this list
  // item is the same as". When an item is added/removed or the order
  // changes, React uses this to update the correct item -- without a key
  // (or with the wrong key), React might update the wrong item, causing
  // unexpected visual bugs (e.g. an input shifting to the wrong row).
  return (
    <ul>
      {tasks.map((task) => (
        <li key={task.id}>{task.text}</li>
      ))}
    </ul>
  );
}
```

As long as the list doesn't change, `key` doesn't seem to matter. But when an item is added, removed, or the order changes, React uses `key` to figure out which item "stayed the same" and which is new -- so it only updates the part that actually changed, instead of rebuilding the whole list.

Common Mistakes

The most common mistake is using the array index as the `key`:

```
function CommonKeyMistakeExample() {
  const items = ["Apple", "Pear", "Banana"];

  return (
    <ul>
      {/* COMMON MISTAKE: using the index as the key. It appears to work
          as long as the list never changes, but when an item is
          added/removed or the order changes, every item's index shifts --
          React may now think the wrong item is "the same item". */}
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

This looks like it works as long as the list never changes. But once an item is added or removed, or the order changes, every item's index shifts -- React can no longer tell, just by looking at the index, whether it's looking at the same item or a different one. This can cause visual bugs, especially in lists with inputs (like checkboxes in a task list), where the wrong item ends up being updated. Use a stable identifier like `id` whenever you can; if you truly have no stable identifier and the list is never reordered or changed, the index can be a last resort.

Summary and Glossary

You render an array on screen using `map()`; each item gets a unique `key` so React can correctly track changes in the list. Whenever possible, use a stable `id` as the `key`, not the array's index -- using the index can cause the wrong items to be updated when the list changes.

Glossary

`map()` — A JavaScript array method that turns every item in an array into something else and returns a new array; used in React to turn lists into JSX elements.

`key` — A unique identifier given to each item in a list, letting React track items across renders.

Reconciliation — The process React uses to find the difference between the previous render and the new one and update only what actually changed; `key` is what lets this process match up list items correctly.

Practical Project

There's a real, runnable example project that brings together the concepts from this category (Events, State, Conditional Rendering, Lists & Keys): **State & Events Demo** -- a simple task list app.

It shows event handlers (`onClick` / `onChange` / `onSubmit`), `useState` , updating based on the previous state, state immutability, conditional rendering with a ternary/ `&&` , and rendering a list with `map()` + `key` , all working together. You can download it and run it yourself, and read through the code line by line:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/state-events
npm run dev
```

The `react-course-projects` repo uses **npm workspaces** -- `npm install` only needs to run once, at the repo root, and every project folder shares the same dependencies (no separate `node_modules` per folder). If you've already run `npm install` at the root, you can just `cd react-course-projects/projects/state-events` and run `npm run dev` .