createContext

`createContext` lets you create a context that components can provide or read.

```
const SomeContext = createContext(defaultValue)
```

Reference

`createContext(defaultValue)`

Call `createContext` outside of any components to create a context.

```
import { createContext } from 'react';
```

```
const ThemeContext = createContext('light');
```

See more examples below.

Parameters

- `defaultValue`: The value that you want the context to have when there is no matching context provider in the tree above the component that reads context. If you don't have any meaningful default value, specify `null`. The default value is meant as a "last resort" fallback. It is static and never changes over time.

Returns

`createContext` returns a context object.

The context object itself does not hold any information. It represents *which* context other components read or provide. Typically, you will use `SomeContext.Provider` in components above to specify the context value, and call `useContext(SomeContext)` in components below to read it. The context object has a few properties:

- `SomeContext.Provider` lets you provide the context value to components.
- `SomeContext.Consumer` is an alternative and rarely used way to read the context value.

`SomeContext.Provider`

Wrap your components into a context provider to specify the value of this context for all components inside:

```
function App() {
  const [theme, setTheme] = useState('light');
```

```
  // ...
  return (
    <ThemeContext.Provider value={theme}>
      <Page />
    </ThemeContext.Provider>
  );
}
```

Props

- `value`: The value that you want to pass to all the components reading this context inside this provider, no matter how deep. The context value can be of any type. A component calling `useContext(SomeContext)` inside of the provider receives the `value` of the innermost corresponding context provider above it.

`SomeContext.Consumer`

Before `useContext` existed, there was an older way to read context:

```
function Button() {
  // 🟡 Legacy way (not recommended)
  return (
    <ThemeContext.Consumer>
      {theme => (
        <button className={theme} />
      )}
    </ThemeContext.Consumer>
  );
}
```

Although this older way still works, but **newly written code should read context with `useContext()` instead:**

```
function Button() {
  // ✅ Recommended way
  const theme = useContext(ThemeContext);
  return <button className={theme} />;
}
```

Props

- `children` : A function. React will call the function you pass with the current context value determined by the same algorithm as `useContext()` does, and render the result you return from this function. React will also re-run this function and update the UI whenever the context from the parent components changes.

Usage

Creating context

Context lets components pass information deep down without explicitly passing props.

Call `createContext` outside any components to create one or more contexts.

```
import { createContext } from 'react';

const ThemeContext = createContext('light');
const AuthContext = createContext(null);
```

`createContext` returns a context object . Components can read context by passing it to `useContext()` :

```
function Button() {
  const theme = useContext( ThemeContext );
  // ...
}

function Profile() {
  const currentUser = useContext( AuthContext );
  // ...
}
```

By default, the values they receive will be the default values you have specified when creating the contexts. However, by itself this isn't useful because the default values never change.

Context is useful because you can **provide other, dynamic values from your components:**

```
function App() {
  const [theme, setTheme] = useState('dark');
  const [currentUser, setCurrentUser] = useState({ name: 'Taylor' });

  // ...

  return (
    <ThemeContext.Provider value={theme}>
      <AuthContext.Provider value={currentUser}>
        <Page />
      </AuthContext.Provider>
    </ThemeContext.Provider>
  );
}
```

Now the `Page` component and any components inside it, no matter how deep, will "see" the passed context values. If the passed context values change, React will re-render the components reading the context as well.

Read more about reading and providing context and see examples.

Importing and exporting context from a file

Often, components in different files will need access to the same context. This is why it's common to declare contexts in a separate file. Then you can use the `export` statement to make context available for other files:

```
// Contexts.js
import { createContext } from 'react';

export const ThemeContext = createContext('light');
export const AuthContext = createContext(null);
```

Components declared in other files can then use the `import` statement to read or provide this context:

```
// Button.js
import { ThemeContext } from './Contexts.js';

function Button() {
  const theme = useContext(ThemeContext);
  // ...
}
```

```
// App.js
import { ThemeContext, AuthContext } from './Contexts.js';

function App() {
  // ...
  return (
    <ThemeContext.Provider value={theme}>
```

```
        <AuthContext.Provider value={currentUser}>
          <Page />
        </AuthContext.Provider>
      </ThemeContext.Provider>
    );
}
```

This works similar to importing and exporting components.

Troubleshooting

I can't find a way to change the context value

Code like this specifies the *default* context value:

```
const ThemeContext = createContext('light');
```

This value never changes. React only uses this value as a fallback if it can't find a matching provider above.

To make context change over time, add state and wrap components in a context provider.

PREVIOUS
cache

NEXT
forwardRef

How do you like these docs?

Take our survey!

Meta Open Source

Learn React

- Quick Start
- Installation
- Describing the UI
- Adding Interactivity
- Managing State
- Escape Hatches

API Reference

- React APIs
- React DOM APIs

Community

- Code of Conduct
- Meet the Team
- Docs Contributors
- Acknowledgements

More

- Blog
- React Native
- Privacy
- Terms