- ⏱ 30 minutes

React Context

At this point, you know how to manage a component's state and pass information down an application's component tree by threading props from parent components to child components. Think of how tedious it could be to pass down information, like current user data or a UI theme, to every component in an application. Instead of threading the information as props from a parent component to its children and grandchildren, you can share information with any of your application's components by using React Context!

React Context gives you a way to pass data through the component tree without having to manually thread props. Context gives developers a convenient way to share and update "global" data across a React application. Now it's time to dive into application-wide data management!

When you finish this article, you should be able to:

- Describe the relationships between provider, consumer, and context
- Create a React provider wrapper component that will manage the value of a context
- Retrieve values from a Context throughout your application using `useContext`
- Create your own React hook to consume a single context every time

Context, Provider, Consumer

Context allows "global" data in a React application and stores a single value. A context's **Provider** is used to wrap a React application to enable all components in the application to access the context's value. A **Consumer** is a React component that reads a context's value. Consumer components must always be nested under Provider components because the Provider must render first (parent components always render before children components).

You'll learn how to create a context, a provider for the context, and context consumers. You'll also learn how to update a context's value.

Using Context

In this demo, you'll create a simple React app that uses Context to allow users to choose what puppy picture to render.

Begin by cloning the starter React app with a folder of `pups` photos from the `Download` link at the bottom of this page.
Run `npm install` and then start your application. You should see a huge and super happy pup running! Take a minute to look at the `App` component in `src/App.js`. `App` is rendering the `PupForm` and `PupImage` components. In the `PupImage` component, the `speedy` pup image is rendered and is not set to change dynamically (yet).
The `PupForm` renders a form that has a `select` dropdown that allows you to select the different pups. Selecting a pup and submitting this form doesn't do anything now, but with the help of context, it will dynamically set the `PupImage` component's image.

Being able to render a pup photo is great, but what about giving your users the chance to select a pup photo to render? Let's allow your users to select a pup photo through a form by passing information through Context!

Creating a Context

The goal is to create a React context that will allow any component to access and update the selected pup.

Make a folder in `src` called `context`. In that folder, create a file called `PupContext.js`.
Use the React.createContext method to create a `PupContext`. In order to create a context, you can simply import the `createContext()` function from `react`, invoke the function to create a Context object, and export the context.

```
// ./src/context/PupContext.js
import { createContext } from 'react';

export const PupContext = createContext();
```

Note that if you invoke the `createContext` method with an argument, the argument will be the context's default value.
Next, you will create a Provider for the `PupContext`.

Creating a Provider

Every context created from the `createContext` method will have a `Provider` component. For example, `PupContext.Provider` is the provider for `PupContext`. If you wrap a

component with this provider, then the component and its descendants will be able to "consume", or read, the context's value. You need to wrap your child components with provider component tags to give them access to the context.
You can add the given provider to the `App` component like so:

```
<PupContext.Provider value={/* some value */}>
  <App />
</PupContext.Provider>
```

But our goal is to have a dynamic context value.

Let's create a provider component that will render `PupContext.Provider` and take the place of `PupContext.Provider` in the above code. If the provider component is named `PupProvider`, the goal is to wrap the `App` component with it like so:

```
<PupProvider>
  <App />
</PupProvider>
```

In the `src/context/PupContext.js` file, create a function component called `PupProvider`. You need this component to render the `PupContext.Provider` with the `App` component nested inside of it. So far, you have only seen components that are rendered like this:

```
<NavBar />
```

How do we create a parent wrapper component that can render its children dynamically? Each React component has a `props` property called `children`. This is a reserved property that holds an array of all the children components wrapped by the parent component.
In the `PupProvider` component, render and interpolate the `props.children` variable wrapped in the `PupContext.Provider` component. Export the `PupProvider` from this file.

```
// ./src/context/PupContext.js
// ...

export function PupProvider(props) {
  return (
    <PupContext.Provider>
      {props.children}
    </PupContext.Provider>
  )
}
```

Congratulations! You created your first wrapper component!

To use the `PupProvider` component, import it into the entry file (`src/index.js`) and wrap `App` with it.

```
<PupProvider>
  <App />
</PupProvider>
```

Dynamic context value

To create a context value that, when changed, will cause consumer components to re-render, you need to add state to the provider wrapper component.

Your wrapper component should set up a `puppyType` state. Import the `banana`, `sleepy`, or `speedy` pup photo to set as the initial state.

```
// ./src/context/PupContext.js
import { createContext, useState } from 'react'
import speedy from '../pups/speedy-pup.jpg';
// ...

export function PupProvider(props) {
  const [puppyType, setPuppyType] = useState(speedy);

  return (
    <PupContext.Provider>
      {props.children}
    </PupContext.Provider>
  )
}
```

Note that you should set the `puppyType` as `speedy` (without quotations) instead of `"speedy"`. Using the version in quotation marks would set the default state to a string instead of a default pup photo to be rendered.
Remember that provider components expect to receive a `value` prop. The `value` prop will hold the context information that will be passed throughout the application. Set the `value` prop of `PupContext.Provider` to an object with keys of `puppyType` and `setPuppyType`. As you know, a component re-renders when its state is updated. This is how you'll update an application's context from nested components.
Your `PupContext` component should look something like this:

```
export function PupProvider(props) {
  const [puppyType, setPuppyType] = useState(speedy);

  return (
    <PupContext.Provider value={{ puppyType, setPuppyType }}>
      {props.children}
    </PupContext.Provider>
  )
}
```

Now that you finished creating and connecting the Provider for the `PupContext`, let's create some context consumers!

useContext

`useContext` is a React hook that allows components to "consume", or read, the value of a given context. If you pass a context into the `useContext` hook, the hook will return the value of that context.
Let's make `PupImage` consume your `PupContext` and render the image set in the context's value!

Import `useContext` from React and `PupContext` from the `PupImage` component file.
Inside the `PupImage` component, invoke `useContext` and pass in the `PupContext` as an argument. Destructure the `puppyType` key from the return value of the `useContext` function. Then set the `src` attribute on the `img` tag to that `puppyType` key.

```
// ./src/components/PupImage/PupImage.js
import { useContext } from 'react';
import { PupContext } from '../../context/PupContext';
import speedy from '../../pups/speedy-pup.jpg';
import banana from '../../pups/banana-pup.jpg';
import sleepy from '../../pups/sleepy-pup.jpg';

const PupImage = () => {
  const { puppyType } = useContext(PupContext);

  return (
    <img src={puppyType} alt="pup" />
  );
};

export default PupImage;
```

Try changing the default `puppyType` coded in the `PupProvider`. When you refresh, you should see the pup image corresponding to the `puppyType` you set in the `PupProvider`!

Update the context value

Now, let's try changing the context value from a different component.
The `PupForm` component is not a parent or a child of `PupImage`, so the context value cannot be sent as `props`. Instead, you will "consume", or access, the value of the `PupContext` using the same `useContext` hook in the `PupForm`.
The `PupForm` will render a `select` dropdown that will update the `selectedPup` component state variable to the pup image selected by the user. When the form is submitted, the context value should change to the pup image selected.
Import `useContext` from React and the `PupContext` in the `PupForm` component file.
Inside the `PupForm` component, invoke `useContext` and pass in the `PupContext` as an argument. Destructure the `puppyType` and `setPuppyType` keys from the return value of the `useContext` function.
Initialize `selectedPuppy` to `puppyType`. Then, when the form is submitted, invoke `setPuppyType` with the `selectedPup` state variable.

```
// ./src/components/PupForm/PupForm.js
import { useState, useContext } from 'react';
import { PupContext } from '../../context/PupContext';
import banana from '../../pups/banana-pup.jpg';
import sleepy from '../../pups/sleepy-pup.jpg';
import speedy from '../../pups/speedy-pup.jpg';
```

```
function PupForm() {
  const { puppyType, setPuppyType } = useContext(PupContext);
  const [selectedPup, setSelectedPup] = useState(puppyType);

  const onSubmit = (e) => {
    e.preventDefault();
    setPuppyType(selectedPup);
  }

  return (
    <form onSubmit={onSubmit}>
      <select
        name="pup"
        onChange={e => setSelectedPup(e.target.value)}
        value={selectedPup}
      >
        <option value="select" disabled>Select a pup!</option>
        <option value={speedy}>Speedy Pup</option>
        <option value={banana}>Banana Pup</option>
        <option value={sleepy}>Sleepy Pup</option>
      </select>
      <button>
        Submit
      </button>
    </form>
  );
}

export default PupForm;
```

Now you should be able to use the dropdown menu to select a pup photo to render! Congratulations, you now know how to use React Context to share and update global information across your application!

`useContext` will cause your component to re-render if the value of the context changes. The flow of context will look something like this:

- The context value is set by the provider and will be stored and maintained as a state variable in the provider component
- The context value can be read by any consumer components
- When the value of the provider component's state variable changes, the provider component will re-render and the context's value will change
- The consumer components are subscribed to changes to the context value through the `useContext` hook so they will re-render even though there are no `props` or component state changes.

Create a context hook

Let's refactor your code to create and use your very own React hook! This hook will consume the `PupContext` and return the value of the `PupContext` when invoked. Inside the `src/context/PupContext.js` file, create and export a function called `usePuppyType`. Import the `useContext` hook from React. Return the invocation of `useContext` passing in `PupContext` as an argument.

```
// ./src/context/PupContext.js
import { createContext, useState, useContext } from 'react'
// ...

export function usePuppyType() {
  return useContext(PupContext);
}
```

Now, in the `PupImage` and `PupForm` components, instead of consuming the `PupContext` directly with `useContext`, import the newly created `usePuppyType` function from the `PupContext` file and invoke it in place of `useContext`.

Here's what `PupImage` should look like now:

```
import { usePuppyType } from '../../context/PupContext'

const PupImage = () => {
  const { puppyType } = usePuppyType();
  return (
    <img src={puppyType} alt="pup" />
  );
};

export default PupImage;
```

Here's what `PupForm` should look like now:

```
import { useState } from 'react';
import { usePuppyType } from '../../context/PupContext';
import banana from '../../pups/banana-pup.jpg';
import sleepy from '../../pups/sleepy-pup.jpg';
import speedy from '../../pups/speedy-pup.jpg';

function PupForm() {
  const { puppyType, setPuppyType } = usePuppyType();
  const [selectedPup, setSelectedPup] = useState(puppyType);

  // ...
}

export default PupForm;
```

Amazing! You just created your first React hook! To create a custom hook, you can use one or more of the existing React hooks. This particular hook that you created returns the value of the `PupContext` and will cause the consumer component to re-render because the hook is using `useContext` internally.

What you learned

In this article, you learned what context, a context provider, and a context consumer are, and how to create a React provider wrapper component that manages the value of a context through component state. You also learned how to retrieve values from a context in a component using `useContext`. Finally, you learned how to create your own React hook to consume the same context every time.

Did you find this lesson helpful?

No

Yes

✓