How Data Flows Through React Nested Components

Typically, the communication pattern of a JavaScript framework is to pass data from parent to child components.

React component hierarchy

Data flow can be complicated to manage, especially with a deeply nested tree of React components. That's because data passes through every nesting level, including components that don't need it. This pattern is called **prop drilling** and it can become a challenging architectural problem for developers to tackle.

Prop drilling: data passes through every nesting level from parent to child components

There are two primary tools In the React ecosystem that offer solutions to this challenge: Redux and Context API. In this article, we'll unpack both technologies to understand how they manage data flow and determine if there is a clear choice for using one over the other.

What Is Redux?

Redux is a library for managing and updating the application state using events called **actions**. It's a centralized store that's shared across the entire application and ensures that the state gets updated in a predictable fashion.

Redux manages and updates the application state using actions

The patterns and tools provided by Redux make it easier to understand when and how the state in the application is updated and how its logic will behave when the update occurs. With this, Redux makes it easier to write predictable and testable code.

Also, Redux leverages React Context. In earlier versions of React, Context was still an experimental feature and intended to manage the global state, just like Redux state.

How Does Redux Work?

Redux is an implementation of Flux and consists of four key parts organized as a one-way data pipeline:

Four key components of Redux (Actions, Dispatcher, Store, View) in one-way data pipeline

The View dispatches actions that describe what happened. Then, the Store receives these actions and determines what state change should occur. After the State updates, the View is rendered with the new state.

The Store is where the state is managed centrally. It's responsible for maintaining the state and receiving actions from the View.

The Store handles the state updates with a function called Reducer (a function that receives a state and a dispatched action from the view to return a new state specified by the action). It's important to note that reducer functions must be pure, meaning the State has to be treated as an immutable object and can't be manipulated directly.

Data is managed and stored in the Store with Redux

The View needs to re-render after the update because the State is being modified outside of React. The store implements the observer pattern with an array that uses a subscribe method to add a new listener and call each listener function whenever the state changes.

In the View, we can subscribe to our component when it mounts. The listening function that we pass to subscribe() will call this.force_update() and will trigger a re-render of the component. The view mentioned above is the one in Flux architecture, but when we put it in the React ecosystem the view is contained in the React component.

Benefits of using Redux

If you choose Redux for your project, some key benefits are:

- **It increases the predictability of a state:** Since reducers are pure functions, they always produce the same result when the same action or state is passed to them.
- **It's highly maintainable:** The structure of any Redux application is relatively standardized since code organization follows strict guidelines with this library.

- **It prevents re-renders:** The state is treated as immutable. The new state is derived from the old one using a shallow copy. This reduces the probability of re-renders substantially, therefore having a positive impact on performance.

- **It makes debugging easier:** By logging actions and the state, Redux makes it easy to have insight into what happens during the lifetime of an application. It has excellent DevTools that allow us to time-travel actions, persist actions on page refresh, etc.

- **It's useful in server-side rendering:** The usefulness and effectiveness of Redux in server-side rendering are also well-proven. Handling the initial render of an application is relatively easy with Redux.

- **It's easy to test:** Redux relies on pure reducer functions. It's easy to test pure functions since they always return the same output given the same input.

What Is Context API?

Context API is a different approach to tackling the data flow problem between React's deeply nested components. Context has been around with React for quite a while, but it has changed significantly since its inception. Up to version 16.3, Context was a way to handle the state data outside the React component tree. It was an experimental feature not recommended for most use cases.

Initially, the problem with legacy context was that updates to values that were passed down with context could be "blocked" if a component skipped rendering through the shouldComponentUpdate lifecycle method. Since many components relied on shouldComponentUpdate for performance optimizations, the legacy context was useless for passing down plain data.

The new version of Context API is a dependency injection mechanism that allows passing data through the component tree without having to pass props down manually at every level.

The most important thing here is that, unlike Redux, Context API is not a state management system. Instead, it's a dependency injection mechanism where you manage a state in a React component. We get a state management system when using it with useContext and useReducer hooks.

How Does Context API Work?

Context API is quite simple. You only need to create a state context using the function createContext and it will return a provider and a consumer. The provider wraps the component tree where you expect the descendants to consume the state. The consumer is the wrapper for the location where the state data is used.

Data flow and management with Context API

Benefits of using Context API

Here are some key benefits of choosing Context API for your project:

- **It's scalable.** Context API can be used for any size of web application.
- **It's less complex than Redux.** The workflow is much simpler than Redux. It doesn't involve the additional parts or boilerplate that Redux requires.
- **It has a lower implementation cost.** In cases where we only use it to avoid prop drilling, we can put aside the implementation of reducers.
- **There's no need to pass data to the children at each level.** The Consumer component can access all the data provided by the Provider Component at any level. This prevents prop drilling.
- **It's easy to maintain and very reusable.** As no prop drilling takes place, if we remove a component from the tree or place it to another level, those components below will not be affected.

- **Integration of React's modules is seamless.** Because it's part of the core React library, we don't need to install, import or maintain any additional libraries.

Context API vs Redux: When to Use One Over the Other

As Context API and Redux are ultimately used to build web applications, the three main goals are:

1. A fast response time
2. Easy to develop
3. Easy to maintain

Let's break down how these technologies perform for each of these goals.

Response Time

An important aspect of response time is the time spent on initial load. That time is directly proportional to the amount of data sent from the server and the speed of the network. If we had two apps with similar code running on the same server and the same network, one using Context API and the other Redux, the app using Redux would take longer because it requires external libraries to function. The actual difference is only about 2 kilobytes.

Another response time component is the time spent to respond to user actions. In this case, the number of operations and the speed of the network are the two most important parameters. Redux requires more computational operations to be performed to respond to a user request. However, this difference is negligible.

Therefore, it’s safe to assert that when it comes to response time there’s not a major difference between the two.

Verdict: No major difference in response time.

Ease of Development

When it comes to ease of development, the most important aspect is the number of lines of code that have to be written to carry out an operation. Let’s look at some specific scenarios to get an estimate for this metric.

Scenario 1: Sharing State with Nested Components in React

Let’s imagine a scenario where we want to make some value available to any component in a given React tree without prop drilling. To solve this problem using Context API, we have to create a context with a default value, wrap a high-level component with a provider for that context and use it in one of their children. To compare with Redux, we have to implement the store by creating the object with the initial state, implement the reducer where we handle each action, return the new state, subscribe to the state change and dispatch the action. And so, Redux would require significantly more lines of code to be written to perform a similar action compared to Context API.

How Context API handles sharing State with React nested components

Scenario 2: Building an App with State Management

Another scenario is where we have a moderately complex state management needs within a specific section of our application. In this case, we could combine Context API with getContext and useReducer hooks. The lines of code would then be determined by the implementation of the reducers. Both Redux and Context API have to subscribe to the state in order to re-render the components that have been modified and they both require a place to manage the state. The lines of code required are similar in both cases.

Scenario 3: Building an App with Undo and Redo Functionality

Consider building an app with Undo and Redo functionality. With Redux, implementing undo functionality is rather easy. This is because the state is immutable and mutations are already described as discrete actions, which is close to the undo stack mental model.

How Redux handles Undo and Redo functionality

For any scenario, we simply need to reshape the state into past, present and future values. Here, past and future are stacks where values are popped and pushed with undo and redo actions in the reducer. With Context API, it's not possible to do this unless we augment the API with libraries or custom code that supports this functionality.

Verdict: ease of development with Redux or Context API depends on what you're building.

Ease of Maintenance

It's necessary to understand what happens in an application to easily maintain it. Developer tooling can be very useful for gaining this insight. For example, Redux enables you to inspect every state and action payload, go back in time by "canceling" actions and identify and analyze the error if a reducer throws one. These capabilities means less time is spent understanding the application and fixing bugs. Unfortunately, Context API is not as strong as Redux in this regard.

Application size and complexity are other important aspects to consider when comparing ease of maintenance. While it's difficult to classify application complexity by using the number of components or nested elements as a metric, it's important to note that small and simple applications will perform just fine with Context API. This is because the application state can generally be managed by passing it through components. That said, larger and more complex apps might require you to leverage the getContext and useReducer hooks alongside with the Context API.

Verdict: Redux provides better support for troubleshooting and testing large and complex applications

Final Thoughts

Redux and Context API are great solutions for managing data flow through React's nested components. They both follow a similar philosophy: the state is taken outside the component tree and consumed as needed. In the case of Redux, external dependencies need to be installed, configured and maintained. Context API is integrated into the React core and requires minimal configuration.

Redux is a complete state manager capable of allowing an app to undo/redo actions and provides advanced developer tooling for debugging. Context API is designed as a dependency injection mechanism that allows making data available through the component tree without being manually passed.

For this reason, Redux is far more complex and abstract, with more concepts to learn, while Context API is simpler to learn, has fewer concepts, and is more intuitive. It is great for encapsulating data for specific contexts and not managing it globally. It avoids the inefficiencies of prop drilling while also being significantly more straightforward to implement than Redux. This table summarizes the comparison of context API and Redux:

Table comparing Context API and Redux

Suppose you just wanted a simple data communication mechanism. In that case, Context API could be considered a better choice. At the same time, Redux is better if working on an app with many components, complicated inter-dependencies, dealing with unique features like undo/redo and requiring advanced debugging capabilities.

//

Pass Data With Props

If you want to share data in react you can use props but in react you can share data in one direction in other way

If you have movies application in this app you have three components {Search,Movies,Login} if you want to share data you can just

share in one component because react work with one direction that is problem if you have big project that is why appeared to important state management {Redux,Contex Api}.

Difference Between Context API and Redux

The main difference between these two libraries is that Redux work in centralized manner,

in other way in redux you can set the data in one components in other components have right to acces in data.

On the other hand, Context deals with them as they happen on the component level that's main you shoulkd to share data in components like props.

Difference betwen context api and props

props work with one direction main data share in one component,context api you can share data with 3 or 4 components

Context API

Context provides a way to pass data through the component tree without having to pass props down manually at every level. (source: React)

Context API is a fairly new concept in the world of React. Its main purpose is to share data between components without using props or actions.

It is designed to share data that can be considered global for a tree of React components, such as theme, or preferred language.

Context can significantly reduce the complexity of state management in your application.

It has 2 core concepts:

Provider

Consumer

The job of the provider is to define and track certain pieces of state.

This state can be accessed by all the children nested inside Provider.

These children are usually referred to as Consumers.

Consumer is every component that is accessing or modifying the state from Context Provider.

Redux

Redux is a JavaScript library that helps to manage data flow in a centralized manner. It stores the entire state of the application.

This state can be accessed by any component without having to pass down props from one component to another. It has 3 core concepts:

Actions

Reducers

Store

Actions are events that send data to the Redux store. They can be triggered by user interaction or called directly by your application.

Each action needs to have a unique type and payload associated with it.

Conclusion

In other explain if you have big project it contains a lot of components you should be use redux becaus context api can't wrappe a lot of components.

Context API is a one of the best state management solution for separate disconnected components in React. It solves the complexity of pass data via props between multiple layers of Components. This article is discussed about the Context API and implement a sample To Do mobile application using Context API. The complete code can be taken from here.

When we discuss about the state management, we can say Redux is a great solution. Do you interest about Redux? Then, my previous post will be helpful. You can read it from here.

But Present developers moved to Context API because of its benefits. Redux as a state management tool is needed to install 'react-redux', 'redux' and 'redux-thunk' libraries. Context API is a part of the 'React'. Therefore, you do not want to install any libraries in order to state management process. It shrinks your bundle size.

1. What is Context API?

Context provides a way to pass data through the component tree without having to pass props down manually at every level. by source

Context API provides clean and easy way to share state between the components without passing the props by React itself.

For an example, you have to pass data from-top level component to nth level component. If you do not use global state, you have to pass data as props every and each component up to nth level component. using Context API, you able to simply cover all down components from your since top-level component.

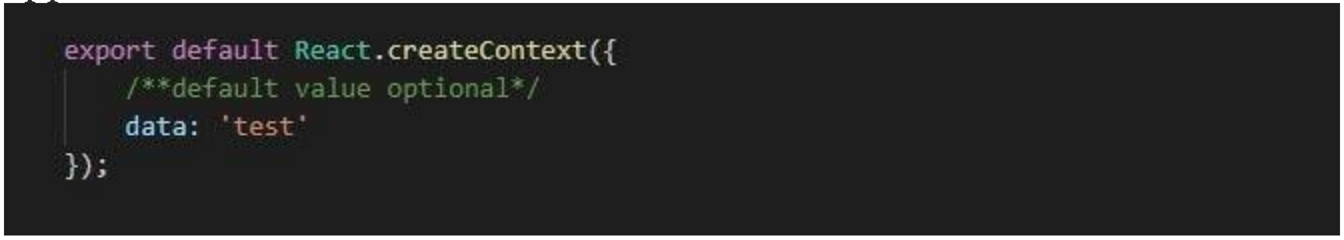
2. Building Block of Context API

We can divide Context API in to three parts.

Context
Provider
Consumer

2.1 Context

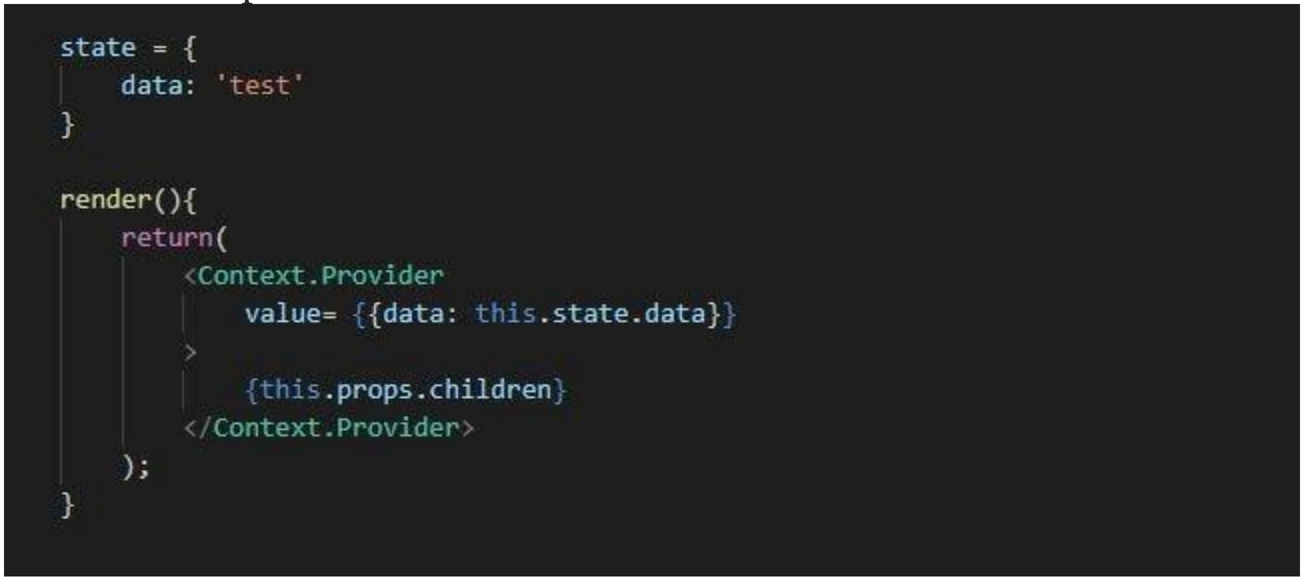
First import React from ‘react’. Then, *createContext()* function from React which takes default value as a first argument. In here optional to pass a Java Script object. There can be multiple context in a single application.



```
export default React.createContext({
    /**default value optional*/
    data: 'test'
});
```

2.2 Provider

After create context, Provider Provides the capability to access context which wrapped from it. It provides the data and functions to pass down to all the components.



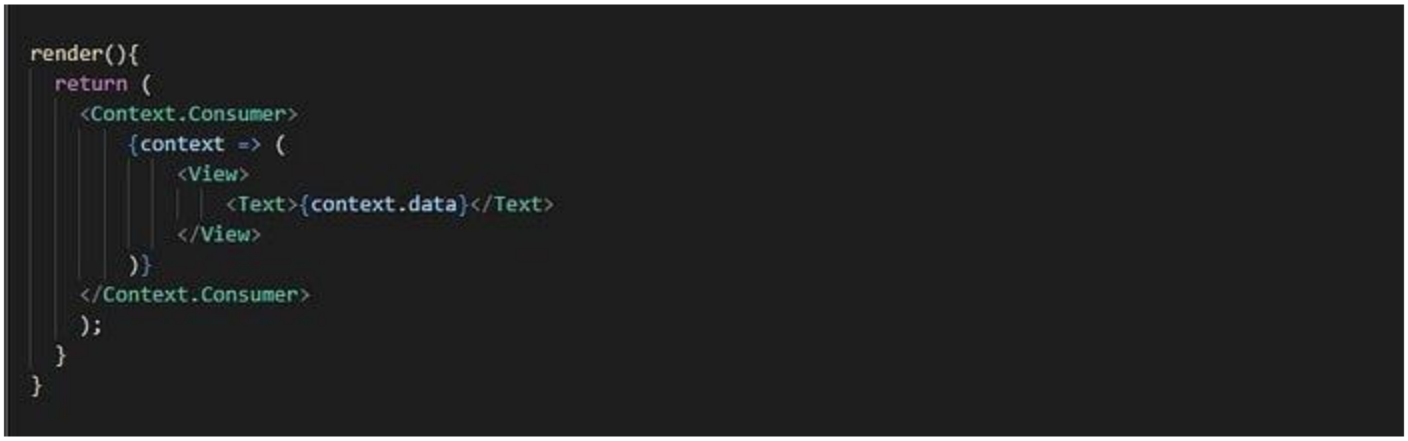
```
state = {
    data: 'test'
}

render(){
    return(
        <Context.Provider
            value= {{data: this.state.data}}
        >
            {this.props.children}
        </Context.Provider>
    );
}
```

2.3 Consumer

Consumer allows to access the value to child components which parent that is wrapped by Provider. It has two types.

1. <Context.Consumer>



```
render(){
  return (
    <Context.Consumer>
        {context => (
            <View>
              <Text>{context.data}</Text>
            </View>
        )}
    </Context.Consumer>
    );
  }
}
```

Context.Consumer can be used both Class based and functional components. How ever, this approach the Context can be access only inside the render.

Creating a global storage with Context API

Create following files:

App.js

```
import React, {useState} from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';
import HomeScreen from './screens/Home';
import DetailsScreen from './screens/Details';
import DisplayScreen from './screens/Display';

const Stack = createNativeStackNavigator();

// create context object, will be used outside App.js
```

```
export const Context1 = React.createContext(null);

function App() {

  const context1InitialState = {
    // can also be null if no default value is to be set
      name: "Ali Khan",
      age: "45",
      city: "Karachi"
    };

    const [personInfo, setPersonInfo] = useState(context1InitialState);

    const context1Setters = {
      setName,
      setAge,
      setCity
    }

    /*
    In an object literal, the spread syntax enumerates the properties
    of an object and adds the key-value pairs to the object being created.
    */


    function setName(name) {
      const newName = { ...personInfo, name };
      setPersonInfo(newName);
    }

    function setAge(age) {
      const newAge = { ...personInfo, age };
      setPersonInfo(newAge);
    }

    function setCity(city) {
      const newState = { ...personInfo, city };
      setPersonInfo(newCity);
    }



  return (
    <Context1.Provider value={{ ...personInfo, ...context1Setters }}>
     <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
        <Stack.Screen name="Display" component={DisplayScreen} />
      </Stack.Navigator>
    </NavigationContainer>
    </Context1.Provider>


  );
}

export default App;
```

screens/Home.js

```
import { useContext } from 'react';
import { View, Button, Text } from 'react-native';

import { Context1 } from '../App.js';


export default function HomeScreen({ navigation }) {

 const context = useContext(Context1);

 const {
  name,
  age,
  city,
} = context


 return (
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Text>Home Screen</Text>

        <Text>The following initialized data is received from App.js</Text>
        <Text>Name: {name}</Text>
        <Text>Age: {age}</Text>
        <Text>City: {city}</Text>
        <Button
          title="Go to Details Screen"
          onPress={() => { navigation.navigate('Details'); }}
        />


        <Button
          title="Go to Display Screen"
          onPress={() => { navigation.navigate('Display'); }}
        />
      </View>
    );
  }
```

screens/Details.js

```
import {useContext} from 'react';
import { View,  Text, Button, TextInput } from 'react-native';

import { Context1 } from '../App.js';


export default function  DetailsScreen({navigation}) {

  const context = useContext(Context1);

    return (
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Text>Details Screen</Text>
        <Text>Getting new inputs from user</Text>

        <TextInput
        style={{borderWidth:1}}
        placeholder='Enter Name'
        onChangeText={(name) => context.setName(name)}
        />

     <Button
          title="Go to Home Screen"
          onPress={() => { navigation.navigate('Home'); }}
        />


        <Button
```

```
          title="Go to Display Screen"
          onPress={() => { navigation.navigate('Display'); }}
        />

      </View>
    );
  }
```

screens/Display.js

```
import {useContext} from 'react';
import { View,  Text, Button } from 'react-native';
import { Context1 } from '../App.js';

export default function  DisplayScreen({navigation}) {

  const context = useContext(Context1);

  const {
   name,
   age,
   city,
 } = context

    return (
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Text>Display Screen</Text>
        <Text>Name: {name}</Text>
        <Text>Age: {age}</Text>
        <Text>City: {city}</Text>

        <Button
          title="Go to Home Screen"
          onPress={() => { navigation.navigate('Home'); }}
        />


        <Button
          title="Go to Details Screen"
          onPress={() => { navigation.navigate('Details'); }}
        />

      </View>
    );
  }
```