Code.Hub

The first Hub for Developers

Ztoupis Konstantinos

Context API

Code.Learn Program: **React**

Props and State

Data is updated and manipulated

- Props is being passed to the child component from a parent component
- State is being managed within the component itself

Prop Drilling

Prop drilling (also called “threading”) refers to the process you have to go through to get data to parts of the React Component tree

Problems of prop drilling

- Over-forwarding props: Components in between are not interested in all props

- Refactor the shape of some data ({user: {name: 'Pol Lop'}} -> {user: {firstName: 'Pol', lastName: 'Lop'}})

Avoid prop drilling

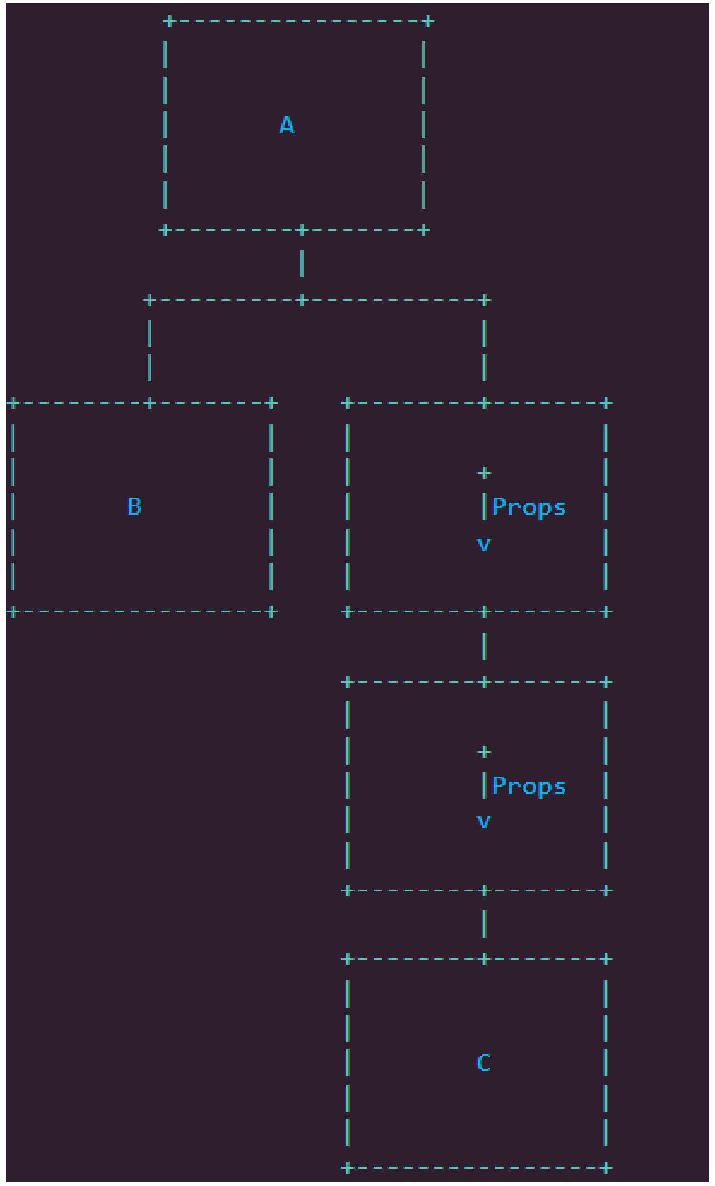
Use big components and break them when is needed

Code.Hub

Context API

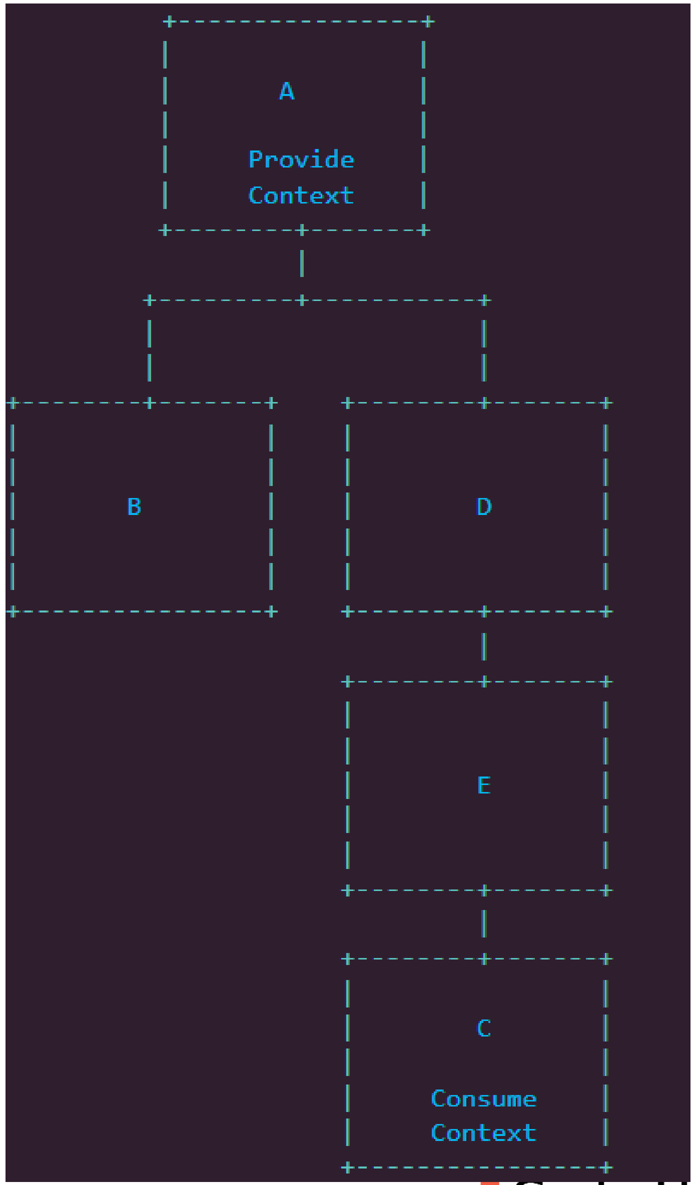
- a powerful feature
- solving issues with prop drilling
- pass props to components that they are really need them
- render a provider anywhere in the app

Prop Drilling vs Context API



prop drilling

context API



Code.Hub

When to Use Context

is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language

Context API

- createContext(): creates context which gives access to a Provider and Consumer component
- Provider: a component provides the context
- Consumer: a component consumes the context

React.createContext

- const MyContext = React.createContext(defaultValue);
- Creates a Context object
- defaultValue: is only used when a component does not have a matching Provider above it in the tree

Context.Provider

Provider component

- used in higher hierarchy of the tree

- accepts a prop called as Value

- acts as a root component in the hierarchical tree such that any child in the tree can access the values that are provided by the context provider

Context.Provider

- <MyContext.Provider value={value}>
- value prop: passed to consuming components that are descendants of this Provider
- Providers can be connected to many consumers
- Providers can be nested to override values deeper within the tree

Context.Consumer

Consumer component:

- consumes the data which is being passed, irregardless of how deeply nested it is located in the component tree
- don’t have to be necessarily be the child of Provider
- can access data from anywhere down the component tree
- <MyContext.Consumer> {value => /* context value */} </MyContext.Consumer>
- requires a function as a child. The function receives the current context value and returns a React node

Context.Consumer

- value passed to the function will be equal to the value prop of the closest Provider for this context above in the tree

- no Provider for this context -> the value will be equal to the defaultValue that was passed to createContext()

useContext

- const value = useContext(MyContext);
- accepts a context object (the value returned from React.createContext) and returns the current context value for that context
- the current context value is determined by the value prop of the nearest <MyContext.Provider> above the calling component in the tree
- is the same as Context.Consumer except that it’s for a functional component
- no wrapping components in a Consumer
- components are simple, easy to read, and easy to test

When should use new Context API?

codebase consists of lot of components that depends on a single piece of data, but are nested deep within the component tree

Final Words

- provides a provider-consumer component pairs to communicate between the nested components in the hierarchy
- an alternative to the state management libraries such as Redux or MobX