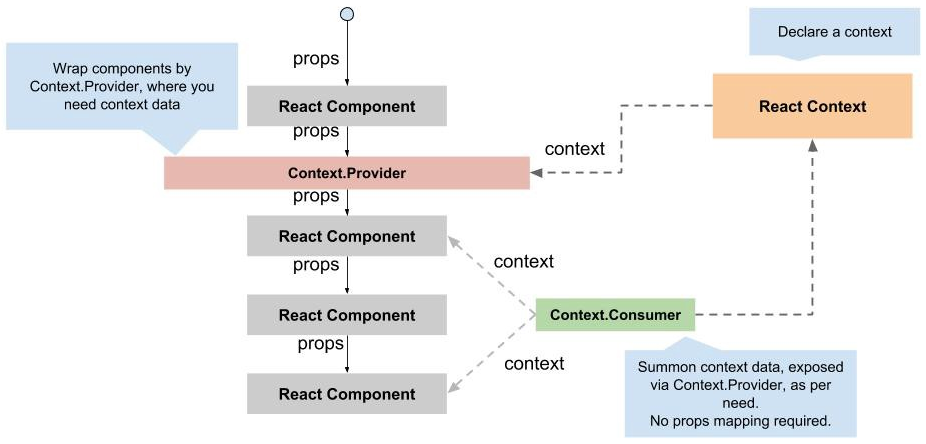
**React State Management – Context API [**<https://reactjs.org/docs/context.html>**]**

So far in our React Programs we maintained the global application state inside the App component that sits on top of the component hierarchy, and passed this state to the components down the hierarchy using **props**. When components down the hierarchy wanted to change the global state stored inside the App component, they invoked call-back functions, which were also passed inside the props. These call-back functions changed the global state in the App component, which then forced a re-render that propagated down the component hierarchy re-rendering all the necessary components. Although this method works, and it is indeed possible to implement all React applications this way, it gets cumbersome especially as the application gets bigger.

To solve this problem, application state managers have been developed. Angular has its own state manager, Vuex and Pinio ARE the state managers for Vue.js. For React, the most popular state managers are the React-Redux (<https://react-redux.js.org/>) and the Context API (<https://reactjs.org/docs/context.html>). React-Redux is somewhat complex and difficult to use. Context API on the other hand is much simpler especially with the newly introduced “useContext” and “useReducer” hooks to function components. That’s why we will be looking only at the Context API in this course. After learning how the Context API works, you may then go ahead and learn Redux yourselves if you will, since the concepts are similar.

**Context API**

Context API provides a way to pass data through the component tree without having to pass props down manually at every level. In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree. The following figure, taken from <https://medium.com/@ipraveen/react-basic-how-react-16-context-api-work-7257591589fc>, illustrates how the Context API works and interacts with the component tree:



As you can see from the figure, we create a context to keep track of some globally-shared app data that sits next to the component tree. To serve this context to the components, we create a **Provider** for this context that wraps any component (called a **Consumer)** that makes use of this context.

**Making use of the Context API**

[The examples given in this section were taken from The Net Ninja’s YouTube channel (<https://www.youtube.com/watch?v=6RhOzQciVwI>).]

Assume that we have two themes that we want all components to make use of: When the theme is light, we want the background to be light-gray and the font-color to be black. When the theme is dark, we want the background to be black and the font-color to be white. In the following two examples, we have a simple Item list app with a Header, an ltemList and a Footer component with each component formatted with light and dark themes.

To see the light theme, look at 09-React-ContextAPI/01-ContextAPI/index1.html

To see the dark theme, look at 09-React-ContextAPI/01-ContextAPI/index2.html

In the previous 2 examples, we achieved the themes through hard-coded CSS. Assume that we want to change the theme dynamically. The way to do this is to store the Theme in the App component, and pass is to all components in the hierarchy using **props** so that they can modify themselves. The following example shows how we can do this:

Look at 09-React-ContextAPI/01-ContextAPI/index3.html

It is now time to separate the Theme state from the App component and put it into a React Context. The way to do this is to first create a Context for the theme as follows:

|  |
| --- |
| const ThemeContext = React.createContext(); |

The second step is to create a Provider for this context. The provider will contain the actual Theme state and any method to manipulate it. To create the Theme Provider, just create a class component, put the data for the Theme inside this provider, and any method to change the data as follows:

|  |
| --- |
| class ThemeContextProvider extends React.Component {  state = {  usingLightTheme: true,  light: {bgColor1: '#eee', bgColor2: '#ccc', color: '#555'},  dark: {bgColor1: '#666', bgColor2: '#333', color: 'white'}  };  toggleTheme = ()=>{this.setState({usingLightTheme: !this.state.usingLightTheme});}  // Returns the JSX for the Theme provider  render(){  return (  <ThemeContext.Provider value={{...this.state, toggleTheme: this.toggleTheme}}>  {this.props.children}  </ThemeContext.Provider>  );  } //end-render  } //end-ThemeContextProvider |

So far we defined the ThemeContext and its provider ThemeContextProvider, but how do we pass this Context to the components? The way to do this is to wrap any component that wants to make use of this Context inside <ThemeContextProvider> class tag. In our example, we want ALL of our components to make use of this Context, so we go our App component and wrap everything inside the ThemeComponentProvider tag as follows:

|  |
| --- |
| <ThemeContextProvider>  <Header/>  <ItemList/>  <Footer/>  </ThemeContextProvider> |

OK. We are now passing the context to all components wrapped inside ThemeContextProvider. But how does a component access this context? There are two ways of doing that:

1. If you have a class component, and that class component is making use of just ONE Context, then there is a very simple way to access that context: Simply define a “static contextType” variable at the beginning of the class, and set it to ThemeContext as follows:

|  |
| --- |
| static contextType = ThemeContext; |

Now, everything inside the ThemeContext can be accessed by “this.context.attributeName”, e.g., “this.context.light”, “this.context.dark”, “this.context.usingLightTheme”. Instead of using “this.context” to refer to each attribute inside the context, it is customary to use the object destructuring operator as follows:

|  |
| --- |
| const {usingLightTheme, light, dark, toggleTheme} = this.context; |

Now, we can access the members of the Context simply by usingLightTheme, light and dark. We can toggle the current theme by calling toggleTheme method, which simply toggles the value of usingLightTheme inside the ThemeContextProvider component, which in turn forces a re-rendering of all components that make use of this Context. Notice that this way of accessing a Context can only be done from a class component, and only if that class component makes use of one single context!

Look at 09-React-ContextAPI/01-ContextAPI/index4.html

1. Another way to access a context from within a class component is to wrap the code that accesses the context inside <ThemeContex.Consumer> tags. Here is how we can re-write our Header class using ThemeContex.Consumer:

|  |
| --- |
| class Header extends React.Component {  render(){  return (  <ThemeContext.Consumer>{(context) =>{  const {usingLightTheme, light, dark, toggleTheme} = context;  const theme = usingLightTheme? light: dark;  const containerStyle = {background: theme.bgColor2, color: theme.color};    return (  <header className="container flex" style={containerStyle}>  <h1 style={{flex: '8'}}>Item List App</h1>  <button style={{flex: '2'}} onClick={toggleTheme.bind(this)}>Toggle Theme</button>  </header>  );  }}  </ThemeContext.Consumer>  );  } //end-render  } //end-Header |

As you can see, ThemeContext.Consumer implements an arrow function whose first parameter is the context. We then use the context within the arrow function and return the JSX. You can look at how the other components have been changed accordingly in the following example:

Look at 09-React-ContextAPI/01-ContextAPI/index5.html

We have now seen the two ways to consume a Context inside the class components. The first method is obviously easier to use, but it allows only one Context to be used/consumed inside the class component. The second method, although cumbersome to use, can also be used inside a function component without any change. Furthermore, it allows multiple contexts to be consumed by a component.

To see how we can consume multiple contexts from within a class component, assume now that in addition to the ThemeContext that stores the current display theme, we have another context that stores the items being manipulated. We can now create a new Context named ItemsContext, put all items in that context, and implement a Provider for it. Notice that the only component that will consume this component is the ItemList component. So we just have to pass this context to the ItemList component.

|  |
| --- |
| <ThemeContextProvider>  <Header/>  <ItemsContextProvider>  <ItemList/>  </ItemsContextProvider>  <Footer/>  </ThemeContextProvider> |

If you want this JSX to be cleaner, you can put all the Providers at the top and make all context accessible from within each component even if they do not use it. It does not matter.

|  |
| --- |
| <ThemeContextProvider>  <ItemsContextProvider>  <Header/>  <ItemList/>  <Footer/>  </ItemsContextProvider>  </ThemeContextProvider> |

Now, to consume both of these contexts within the ListItems component, we create two Consumers and put one of them inside the other as follows:

|  |
| --- |
| class ItemList extends React.Component {  render(){  return (  <ThemeContext.Consumer>{(context) =>{  const {usingLightTheme, light, dark} = context;  const theme = usingLightTheme? light: dark;  const containerStyle = {background: theme.bgColor1, color: theme.color};  return (  <ItemsContext.Consumer>{(context2) => {  const {items} = context2;  return (  <section className='container' style={containerStyle}>  <ul>  {items.map((item)=>{  return <li key={item.id} style={{background: theme.bgColor2}}>{item.title}</li>}  )}  </ul>  </section>  );  }}  </ItemsContext.Consumer>  );  }}  </ThemeContext.Consumer>  );  } //end-render  } //end-ItemList |

You can see that this is a very ugly code, and if you have a third or a fourth context that you may have to consume in this component, the code will get even uglier. To cope with this problem, a useContext hook has recently been introduced to React to make use of one or more contexts in a much cleaner way. But you must be aware that this useContext hook can only be used within function components not within class components. In class components, you have to continue using this old style of Contex.Consumer tags.

Look at 09-React-ContextAPI/01-ContextAPI/index6.html

**useContext Hook**

It is now time to look at the useContext hook. Although it can only be used from within function components, it allows a much cleaner code to be written. We will now re-write the previous example using only function components. So, not only the Context Providers will be function components, but also the consumer components will be function components and will access our two Contexes using the “useContex” hook. Here is the code:

Look at 09-React-ContextAPI/01-ContextAPI/index7.html

When we look at this code, the first thing we see is that a Consumer of a Context needs to just use useContext(ContextName) to make use of any context. useContext returns what is being exported in the ContextProvider’s value={} attribute. In our example, the ThemeContextProvider exports its internal data and one method to change it:

|  |
| --- |
| state = {  usingLightTheme: true,  light: {bgColor1: '#eee', bgColor2: '#ccc', color: '#555'},  dark: {bgColor1: '#666', bgColor2: '#333', color: 'white'}  };  toggleTheme = ()=>{this.setState({usingLightTheme: !this.state.usingLightTheme});}  <ThemeContext.Provider value={{...this.state, toggleTheme: this.toggleTheme}}> |

This means that useContext will return a structure containing the state and the method to toggle the theme. That is, it returns a structure containing the following members.

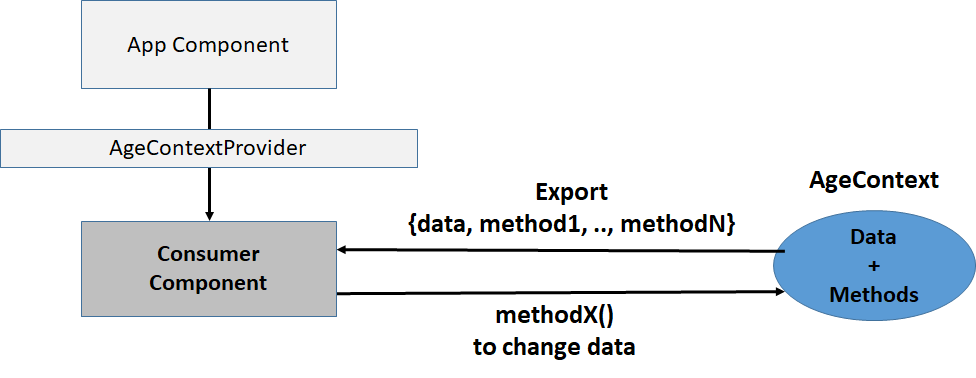
|  |
| --- |
| {usingLightTheme, light, dark, toggleTheme} |

So if we use object de-structuring, we can easily access its members as follows. It is as simple as this to consume a context with useContext hook.

|  |
| --- |
| const {usingLightTheme, light, dark, toggleTheme} = useContext(ThemeContext) |

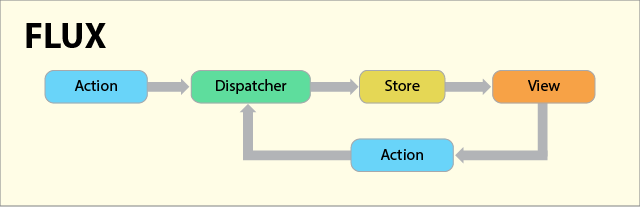
**useReducer Hook & the Flux Pattern**

Consider a Context that stores some data but exports say 5 methods with which the consumers can change this data. Let’s call these methods method1, method2, method3, method4 and method5. With “useContext” hook, we can implement this as follows: Simply create a context and a provider for the context. Implement all change methods inside this provider, and pass both the data and the change methods as “value” to the consumers.



Look at 09-React-ContextAPI/01-ContextAPI/index8.html

Obviously, you can implement any React application with this model of programming. But as the number of call-back methods increases, it becomes cumbersome to pass all call-back methods inside the value parameter. Ideally, we want a single **dispatch** method that can be used to send actions to the Context providers (called data **stores**), which then changes its current state using what is called a **Reducer**. The dispatch method must obviously take two arguments: (1) the **type** of operation being performed, e.g., add, delete, update, …, (2) arguments that this operation takes, e.g., a new item to add, id of the item to delete, etc.. This is called the Flux model, and schematically looks like as follows:



**Actions** are representations of the ways users can interact with the application. They are JS objects, whose only requirement is that they must contain a mandatory **type** field, although they’ll also usually contain some data as well. They look a little something like this:

|  |
| --- |
| {  type: 'ADD\_VALUE',  value: '7'  } |

The **Dispatcher** receives the action and forwards it to each of the application’s stores. A few important things to note here:

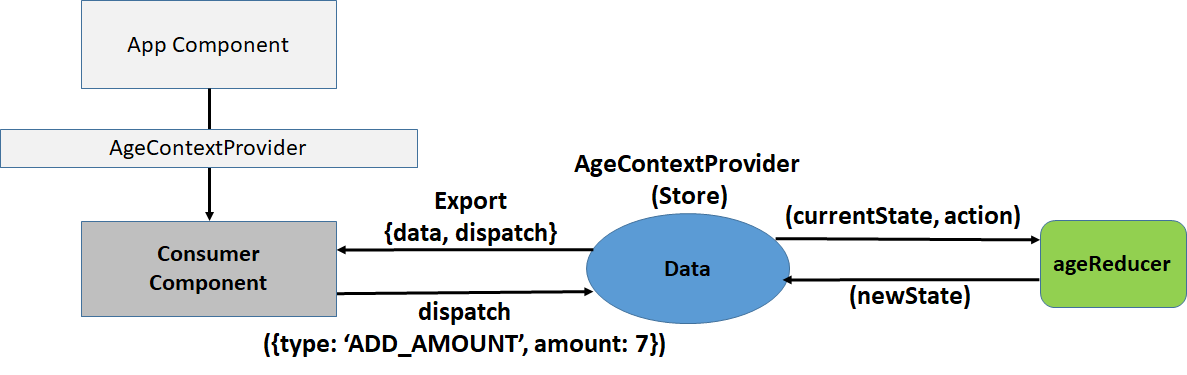
* Any Flux application only ever has one dispatcher.
* ***Every action is sent to every store.***

In other words, you could say that the dispatcher exists to direct traffic, acting as a central interface between actions and stores. That’s all there is to it!

**Stores** are the structures which hold the application’s state, as well as the logic around how to update that state. As such, they are JS objects (usually defined as classes which are subsequently instantiated). An application can have multiple stores, each of which is responsible for a different portion of the application’s state. Stores register with the dispatcher upon creation, and operate according to a couple core rules:

* State must only be changed (or mutated) in response to a received action. As such, stores must not have setters, only getters.
* Each time a store’s data changes, it must emit a change event, which will be broadcast to the application’s views and instruct them to re-render accordingly.

When we implement the previous example based on the Flux pattern using a reducer, i.e., useReducer hook, the new schematic for the application becomes the following:



As you can see, we have one data store (AgeContexProvider), which exports the data for reading and a single dispatch function with which the consumers can take action, i.e., can dispatch actions to the store. When the store receives an action, it sends the current state along with the received action to the reducer. The reducer is nothing but a function that contains the logic to implement the necessary action, which simply creates a new state and returns it. The newly returned state is then committed to the data store. Since the data in the store has now changed, this invokes a re-rendering of all components that make use of this store, i.e., this Context. Here is the code for this new application:

Look at 09-React-ContextAPI/01-ContextAPI/index9.html

You can also find an implementation of our item list app with the Context API in:

Look at 09-React-ContextAPI/02-ItemListAppWithContextAPI1

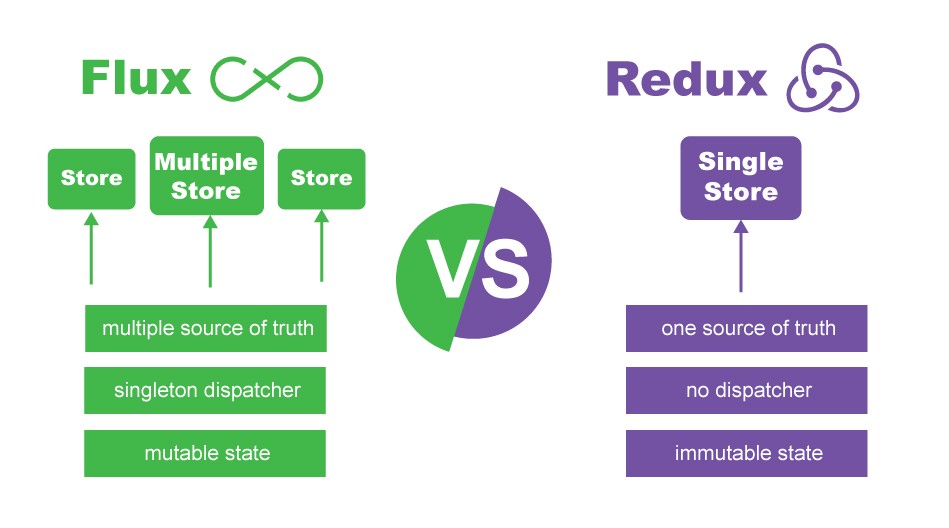
Our ItemListApp using Context API and talking to the json-server. To run this app, first start the json-server with db.json stored in data folder.

Look at 09-React-ContextAPI/03-ItemListAppWithContextAPI2

**Flux vs Redux**

As we mentioned at the beginning of this chapter, Redux is another popular library used for global state management. For a discussion on the differences between Flux & Redux for state management, read the following article: <https://medium.com/@dakota.lillie/flux-vs-redux-a-comparison-bbd5000d5111>. The figure below was taken from this article, and summarizes the main differences between the two models for global state management. Here is another article comparing the two patterns: <https://yourstory.com/mystory/flux-vs-redux>

**Should we use Flux or Redux**? We know that “useReducer” hook is already part of the core React library, and as far as I can tell, should be good enough for all kinds of applications. Redux on the other hand, is a third party library that you must install. Interested students can go to Redux’s Web site (<https://redux.js.org/>) for more information. Since there are a lot of applications on the market that make use of Redux, it might be a good idea to know how it works. Here us a short video describing how Redux works: <https://www.youtube.com/watch?v=_shA5Xwe8_4>. Since the general idea is the same as in Flux, you should not have any difficulty in getting a grasp of Redux in short time.



Just so you can compare the Flux & Redux patterns, here is a re-implementation of the “age” store using react-redux: Look at 09-React-ContextAPI/01-ContextAPI/index10.html

To run this code, you must include both the redux & react-redux libraries into your code as follows:

|  |
| --- |
| <script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.2.0/redux.min.js"></script>  <script src="https://cdnjs.cloudflare.com/ajax/libs/react-redux/8.0.5/react-redux.min.js"></script> |

Similar to the Flux pattern, you create a reducer and create store as follows:

        // Initial state

        const initialState = {age: 50};

        // This is the reducer for reduxStore

        function reduxReducer(state=initialState, action){

          switch (action.type){

              case 'RESET': return {age: 50};

              case 'ADD\_ONE': return {age: state.age+1};

              case 'ADD\_TEN': return {age: state.age+10};

              case 'SUBTRACT\_ONE': return {age: state.age-1};

              case 'SUBTRACT\_TEN': return {age: state.age-10}

              case 'ADD\_AMOUNT': return {age: state.age+parseInt(action.amount)};

              default: return state;

          } //end-switch

        } //end-reduxReducer

        // Get createStore from Redux

        const {createStore} = Redux;

        // Create a Redux store named reduxStore

        const reduxStore = createStore(reduxReducer)

We then import the Provider component from react-redux and wrap this around all components that will make use of this redux store:

        const {Provider} = ReactRedux;

        function App() {

            return (

              <Provider store={reduxStore}>

                <Consumer />

              </Provider>

            );

        } // end-App

Finally, here is how we get the state and the dispatch method from the Redux store:

        // Import two hooks from react-redux

        const {useSelector, useDispatch} = ReactRedux;

        // A simple consumer component

        function Consumer(){

          // Get the age and dispatch function from the Redux store

          const age = useSelector(state=>state.age)

          const dispatch = useDispatch()

The rest of the code is the same as the Flux pattern. You can see that the Flux pattern and the Redux pattern for state management is very similar. Note that Redux recommends using just one store for the application, where you store everything. For further details on Redux, refer to: <https://react-redux.js.org/>

09-React-ContextAPI/01-ContextAPI/index11.html defines actions and dispatches these actions. This is a common practice in redux applications.

Here is another example that has multiple values (age & color) in the store: 09-React-ContextAPI/01-ContextAPI/index12.html

You can also look at <https://redux-toolkit.js.org/>, which is a library that makes it simpler to use Redux in React applications. Here is a Redux-toolkit tutorial: <https://www.youtube.com/watch?v=iBUJVy8phqw>

Here is a video by **Jake Herrington** about all React State Management tools from hooks to Context API to Redux: <https://www.youtube.com/watch?v=-bEzt5ISACA>

Here is a link to his YouTube channel: <https://www.youtube.com/@jherr>