There are many flux implementations available in the market but the redux has been the most popular one in the libraries.

* All the states of the application are stored in an single state which acts as an single store.
* To setup the redux application, we require less boilerplate than any other application.
* All the components are automatically subscribed to the redux store. So, we don’t have to attach create emitter to be subscribed to the dispatchers.
* The redux can also be used in the server-side rendering of the application.
* Redux uses the immutable store, which has a number of benefits with the application
* It also contains the hot reloading feature which helps to instantly see the changes without using the client- state.
* Time-Travel debugging which will help us to go backward and forward to test and debug the application.
* The redux is also very light to handle.

In the flux, we have to write a large amount of boilerplate and redux just handles the all the boilerplate and provide with solution but we don’t get to see the redux in detail working.

In the redux, when an action occurs, the dispatcher is going to notify the store whether there are any actions have occurred.

When the store changes, any components which are listening to the store for the data automatically change. The user interacts with the UI and the unidirectional flow starts all over again.

Think of action as deleting the user,when there is an change in the UI. The dispatcher is going to notify the change to the store and the store handles the state and deletes the specific user. The react just reacts to the UI.

Action:

* Actions could be as creating the user, deleting the user and working with the user.
* They are triggered by the user interactions
* The actions act as an payload for the dispatcher.
* Action creators are also known as the dispatch helper.

The actions can be triggered in two ways and they are when there is an change in the UI and on the page load, on errors during calls to the server.

The payload is similar to an object, it contains an type and data inside it. The names of the object can vary and they don’t have to be the same.



Dispatcher:

* All the data flows through the dispatcher, it acts an like an central hub for the application and there is only one dispatcher for the whole application.
* It holds a list of callbacks
* Stores register with the dispatcher, so they can be notified if the state changes.
* The dispatcher invokes the callbacks that have been registered with it and then broadcasts the payload that is received from the action.
* This delivers the actions to the relevant store.
* Each action updates specific stores based on the callbacks that are registered with it

We are going to save all the actions in an constants file

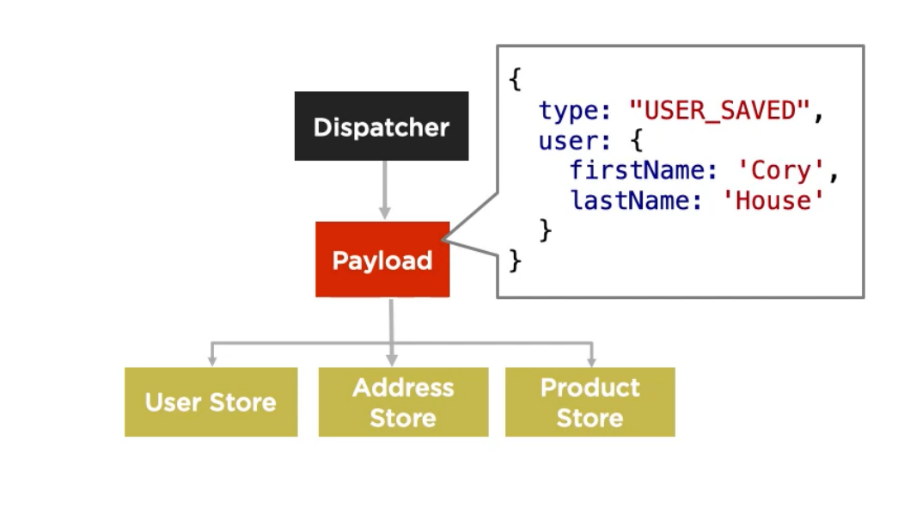
Store:

* The store is where all the data has been saved, the state logic and the data retrieval methods
* The store is not an model but it contains models.
* We can either have an single store or many
* The stores are updated because they have the register callbacks with dispatchers.
* The application should never be registered with the dispatcher. It should only be registered with the store
* The flux uses the node’s event emitter. This allows our store to both listen and broadcast the events to the application
* Only the store can update the data
* When the store is updated, they emit a change event so react can get the data

The structure of a store:

* Extend EventEmitter //So the store can emit the events to the react application.
* We expose methods for adding the data and removing the data(addChangeListener & removeChangeListener)
* EmitChange// This is used for emitting the change.

The dispatcher tells the type of action and based on the action; the store is going to handle the payload. We can also update the stores in an specific order.

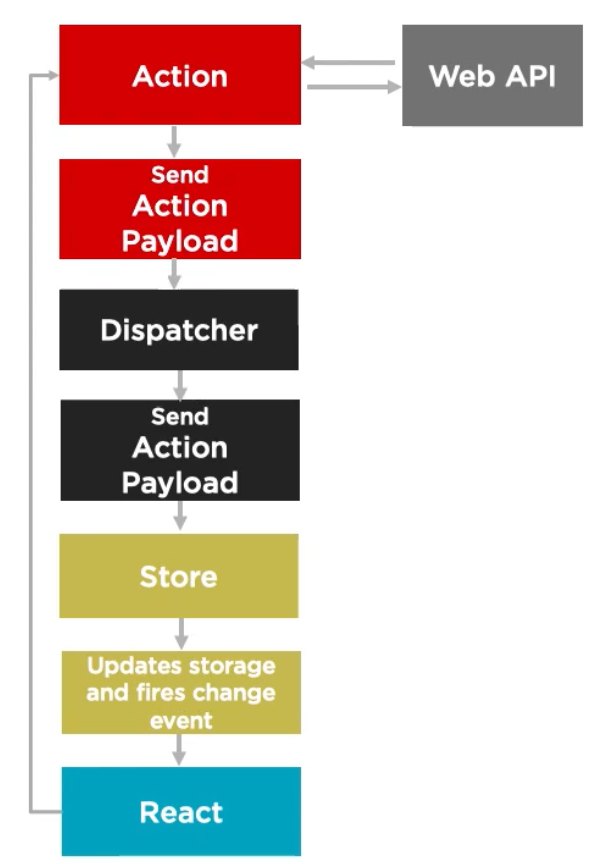


Controller View:

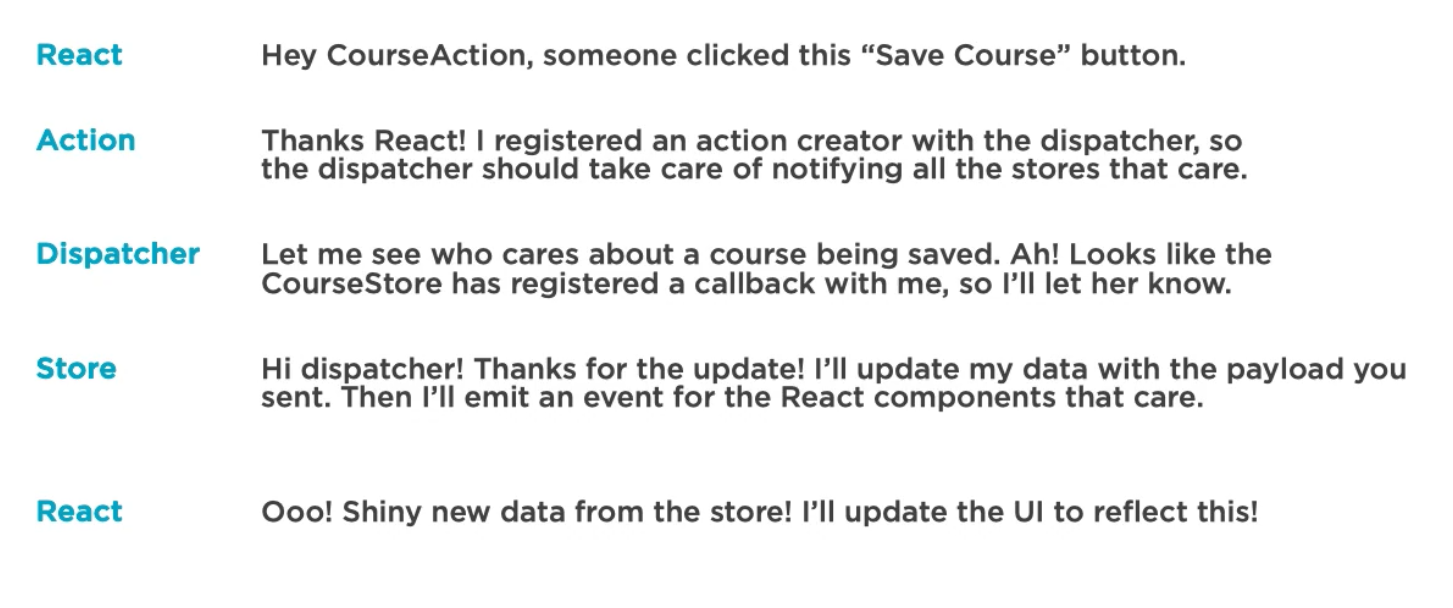
* The controller view is like an root view and it is going to handle the data which will sent to the child views.
* The controller view are the ones who interact with the stores
* When the stores receive an update, the update will be sent to the controller view and it has to send the update to the child view.
* Child components are updated by the help of the props.
* The controller view holds the data in the state and sends the data as the props to the children.

Example:

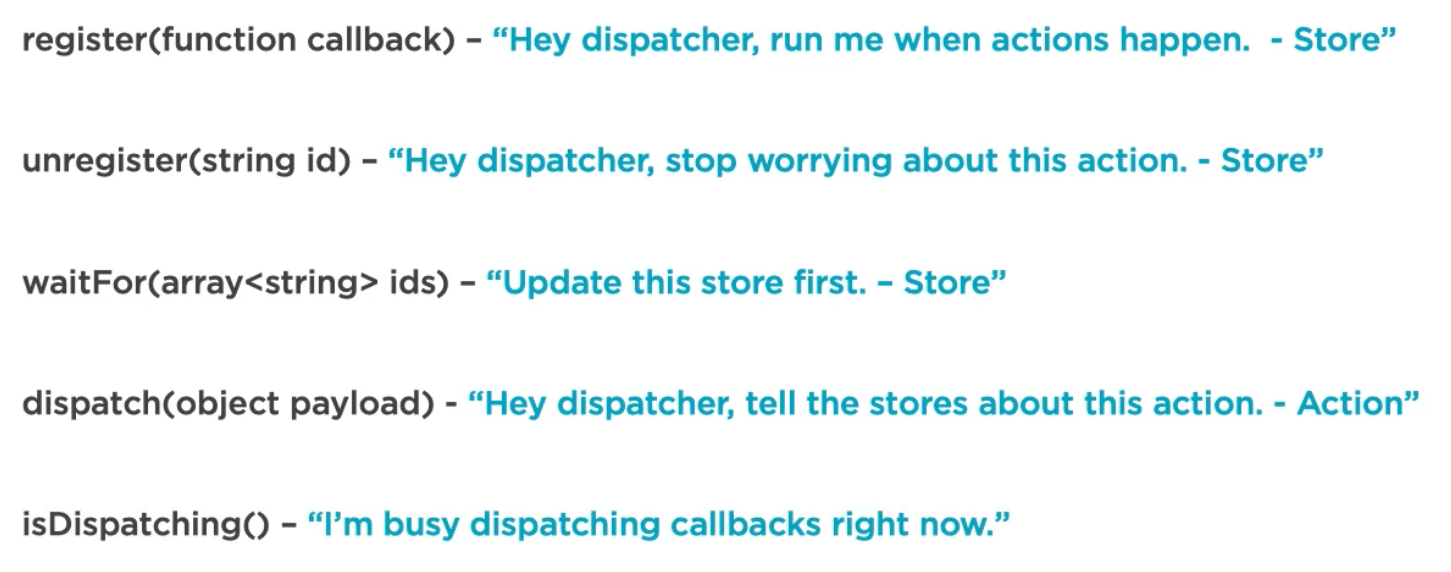
* First, the user is going to save the data which will create an action.
* The action with the type and payload will be sent to the dispatcher
* The actions also talk with the web api to fetch and get the data
* The dispatcher is going to check for any sort of registered call back for the action type
* Based on the action type, the dispatcher is going to send the payload to the specific store.
* The store is going to be updated based on the payload received to it.
* Once the store is updated, It updates the storage and fires the change event
* As the data is changed, the react is going to render itself.
* This process is going to be kept on repeated



A chat with the flux :



How does the flux API work:

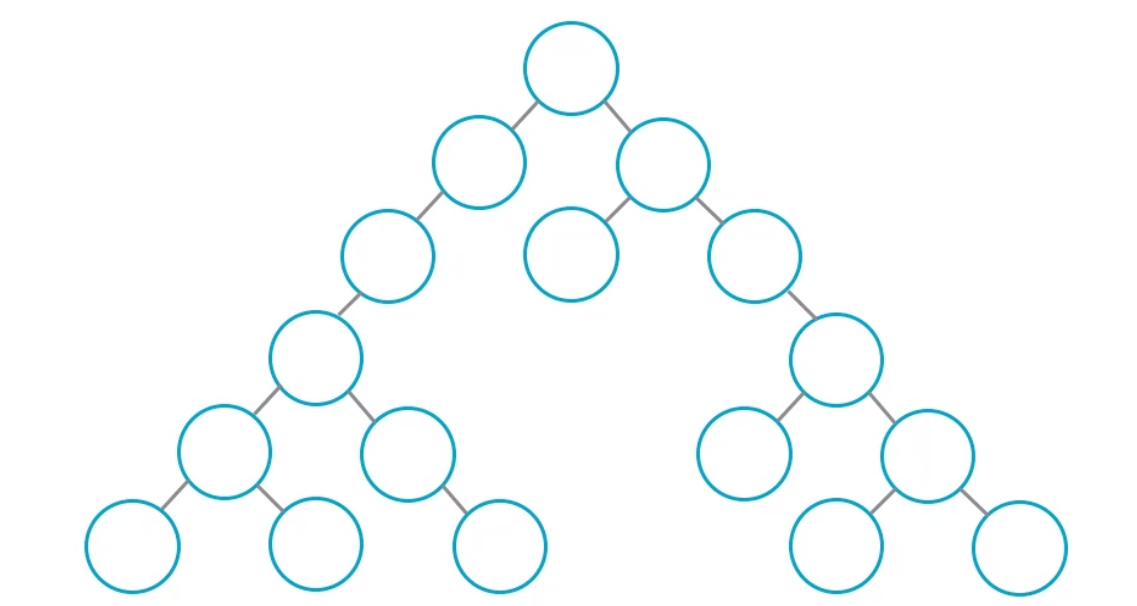


Flux is not an publish-subscribe model:

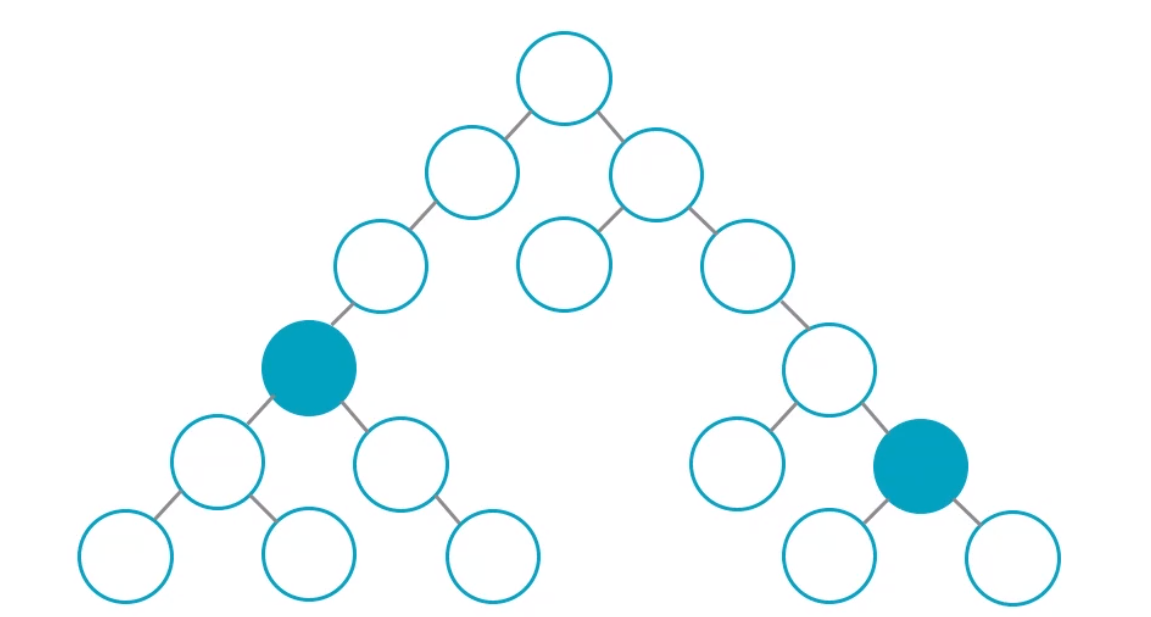
* Every payload is sent to all the registered callbacks
* Callbacks can wait for other callbacks.

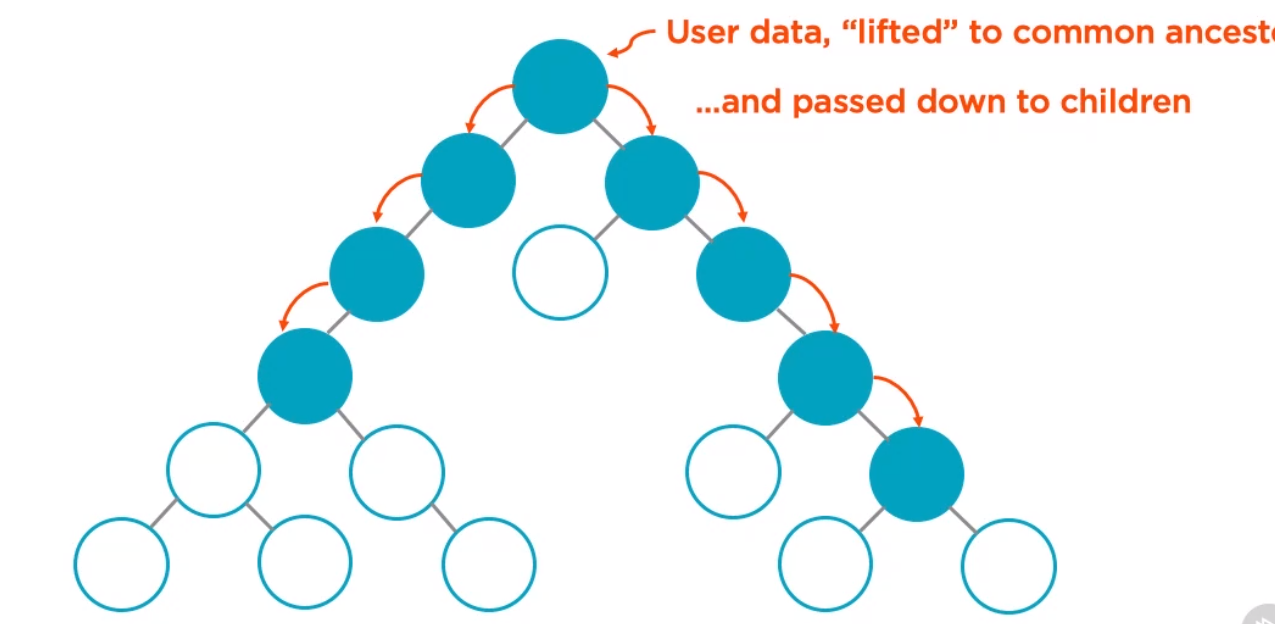
The initial setup for the redux takes some time to setup. It will handle us to solve the components very easily

The below is an representation of how the react works, the root node is going to send the data to the child in the form of the props and it self manages the whole state.

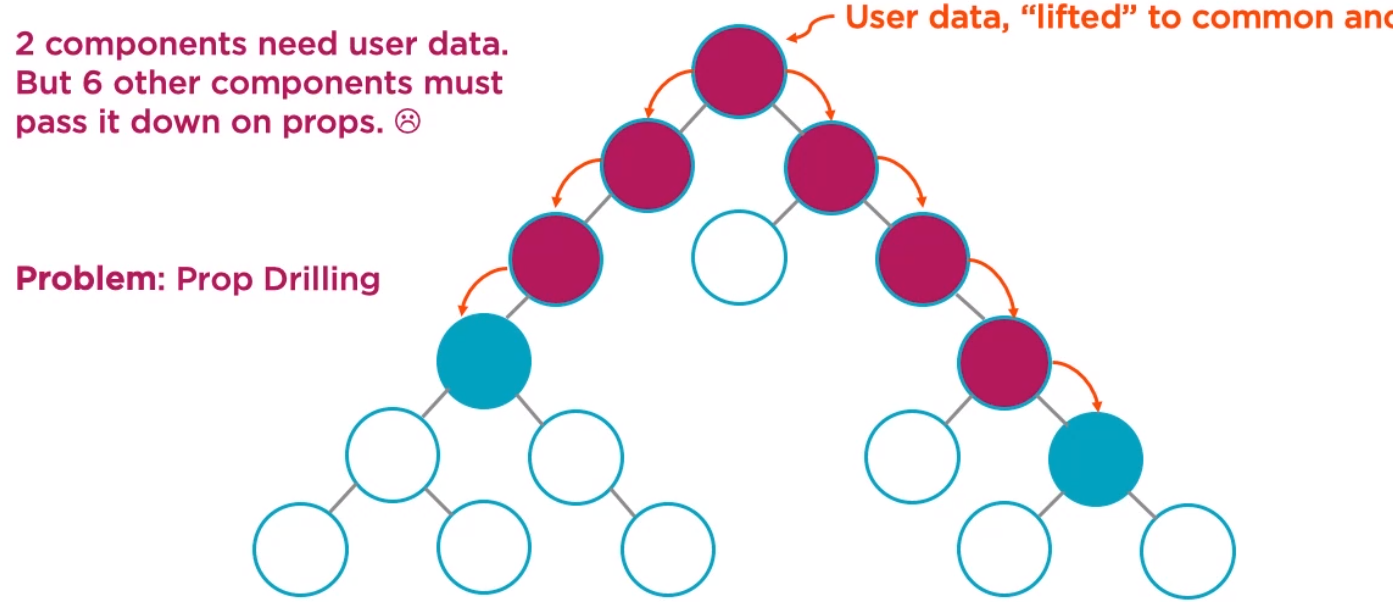


What if there are different components which require the same data, how are we going to handle that part?

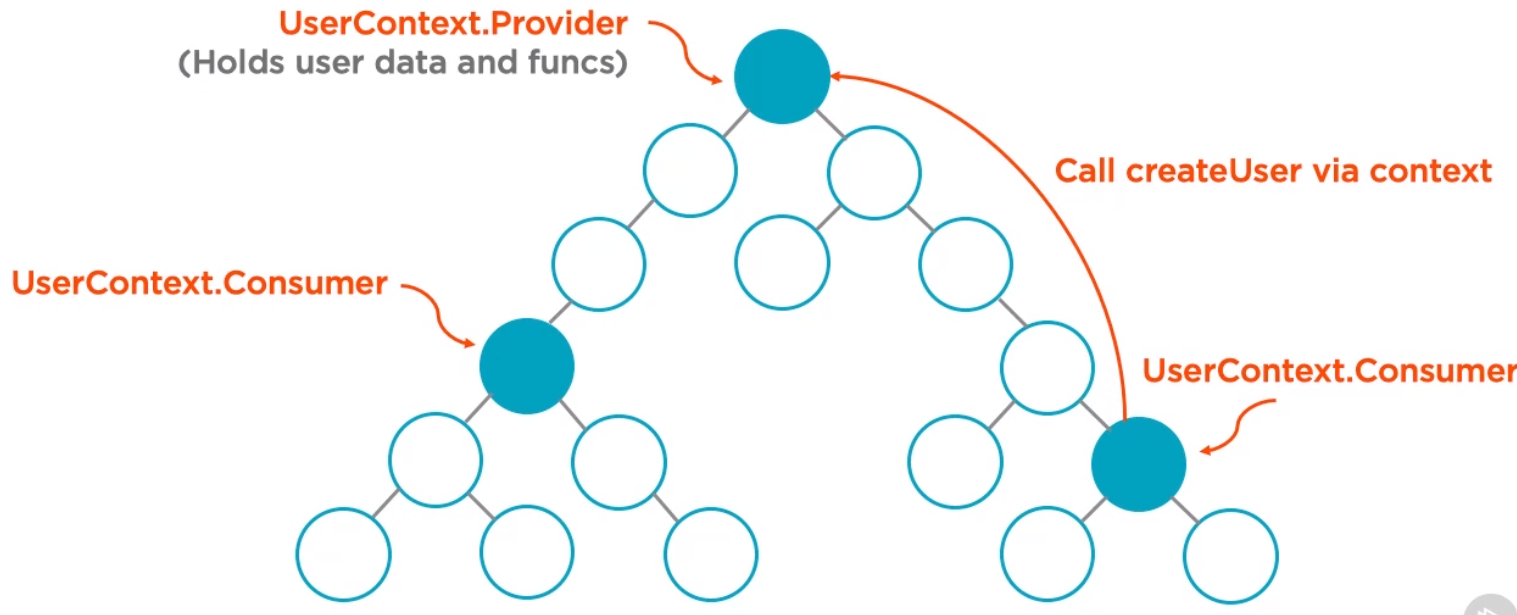


* The first option is to get the data from the root node to the child nodes by the help of the state and props to the children.
* 

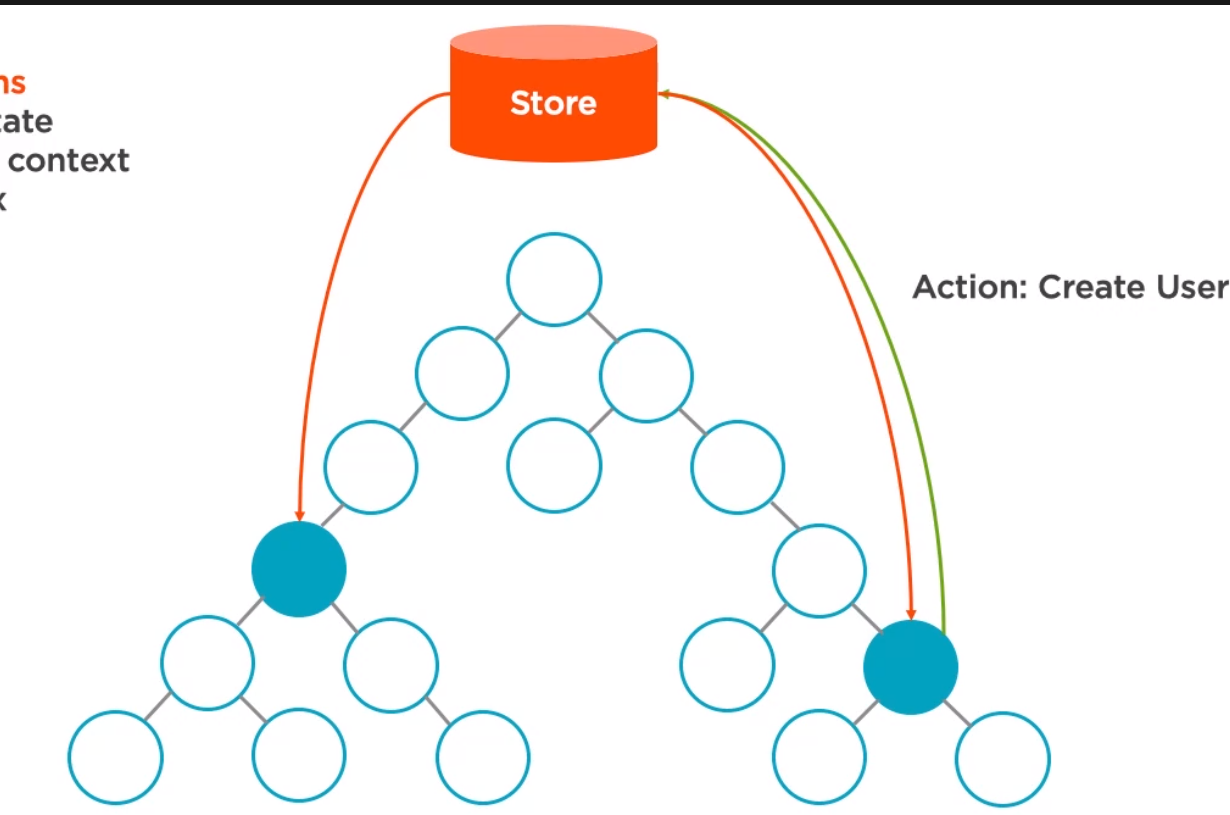
The only components which require the data are two components but we are passing the throughout the other props which are just passing down the props, this problem is known as prop drilling.



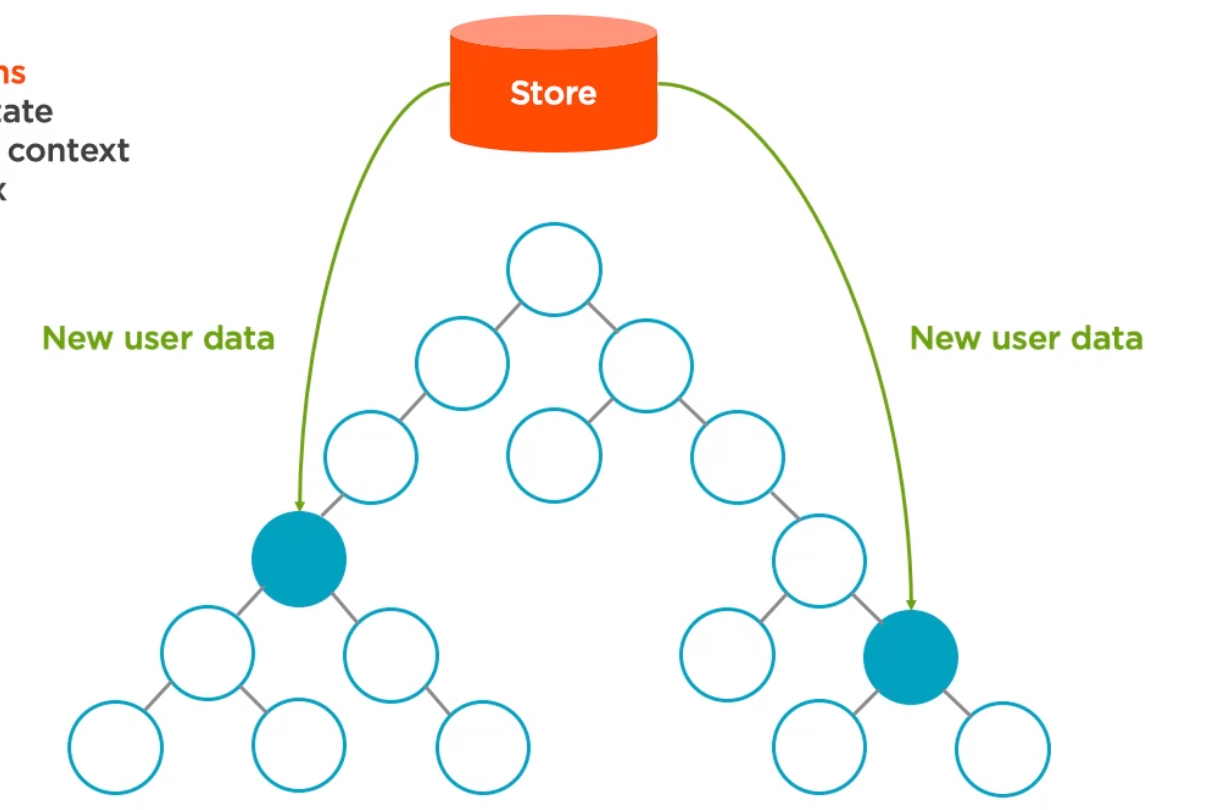
* The other option is by the help of the reacts context. The usercontext.provider holds the user data and the functions and the components consume the usercontext by the help of the consumer.



* The third option is **Redux**.
* In the redux, we are going to have an **centralized store**. The store is similar to an local client side database. In the store, we are going to save the global data and the components can directly access the global data from the store.
* To change the data in the store, we have to dispatch an action to the store.



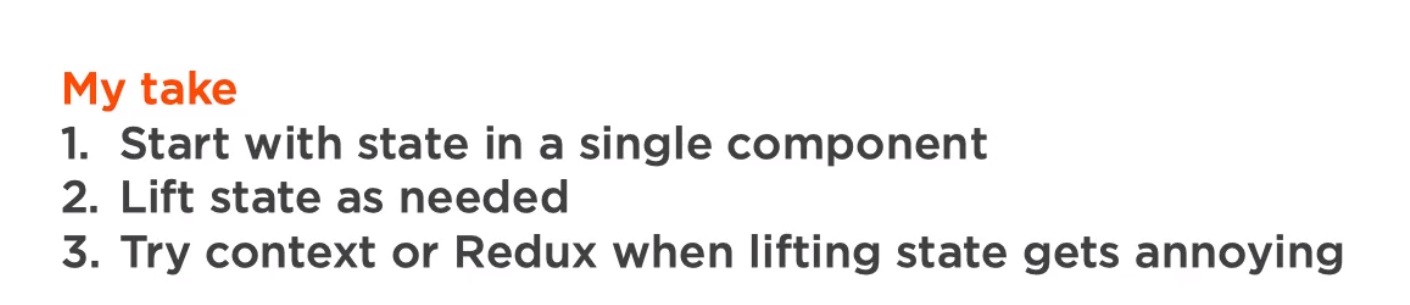
-After the action has been dispactched, the store is going to update the new values in it and send the new data to the components.



Where redux can perform brighter:

* When we complex data flows
* Same data in the different components
* When there is no parent-child relationship data
* When there are many types of actions such as the reads, writes and deletes to the application.

When to use redux:



**Principles of Redux:**

* One Immutable Store //By immutable, the state cant be changed directly, we have to use the dispatcher and action to change it.
* The only way to change the state Is through the action triggers in the application.
* The state changes are handled by the pure functions. These functions are known as the reducers, the reducers accepts a current state in the action and returns a new state to the application

Flux vs Redux

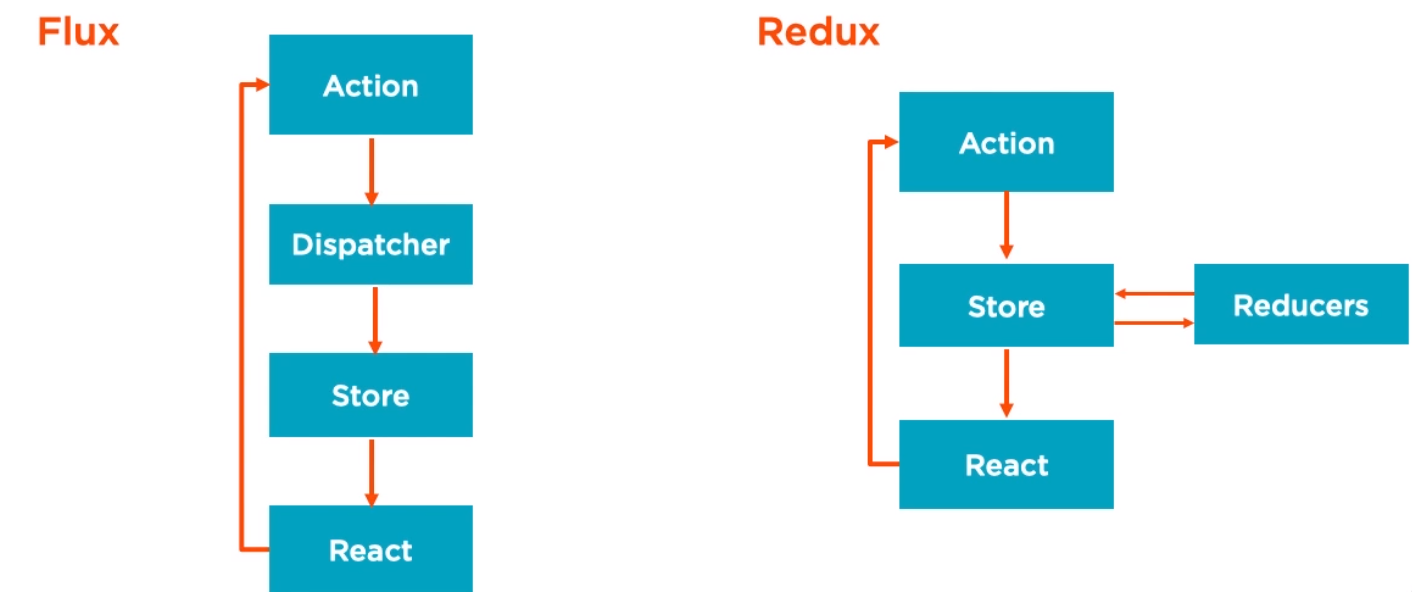
* In both the applications, the data flows down and the actions flow up which is the unidirectional data flow.
* Both have the same actions to change the state
* Both use the store to save the state, but in redux we only have one store whereas in the flux we have multiple stores to save the data.

In redux, we reducers, which take an current state in action and then return a new state.

Containers are just react components but their use is specific. They contain the logic for handling the data and actions, which are passed down to the components as props.

The redux store is immutable

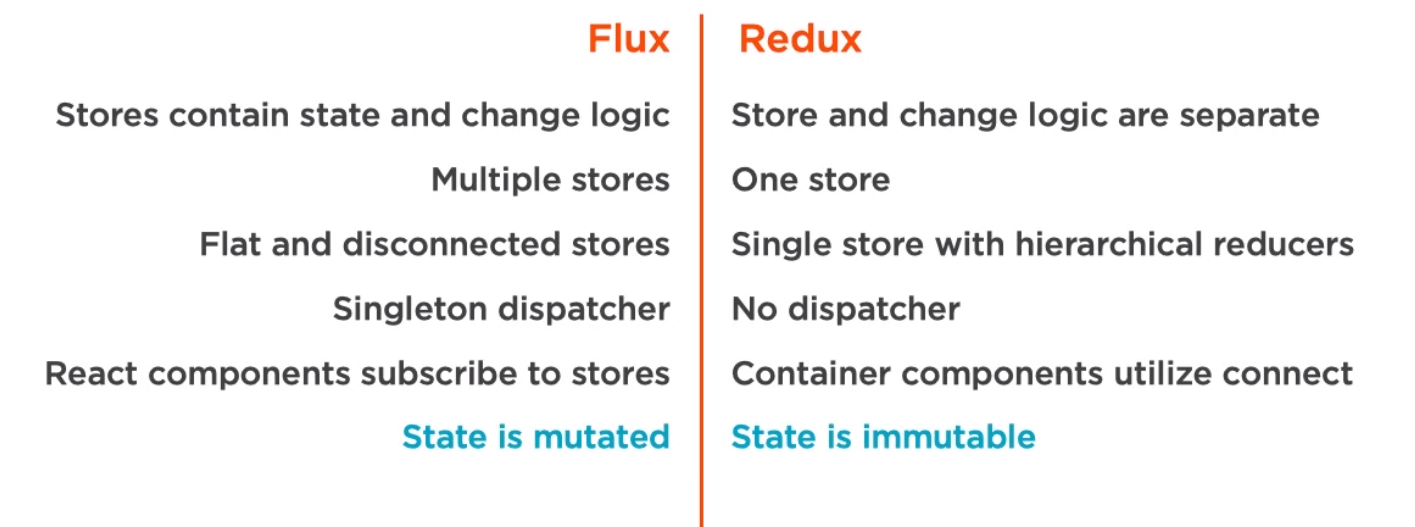
The flow of the flux & redux:



The reducer is responsible for taking the state from the action and then updating to provide a new state.

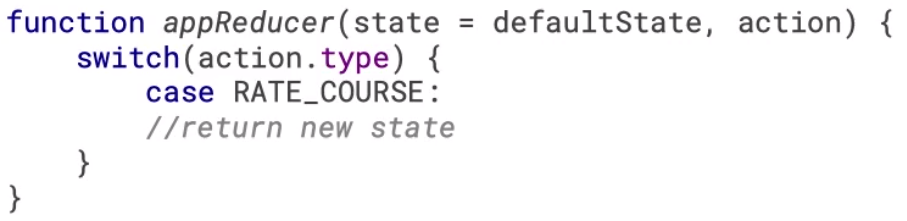
Differences between the flux and redux:

The stores contain the logic for changing the state and storing the state in the flux but the state is stored in the store and the logic for changing the state is stored in the reducers.



Redux Flow:

* We need to have the type of the action and the payload in the redux too
* This action will ultimately be handled by an reducer that returns new state based on the action passed.

rr

The new state is passed to the store and the store will be updated.

Any components which are connected to the redux store will be re-rendered

Building the react – redux development environment:

* Compile JSX
* Transpile JS
* Linting the JS
* Generate index.html
* Reload on save
* Handle everything by the help of one command

The container components vs the presentational components:

* The container components are concerned with the behaviour and data and actions and they mostly don’t focus on the UI
* The containers in the redux can be created by the help of the function redux connect.
* The presentational components are pure dumb and they don’t focus on the data and mainly focus on the presentation
* The container components send the data and actions down to the presentational components
* The presentational receive the data and actions via props to itself.
* The containers know about the redux implementation whereas the presentation components do not know anything about the redux.
* The container components are stateful and they contain some state whereas the presentation components don’t contain any sort of state.