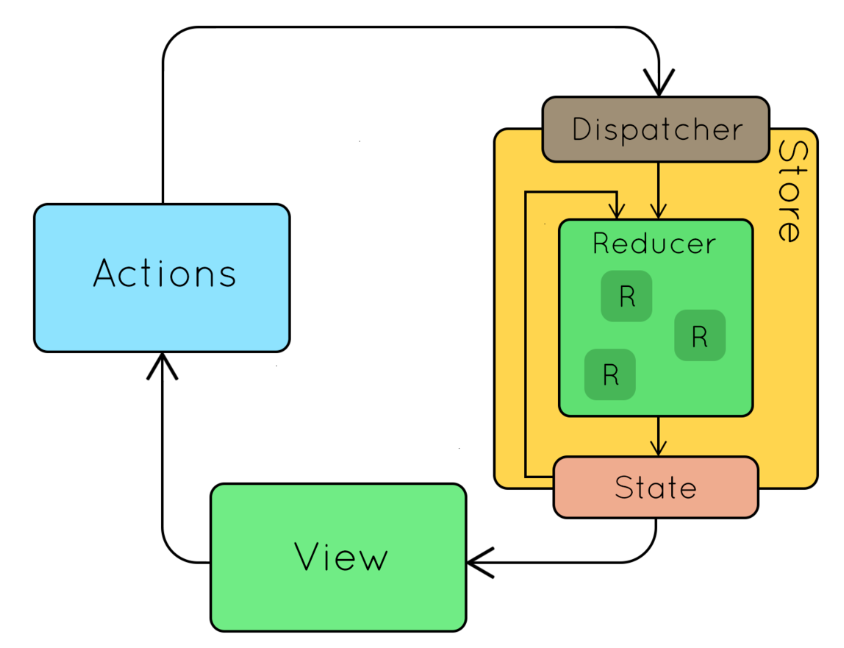
**React 6:**

**Going further with unidirectional data flow:**

In React, we strive for a unidirectional data flow, or to keep our data moving from top to bottom, never bottom to top. This means that our data will all live at the top of the app. For a large app with a lot of global, changing data, this can be a complex process. Redux was created to help with this complexity.   
  
Redux provides a pattern that allow us to keep state in one place and interact with it from anywhere. Anytime the state changes, instead of passing that change down the line of components to the children components who need it, those components will be notified of the change directly, so they can rerender. Similarly, a component communicates necessary changes to state directly to the store, where all the global state management happens.   
  
**A way to imagine Redux**  
In a front-end development conference, two developers from LeadScanr showed the following gif to help people visualize the Redux flow



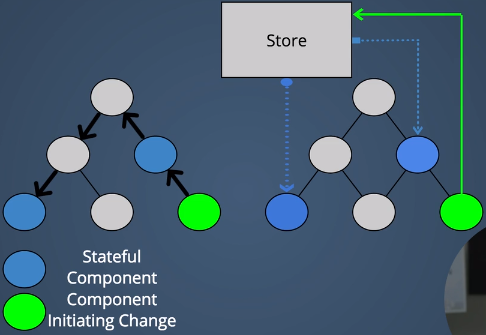
Smart components send actions to the store, which dispatches them to the reducers. The reducers take the old state and the action to form a new state, which will be sent to every component which is subscribed. The components then decide whether or not to re-render.   
  
**Immutability is essential**  
Inside the store, we potentially could be creating a new state every time there is an event. With objects and arrays, it's easy to accidentally give a reference to an already existing object instead of creating a new one. This creates a lot of potential problems the next time we want to make a change. We have to be very careful to create a new state instead of just changing the existing one. A couple code patterns in Javascript make this a lot easier:

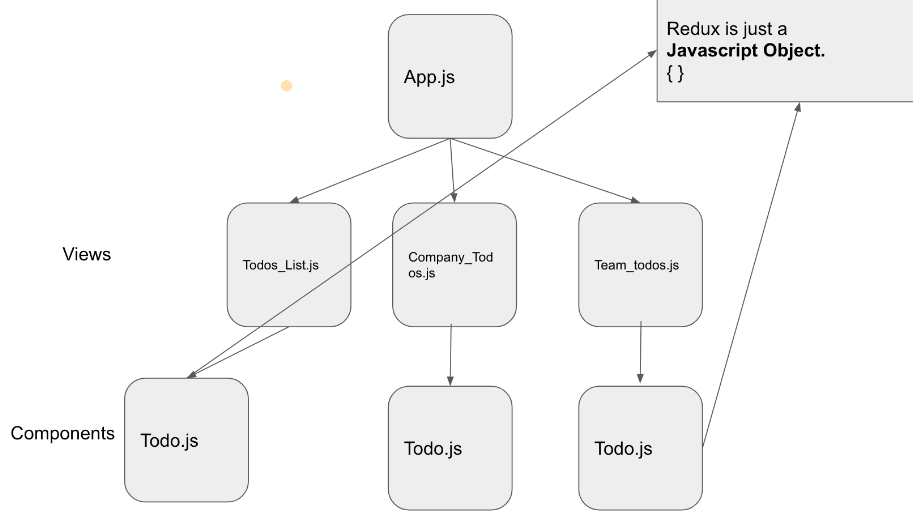
* Use Object.assign( ) and pass in a new object as the first parameter and state as another parameter. This will create a new object with all the properties from any objects that you pass in. e.g. return Object.assign({}, state, {prop: newValue});
* Use the spread operator (...) instead of .push( ). The spread operator takes all the elements from an array. This way, you can make sure all of the elements from the old state will will be there, and it will still be a completely new array. e.g. return [...state, newValue, newValue2];

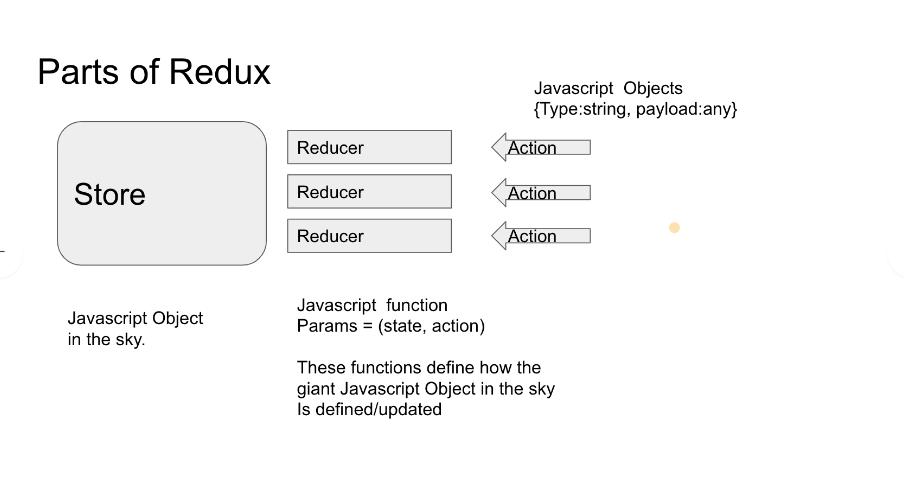
If we have a large React app with many components, the flow of data can get very complex, especially when we need to notify other components of a change of data, or state. Facebook created a solution called Flux, which was modified to create a state container called Redux. In this video, we'll learn about the problems that Redux seeks to solve and why so many React developers use it.

**In the first example below, a component changing state must notify its parent. This parent has state, and it knows there is another component with state that needs to know, but it can not cut across. It must notify its parent, the Root, who must notify its child without state, who must notify its child with state to finally get the message through. This take a very long time, and becomes complicated like a game of Telephone.**

**In the second example we introduce the Store. The child component that needs to update state will no longer update its parent, it is going to notify the Store, who immediately notifies any component that is STORING state, without needing to follow a path to it.**







**Immutable Data in Javascript**

Javascript is pretty flexible when compared to other programming languages out there. Sometimes that's really powerful and sometimes that's really dangerous. When updating state in our React app, it's important to be aware of the dangers of Javascript's flexibility, and to work around them. In this video, we'll talk about how treating our data as immutable can save us a lot of headaches when developing in React.

This works to update the values of properties on an object, but it does not work with immutable data. When you examine the oldUser, you see that the color has also changed to green because they are equal to each other. Objects in Javascript are mutable. This DOES make a copy and updates it, but since the copy is equal to the original, they must always be the same.

Var oldUser = {

Name: “brett”,

Color: “blue”

}

Var newUser = oldUser;

newUser.color = “Green”;

newUser === oldUser; True

THIS WILL CREATE A FRESH MEMORY ALLOCATION:

**Object.Assign()**

**SYNTAX: object.assign({}, object);**

This will allow you to take all the keys and values from oldUser and creates a merged object. You can copy and use new object instead of using or updating both objects.

var newUser = object.assign({}, oldUser)

newUser.color = ‘green’;

newUser === oldUser; False

These are objects that are separate from each other.

YOU MUST BE CAREFUL: An object inside of an object, when updating, can still update the original object. You can copy the reference of a new object from the updated object to the original object. Object.assign can only update values from the first level, no nested values will be updated. (Refer to Immutable JS Library for more help).

To update the new object inside of the new object, without changing the original copy, use this syntax:  
newUser.favorites = {

Food: “broccoli”;

}

Treat your data as immutable while using Redux to avoid problems.

**Redux Reducers**

The reducer is where all of the changes occur. The reducer function that takes the state of the app and figures out what change should occur. TREAT STATE AS IMMUTABLE. You are returning a NEW object with NEW values, but the old object remains the same.

Var state = {

Name: ‘brett’,

Number: 5

}

Var action = {

Type: ‘CHANGE\_NAME’

//The JavaScript action will usually be an object with a key that has a ‘string’ type value

//The string should describe the action, and Reducer will take the action\object to figure out how to return new state

//with the changes specified by your action type.

Payload: ‘Jeremy’

//Action will typically have a payload, which is not necessary, but helpful

}

Function reducer(state, action){

//This calls the old state without any changes made to it yet

//The function takes the state and the action to figure out how to change old state into new state.

Switch(action.type){ //Research Switch Statements in JS for more functionality

Case ‘CHANGE\_NAME’:

Return {

Name: state.number,

Name: action.payload //To update from old to new

}

}

}

Var newState = reducer(state, action);

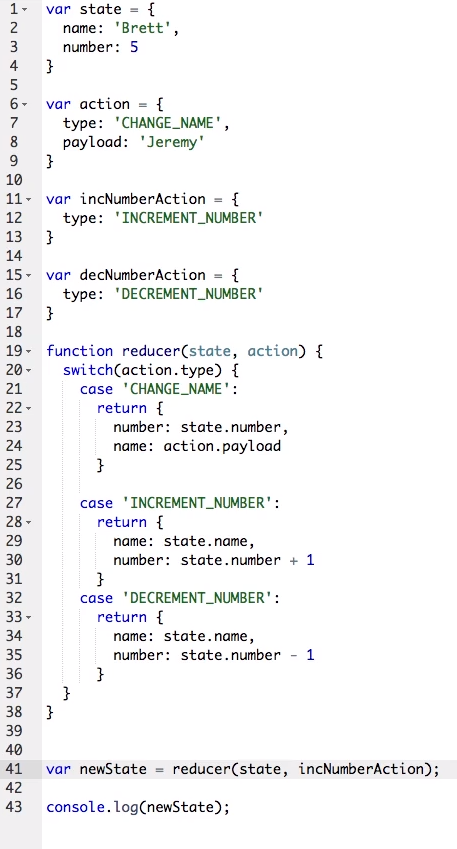
Console.log(newState);

{number: 5,

Name: ‘Jeremy’

}

The action does not know what the whole state of the application is, it just makes the change to its portion and the reducer function decides what to do with the total state, then returns an entirely new state, which is not equal to the original copy of state.



**Create a Store**

In Redux, the store is where we maintain state, handle actions, and dispatch changes. In this video, we'll talk about how to bring Redux into your React app, and how to create and use a store.

**Create a Reducer**

One of the main pieces of Redux is called a reducer. Reducers take in actions and state, and return a new state. This pattern is important, because it leaves full control of state in the store. In this video, we'll learn how to create a reducer and how it works.

