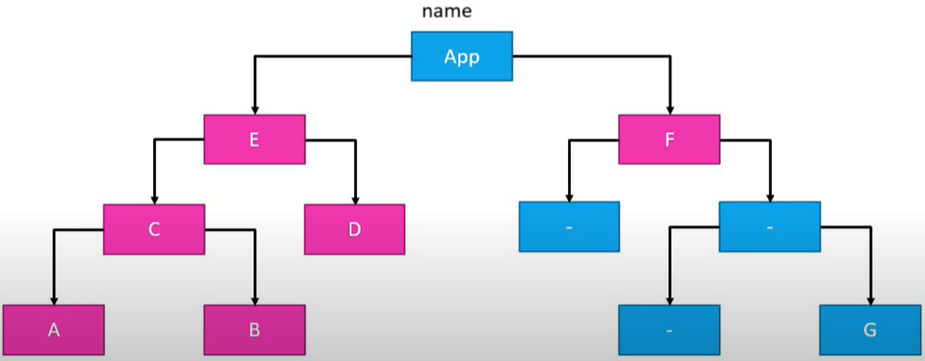
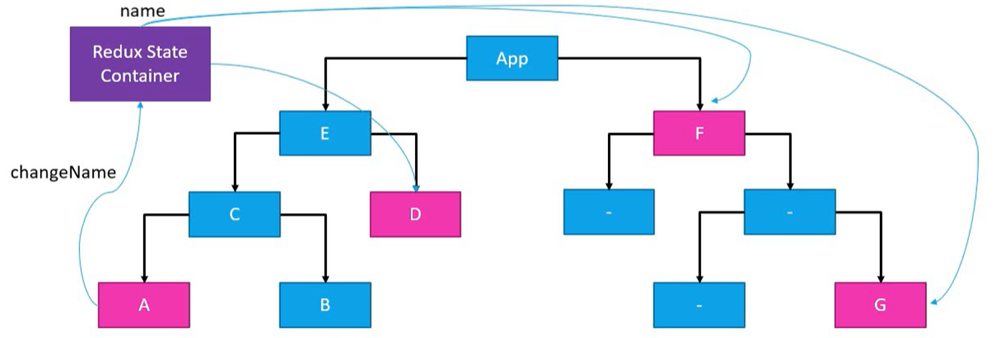
**Redux Notes**

**Introduction**

* Redux is a library that provides a predictable state container for JavaScript apps.
* This means Redux is for all JS applications such as Vue, Angular, Vanilla JS, and not just React
* As a state container, Redux stores and manages the state of your application. The state of your application is the state represented by all the individual components of that app
* Redux is predictable since all state changes are explicit and it is possible to keep track of them
* Why use Redux in React
  + 
  + Suppose we have a state variable that we want to share between component A, B, D, and F called name. We would have to create that state in the App component and pass it down as props. There is a lot of passing and components such as C and E get the state variable even though they don’t need it. Updates to the state also causes many rerenders.
  + 
  + With redux, the state is contained outside the components. If component A wants to update the state, it communicates with the state container which updates the state in a predictable manner. This updated state only passed to the components that are using that state.
* It may seem like react context can help solve this state passing as props, and that we could also use useContext and useReducer. But, Redux 1.0 was released in 2015 when useContext and useReducer were not available. Even after the context hook was released, Redux was still relevant.
* React and Redux are two separate packages that are independent of each other. To directly use Redux in your React application can be difficult, so we use the React-Redux package. React-Redux is the official Redux UI binding library for React. Moreover, this library will provide certain functions that will help us connect our React application with Redux.
* When choosing when to use Redux, we should use it when there are a couple of different routes, a number of components that need to share state. In theory we could use it in an application with just 5 components, but does it add any value? Not really.

**Three Core Concepts**

* Consider the following example:
  + Entities
    - Shop – stores cake on a shelf
    - Shopkeeper – at the front of the store
    - Customer – at the store entrance
  + Activities
    - Buy\_Cake – the customer tells the shopkeeper they want to buy a cake. There must be an interaction between the customer and shopkeeper, the customer cannot jump the counter.
    - Remove\_Cake – upon getting a buy\_cake request from the customer, the shopkeeper will remove a cake from the shelf
    - Receipt – upon removing the cake from the shelf, the shopkeeper will deduct 1 from the count of his cakes to keep track
* Table

  Description automatically generated
* Thus the three core concepts of redux are the following:
  + A store that holds the state of your application
  + An action that describes the changes in the state of the application
  + A reducer which carries out the state transitions depending on the action

**Three Principles**

* First principle:
  + ‘The state of your whole application is stored in an object tree with a single store’
  + This means all the application state should be in a single object which will be managed by the Redux store.
  + Going to the cake example, the cake shop would be the store and it might be represented by the following object: { numberOfCakes : 10 }
* Second principle:
  + ‘The only way to change the state is to emit an action, an object describing what happened’
  + To update the state of the app, we need to let Redux know about that update via an action. We are not allowed to directly update the state object.
  + Going to the cake example, the customer is not allowed to jump the counter and get the cake themselves, they must let the shopkeeper know of the action of buying a cake.
  + An action is an object that has a ‘type’ property and the value of the ‘type’ property is the action. For example, the action object might look like {type: ‘BUY\_CAKE’}
* Third principle:
  + ‘To specify how the state tree is transformed by actions, you write pure reducers’
  + A pure reducer is a pure function that takes in the currentState and incoming action as parameters and returns a newState. Not that since a pure reducer is a pure function, it should not have side-effects such as updating the state, but rather a return value of the newState.
  + Going to the cake example, the reducer is the shopkeeper. When the shopkeeper receives an action, such as buying a cake, he will reduce his cake count by 1.
  + Text

    Description automatically generated
* Three Principle Overview:
  + Diagram

    Description automatically generated
  + We have a JS application (in green)
  + The state of this app is maintained separately in a Redux store (in purple)
  + There is an arrow from the redux store to JS app. This arrow indicates that the JS app is always subscribed to the store. This means any state changes in the redux store will cause the app to change.
  + There is a red arrow from the JS app to the redux store. This arrow indicates that the app cannot directly update the state within the store.
  + There is an action (in blue)
  + There is an arrow from the JS app to the action. If the app wants to update the state, the app has to dispatch an action. Dispatch is the proper terminology.
  + There is the reducer (in orange).
  + There is an arrow from the action to the reducer. This arrow represents that once an action is dispatched, the reducer handles the action.
  + There is an arrow from the reducer to the store. After handling the action, the reducer returns a new state. This arrow represents that the reducer will cause the state inside the redux store to be updated.
  + This updated state in the redux store causes the JS app to update since the JS app is subscribed to any changes to the store

**Redux Project Setup**

* Make sure we have node and npm installed
* Create a project folder (note that the folder name cannot be ‘redux’ since it’ll have a name conflict in the package.json file), and in it, run ‘npm init --yes’ which will initialize a package.json file with the default settings
* We then install redux and add it as a dependency for our project by installing ‘npm install redux’
* We will now create a JS in our project folder to write our code
* Graphical user interface, application

  Description automatically generated

**Actions**

* Recall: We already know that actions are the only way our application can interact with the store. Actions are plain JS objects with a ‘type’ property that indicates the type of action being performed.
* The value of the ‘type’ property is generally defined as string constants for good style and reusability.
* The action object does not need to have only the ‘type’ property. It can have other properties as well.
* In redux, we could implement an action creator. An action creator is a function that returns an action. An action creator is not necessary, but it is good programming style since if we need to change action object (we need to add/remove a property and its value), we only have to change it in one place which makes it every reusable.
* Text

  Description automatically generated
* Notice we defined a constant BUY\_CAKE that has a value of a string that indicates the type of action.
* Notice above, buyCake is an action creator since it is a function that returns an action object which has the ‘type’ property

**Reducers**

* Recall: We already know that reducers respond to actions and causes the state within the store to update. A reducer is a pure function that takes in the currentState and incoming action as parameters and returns a newState.
* Text

  Description automatically generated
* We will create an object that represents the initial state of the application called initialState.
* We have created a reducer function that takes in the current state and incoming action as parameters. Notice that we use function default parameters to set the value of the state object parameter to be the initialState object. This is because we will later learn that when Redux calls the reducer for the first time, the value of state will be undefined.
* Inside the reducer function, we check if the ‘type’ property of the action has a value equal to BUY\_CAKE. If it does, it creates a copy of the state via the spread operator and decrements the ‘numCakes’ property by 1. Notice this is a pure function since we did not directly change the state parameter object, we created a copy via the spread operator to create a reference to a new object.
* If the ‘type’ property of the action does not have a value equal to BUY\_CAKE, it will return the state parameter object. Note that the reducer is still a pure function since even though we are returning the same reference as the same state object, we did not change the state object.

**Store**

* There is only one store for the entire application
* The store is responsible for:
  + Holding all the state of an application
  + Provides a method called getState() which gives the application to the state the store holds
  + Provides a method called dispatch() to allow updates to the state. This dispatch function accepts actions as its parameter
  + Registering listeners via the subscribe method. The subscribe function takes in a function as its parameter which gets executed anytime an action is dispatched (note that the action does not have state). The return value of the subscribe function is a function that allows us to unregister listeners.
* Text

  Description automatically generated
* Output: Text

  Description automatically generated
* In line 1, we must import the redux library. Since we are running this application as a node.js application, we will use the require syntax which will allow us to import libraries.
* The redux library provides a function called createStore. In line 2, we store the redux.createStore function inside the createStore function. This just makes using the createStore function easier since instead of saying redux.createStrore, we can just say createStore.
* In line 29, we create a store by executing the createStore function that we initialized in line 2. This createStore function takes in a reducer as it’s parameter. The reducer function allows the store to update state based on received actions. Note that the reducer function has the initial state of the application (since we set it using function default parameters). This createStore function returns a store which we can store in a constant (we call this constant whenever we want to use the store).
* In line 30, we make use of the getState function. When executed, this function returns the state object stored within the redux store. Since we haven’t changed the redux store yet, the state stored inside the redux store is the initial state. That is why ‘Initial State { numCakes: 10 }’ is logged.
* In line 31, we make use of the subscribe function. This subscribe function takes in a function as its parameter which gets executed anytime an action is dispatched. Note that this function still gets executed even if the action dispatched does not cause the state stored in the redux store to change.
* In this example, it prints the updated state after the redux store change using the getState function. Note that the return value of the subscribe function is a function that allows us to unregister listeners. In this case, we store this returned function inside the unsubscribe function constant
* In line 34, we make use of the dispatch function. This function takes in an action as its parameter. We could directly pass in an object with a ‘type’ parameter, but we already have a buyCake() action creator function which returns an action object so we will use that instead. In our example, the buyCake function returns the { type: BUY\_CAKE, info: "first redux action" } action object which gets passed to the dispatch method. This dispatch method causes the reducer to handle this incoming action object. If the value of the ‘type’ property of the action object is equal to BUY\_CAKE, we return a copy of the new state object with the ‘numCakes’ property decrement by 1. In our example, the action object’s value of the ‘type’ property is indeed equal to BUY\_CAKE, so we return a copy of the new state object with the ‘numCakes’ property decrement by 1. Thus, the state object returned is {numCakes: 9} since the initial state object was { numCakes: 10 } and we decrement the ‘numCakes’ property by 1. Since we dispatched an action, the parameter function of the subscribe method gets executed. This paramter function logs out the updated state which is why ‘Updated State { numCakes: 9 }’ is logged. A similar thing happens for lines 35 and 36.
* In line 37, we execute the unsubscribe method since we no longer want our application to listen to changes in the redux store.

**Multiple Reducers and Combining Reducers**

* Going back to the cake example, suppose we also want to sell ice cream. We can define an action called BUY\_ICE\_CREAM, create a new buyIceCream action creator, modify the initialState to also include numIceCreams, and modify the reducer to manage the BUY\_ICE\_CREAM action. The code for the modification is shown below.
* Text

  Description automatically generated
* Output: Text

  Description automatically generated
* While this approach works, this approach is not scalable as the reducer function gets very complicated. If we wanted to also sell cookies, burgers, etc, the reducer function would get extremely complicated and hard to debug.
* Rather, a better solution is to split up the state and have two reducers. The state will be split into ice creams and cakes. One reducer only handles actions regarding ice creams, and the other only handles actions regarding cakes.
* Text

  Description automatically generated
* However, the createStore function can only take 1 parameter, hence only 1 reducer. We have two reducers, namely cakeReducer and iceCreamReducer. To resolve this problem, we will combine the two reducers into 1 reducer via Redux provided function called combineReducers. combineReducers allow us to combine multiple reducers into a single reducer which can then be passed to the create store method. We can store the redux.combineReducers function inside a constant combineReducers function. This just makes using the combineReducers function easier since instead of saying redux.combineReducers, we can just say combineReducers.
* The combineReducers function accepts an object. Each key/value pair in this object corresponds to a reducer. Note that the keys can be named whatever, but conventionally, the key name is the name of the state the reducer handles. This combineReducers function returns a reducer which is conventionally called rootReducer. rootReducer is what we will pass into createStore
* Text

  Description automatically generated with medium confidence
* Thus, the final code with separate reducers and the use of the combineReducer method is shown below:
* Text

  Description automatically generated
* Output: Text

  Description automatically generated
* There is a slight change as the state object has changed. Before, the initial state was {numCakes: 10, numIceCreams: 5}. Now, it is { cake: { numCakes: 10 }, iceCream: { numIceCreams: 5 } }. Note the key names are ‘cake’ and ‘iceCream’ because those were the key names used in the object that was passed into the combineReducer function. Thus, if we want to access the number of cakes, we use store.getState().cake.numCakes instead of store.getState().numCakes.
* Also note that when an action is dispatched, both the cakeReducer and iceCreamReducer act on the action. The reducer that is executed first is which ever reducer comes first in the object that was passed to the combineReducer function. The reducer that is executed second is which ever reducer comes second in the object that was passed to the combineReducer function. As well, each reducer manages only its part of the global state. For demonstration purposes, suppose we had the following reducers and combineReducer. The following is bad practice since reducers are supposed to be pure functions with no side effects:
* Text

  Description automatically generated
* For example, suppose we did store.dispatch(buyCake()) which would dispatch a BUY\_CAKE action.
* iceCreamReducer would act on this action before cakeReducer since iceCreamReducer comes before cakeReducer in the object passed to the combineReducer function. When iceCreamReducer acts on the action, it manages only its part of the global state. Thus, it only affects iceCreamState and not cakeState nor the state object stored in the redux store. This iceCreamReducer logs ‘ice cream reducer’ followed by the iceCreamState ‘{ numIceCreams: 5 }’. Since the action type is not BUY\_ICE\_CREAM, the iceCreamReducer just returns the current state.
* Then, cakeReducer would act on this action. When cakeReducer acts on the action, it manages only its part of the global state. Thus, it only affects cakeState and not iceCreamState nor the state object stored in the redux store. This cakeReducer logs ‘cake reducer’ followed by the cakeState ‘{ numCakes: 10 }’. Since the action type is BUY\_CAKE, the cakeReducer just returns an updated state with the numCakes decremented by 1.
* Since we set up a subscription to the redux store, it now logs the updated state which is ‘Updated State { iceCream: { numIceCreams: 5 }, cake: { numCakes: 9 } }’
* When our application grows in size, we can split the reducers into separate files and keep them independent of each other.