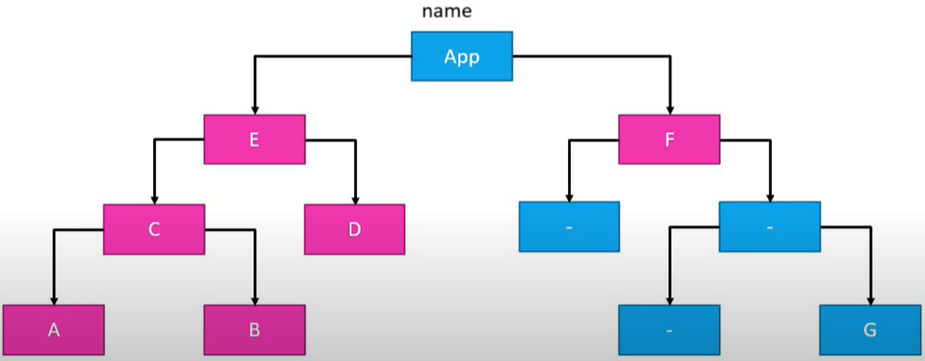
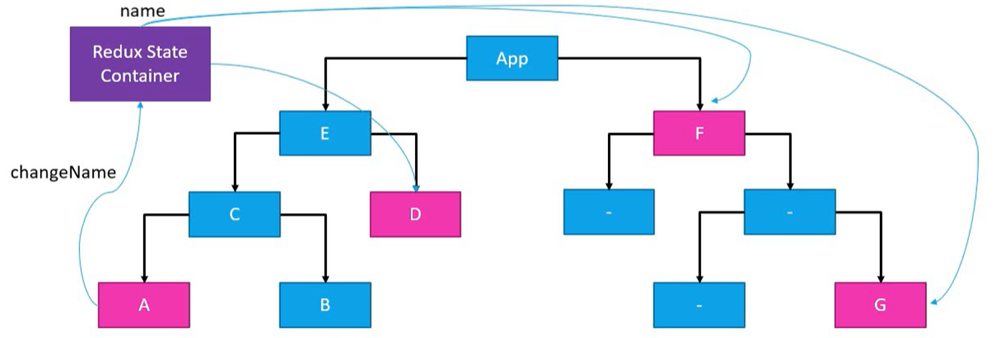
**Redux Notes**

**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.

**Middleware**

* Redux Middleware allows you to intercept every dispatched the moment before the action reaches the reducer so you can make changes to the action or cancel the action.
* Use middleware for logging, crash reporting, performing async tasks, etc
* Ex: we will explore the redux-logger middleware
  + This library allows u to log all the information related to redux in your application
  + Run in project cmd: ‘npm install redux-logger’
  + To use this package, follow this documentation: <https://www.npmjs.com/package/redux-logger>
  + First, we need to require this package and store it in a constant called reduxLogger
  + Then we create an instance of the logger by executing the createLogger function that is a property of reduxLogger
  + 
* How to include middleware
  + The redux library provides a function called applyMiddleware that allows you to apply middleware
  + 
  + Notice in the code above, the right hand side is a function callback and not a function execution. Thus, the applyMiddleware constant is a function
  + In the createStore function, we can not only pass in a reducer parameter, but also a middleware parameter. To do so, we pass in applyMiddleware(nameOfMiddleware).
  + In our reduxLogger example, we do the following:
  + 
  + Note that we only passed in 1 middleware, which is the logger middleware. We can actually pass in as many middleware as we want.
  + Thus if we have the following code (note there is no subscribe function), we get the following output:
  + Text

    Description automatically generated
  + Output: Text

    Description automatically generated
  + TLDR: to use a middleware, import applyMiddleware from redux, pass applyMiddleware as an argument to the createStore function, and pass in the middleware to the applyMiddleware method.

**Async Actions/ Redux Thunk Middleware**

* As of now, all of our actions are synchronous such as dispatching BUY\_CAKE which immediately decrements the numOfCakes by 1.
* Suppose we want to asynchronously fetch data from an API and store that data in the redux store. This would require an asynchronous action.
* Ex: suppose we want to fetch a list of users from an API end point and store it in the redux store
* We would have to consider the state, action, and reducers for this application
* When fetching data, the state of our application is usually object with a ‘loading’, ‘data’, and ‘error’, property.
  + A picture containing text

    Description automatically generated
  + The loading property is true if we are in the process of fetching the data and still waiting for a response from an API.
  + The data property is an array of all the users. Initially, the array will be empty since we have yet to get a response from the API.
  + The error property is a string that represents the error sent back from the API if there is one. Initially, the error message will be empty since we have yet to get a response from an API so we don’t even know if there is an error.
* The actions we would be the following:
  + FETCH\_USER\_REQUEST: this action fetches a list of users
  + FETCH\_USER\_SUCCESS: this action can only be dispatched if the FETCH\_USER\_REQUEST action successfully fetched the data.
  + FETCH\_USER\_FAILURE: this action can only be dispatched if the FETCH\_USER\_REQUEST action failed to fetch the data.
* The reducer function would be the following pseudocode:
  + Text

    Description automatically generated with medium confidence
* The code would look like the following:
* Text

  Description automatically generated
* Note that the above code is incomplete since we did not make an asynchronous api call yet.
* Instead of using fetch for our api requests, we will use axios which we can import by running in the project cmd : ‘npm install axios’
* Require axios as shown below
* 
* We will also need to import the redux-thunk middleware package. This is a package from the redux ecosystem and is the standard way to define async action creators.
* To import this package, run in the project cmd: ‘npm install redux-thunk’
* Require this package as shown below
* 
* Create an applyMiddleware function from redux and pass it as a parameter to the createStore function with the thunkMiddleware being the parameter to the applyMiddleware function
* 
* Now, we will create an async action creator which will the following:
* Text

  Description automatically generated
* We define a function called fetchUsers which will be our action creator.
* We learned that action creators are supposed to return actions. However, the thunk middleware allows an action creator to return a function instead of an action object.
* Thus, the return value of the fetchUsers function is a function.
* This return function does not have to be pure, so it can have side effects such as async api calls and dispatching actions.
* To allow this return function to dispatch actions, we pass in the dispatch method as its argument.
* Before we fetch the data, we create an action using the fetchUserRequest action creator and dispatch this action to change the state in the redux store to { loading: true, users: [], error: '' }. This new state gets logged because of the subscription to the store.
* Then, we actually send the api request using axios by saying ‘axios.get(‘url’)’ which returns a promise.
* If the promise is resolved, the resolve value is an object with a ‘data’ property. we can tag a .then method which takes the response’s data and cleanining it up to only include user id’s by saying ‘response.data.map((user) => user.id)). We then pass this array of user ids to the fetchUsersRequest action creator which returns an action that gets dispatched which changes the state in the redux store to { loading: false, users: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], error: ‘’}. This new state gets logged because of the subscription to the store.
* If the promise is rejected, the reject value is an error. we can tag a .catch method which takes this error as a parameter and passes the error’s ‘message’ property as a parameter to the fetchUsersFailure action creator which returns an action that gets dispatched which changes the state in the redux store to { loading: false, users: [], error: ‘error message’}. This new state gets logged because of the subscription to the store.
* Now, we can execute the fetchUser action creator inside dispatch function which executes the return function of fetchUsers (possible because of thunk middleware):
* Graphical user interface, text

  Description automatically generated
* Thus, the entire code looks like the following:
* Text

  Description automatically generated
* Text

  Description automatically generated
* Text

  Description automatically generated
* Now, if we run the file in our terminal, the output would be:
* Text

  Description automatically generated
* Note that we don’t have an unsubscribe function for the store.
* If we try the following code, we get the following output:
* Text

  Description automatically generated
* 
* Note that the redux store’s state is updated with the new users, but the application does not know about the updated state since it unsubscribed before the redux store’s state got updated.
* To explain why the updated state after the fetching occurs does not get logged, recall the event loop.
* In line 75, we create a subscription to the redux store which executes the parameter function upon every action dispatched.
* In line 78, we execute the fetchUsers action creator. In the fetchUsers return function, we dispatch the FETCH\_USERS\_REQUEST via the fetchUsersRequest action creator. Since we dispatched an action, the store subscription function gets executed and so we log : { loading: true, users: [], error: '' }
* We then make a fetch request by saying axios.get("https://jsonplaceholder.typicode.com/users") which by itself is synchronous just like fetch. It consuming the promise via .then or .catch that is asynchronous since we have to wait for the promise from axios.get("https://jsonplaceholder.typicode.com/users") to resolve or reject.
* Thus, after we send the api request with axios.get but before we consume the promise via .then, we exit the fetchUsers return function, leaving line 78 and continuing to line 79.
* In line 79, we execute the unsubscribe function, so our application no longer listens to any changes in the redux store.
* Whenever the promise resolves/rejects, the .then/.catch functions consume the promise and dispatches a new action which will cause the redux store to be updated. However, since the application is no longer subscribed to the redux store, the application does not know about the updated state in the redux store.
* Note that it doesn’t matter if the promise from axios.get instantly resolves/rejects, the output will be the same since the .then and .catch methods are pushed into the task queue. Thus, .then and .catch are executed only after the stack frame is empty. In order for the stack from to be empty, the unsubscribe method in line 79 must be executed. Thus, the unsubscribe method will be executed before the promise from axios.get gets consumed via .then or .catch.
* To show that the state in the redux store has indeed updated, we can add the following code:
* Text

  Description automatically generated
* Output: Text

  Description automatically generated

**React Redux: Setup**

* First, we create a react app
* Then, we run in cmd: ‘npm install redux react-redux’ to install the two libraries we will need

**React Redux: Actions**

* There are different folder structures that people use, there is no one right way.
* We will use the following folder structure:
* We will create a folder called ‘redux’ in the ‘src’ folder. Everything to do with redux will be contained in this particular folder.
* Then, we can create folders by feature. For instance, if we refer to the cake/ice cream shop example, the cake could be one feature and the ice cream shop could be another feature. For now, we’ll just create the cake feature by creating a folder called ‘cake’ inside the ‘redux’ folder.
* Inside the ‘cake’ folder, we will create a new js file called ‘cakeTypes.js’. Within this file, we will define our string constants that represent actions.
* Text

  Description automatically generated
* Inside the ‘cake’ folder, we will create a new js file called ‘cakeActions.js’. Within this file, we will define our action creator which uses the constants from the ‘cakeTypes.js’ file.
* Text

  Description automatically generated
* The folder structure should look like the following:
* Graphical user interface, application

  Description automatically generated

**React Redux: Reducers**

* Inside the ‘cake’ folder, we will create a new js file called ‘cakeReducer.js’. Within this file, we will define our reducer along with the initial state. When creating the reducer, we will need to import the actions from ‘cakeTypes.js’
* Text

  Description automatically generated
* The folder structure should now look like the following:
* Text

  Description automatically generated

**React Redux: Store**

* Inside the ‘redux’ folder, we will create our redux store.
* A screenshot of a computer

  Description automatically generated with low confidence
* Notice that we are importing the createStore function from the ‘redux’ package that we installed via npm. Also note the use of ES6 import syntax.
* Text

  Description automatically generated

**React Redux: React-Redux Library**

* We have created our redux store, now we need to provide our store to our react application, so all the components have access to the store.
* To do so, we will make use of the react-redux library that we earlier imported.
* The react-redux library exports a component called Provider which allows us to provide our redux store to our react application.
* In the App component, we will import the Provider component from ‘react-redux’
* Then, we will wrap the Provider component around the JSX that the App component returns. By placing the Provider component around the JSX that the App component returns, basically all the components will have access to Provider.
* In order to let the Provider component know about our redux store, we have to pass in the redux store as the value to the ‘store’ prop in the Provider component. Thus, we also have to import redux store.
* Text

  Description automatically generated

**React Redux: Subscribing to Redux Store and Dispatching Actions**

* We will create a component called CakeContainer and it will exist inside a components folder inside the src folder. This cakeContainer will replace the div inside the App component. The cake container will initially look like the following.
* Text

  Description automatically generated
* In this component, we want to be able to subscribe to the redux store so we can display the number of cakes there are. As well, we also want to be able to dispatch a BUY\_CAKE action upon button click.
* To do so, we will need three steps.
* Step 1
  + First, we will define a new function called mapStateToProps.
  + We can name it anything we want, but it is convention to name it as the above.
  + This mapStateToProps will be used to access the desired properties of the redux state in our CakeContainer component.
  + This function takes in the redux store’s state as a parameter and returns an object. In this cakeContainer component, we want to access the ‘numOfCakes’ state property from the redux store. Thus, we return an object with a ‘numOfCakes’ property and the corresponding value will be state.numOfCakes.
  + Text

    Description automatically generated
  + Aside: In the official react-redux documentation, they maintain a separate file called selectors (similar to how we maintained a different file for actions and reducers). This file returns some state information from the redux store. In our example, selecting the number of cakes is pretty easy since we just need to write ‘state.numOfCakes’. However, in larger applications, we may perform some data transformations and then select only what is required, which can get very complicated and deserves to be in its own file.
* Step 2
  + Second, we will define a new function called mapDisptachToProps
  + We can name it anything we want, but it is convention to name it as the above.
  + This mapDisptachToProps will be used to dispatch the BUY\_CAKE action in our CakeContainer component.
  + This function takes in the redux dispatch method as a parameter and returns an object. In this cakeContainer component, we want to dispatch a BUY\_CAKE action. Thus, we return an object with the a ‘buyCake’ property and the corresponding value will be a function. This function dispatches the BUY\_CAKE action created by executing the buyCake action creator.
  + Text

    Description automatically generated
  + Note that we will have to import the buyCake action creator from the cakeActions.js file. While we could directly import the buyCake action creator from cakeAction.js into CakeContainer.js, we can also create one single js file that will export all our action creators. Thus, in the ‘redux’ folder, we can create a file called index.js which looks like the following:
  + A screenshot of a computer

    Description automatically generated with low confidence A screenshot of a computer

    Description automatically generated with medium confidence
  + Then, we can import this buyCake action creator into the CakeContainer component as shown below.
  + 
* Step 3
  + Now, we will connect the mapStateToProps and mapDispatchToProps with the react component.
  + To do so, we will use the ‘connect’ function that returns a HOC from the react-redux library.
  + 
  + When we export the CakeContainer component, we can apply the connect function (note the use of currying)
  + 
  + This connect function connects the CakeContainer component to the redux store.
  + Provider provides the redux store the whole app, and when a component is connected (using connect()) it also is now subscribed to the state changes.
  + The connect function also enables the mapStateToProps and mapDispatchToProps functions to do the following:
    - In the mapStateToProps function, the state from the redux store is mapped to our component props. Thus, the CakeContainer also receives a new ‘numOfCakes’ props which reflects the value of state.numOfCakes. Note that CakeContainer’s ‘numOfCakes’ props is named ‘numOfCakes’ because that was the name we chose for the property in the object that the mapStateToProps returns.
    - Similarly, in the mapDispatchToProps, the dispatch of the BUY\_CAKE action will be mapped to our component props. Thus, the CakeContainer also receives a new ‘buyCake’ props which dispatches the BUY\_CAKE action upon function execution. Note that CakeContainer’s ‘buyCake’ props is named ‘buyCake’ because that was the name we chose for the property in the object that the mapDispatchToProps returns.
  + Thus, our CakeContainer component now looks like the following:
  + Text

    Description automatically generated
  + Webpage Output: A picture containing text

    Description automatically generated
  + Note that when we click the ‘Buy Cake’ button, the BUY\_CAKE action is dispatched and handled by the reducer which decrements the ‘numOfCakes’ property in the redux store state object by 1. This change is then reflected in the application since the application is subscribed to the redux store.

**React Redux: Hooks**

* In React Redux version 7.1, hooks were added so React Redux now offers a set of hook APIs as an alternative to the existing connect higher-order component.
* The APIs allow you to subscribe to the redux store and dispatch actions without having to wrap your components in connect
* In the following points, we will learn how to use hooks instead of connect

**React Redux: useSelector**

* useSelector is a hook that the React Redux library provides which acts as an alternative to the mapStateToProps function.
* We can use the useSelector hook to get the value associated with any state property within the redux store.
* To use useSelector, we must import the hook from ‘react-redux’
* 
* Inside the component that we want to access the state in, we call the useSelector function. This function takes in a function callback as its parameter. This callback function is called the selector function. This selector function takes in the redux store’s state as its parameter and returns a value. In theory, this value can be anything such as 3 or ‘hello’. But practically speaking, the return value of the selector function often is a property of the state that we are trying to access. The useSelector hook returns whatever is returned by the selector function and we can store that return value in a constant. We can then use this constant as our state variable property within the component.
* For instance we can use useSelector to access the numOfCakes property from the redux store.
* 
* Thus, the new component that uses useSelector instead of mapStateToProps would look like the following:
* Text

  Description automatically generated
* Note that the above component does not have the ability to dispatch actions yet. But the constant numOfCakes reflects the redux store’s ‘numOfCakes’ property. When the ‘numOfCakes’ property of the redux store’s state changes, the HookCakeContainer component will rerender and update the value of the numOfCakes constant.

**React Redux: useDispatch**

* useDispatch is a hook that the React Redux library provides which acts as an alternative to the mapDispatchToProps function.
* useDispatch is a hook that the React Redux library provides which allows us to dispatch actions.
* To use useDispatch, we must import the hook from ‘react-redux’
* 
* Inside the component that we want to dispatch the action in, we call the useDispatch function. This function doesn’t take any parameters and returns a reference to the dispatch function from the Redux store. We can store that reference in a constant called dispatch. This dispatch constant can be used to dispatch actions as needed.
* For instance, we can use useDispatch to dispatch the BUY\_CAKE action to the redux store. Note that we would have to import the buyCake action creator and pass it as a parameter to the dispatch constant.
* 
* Thus, the new component that uses useDisptach instead of mapDisptachToProps would look like the following:
* Text

  Description automatically generated