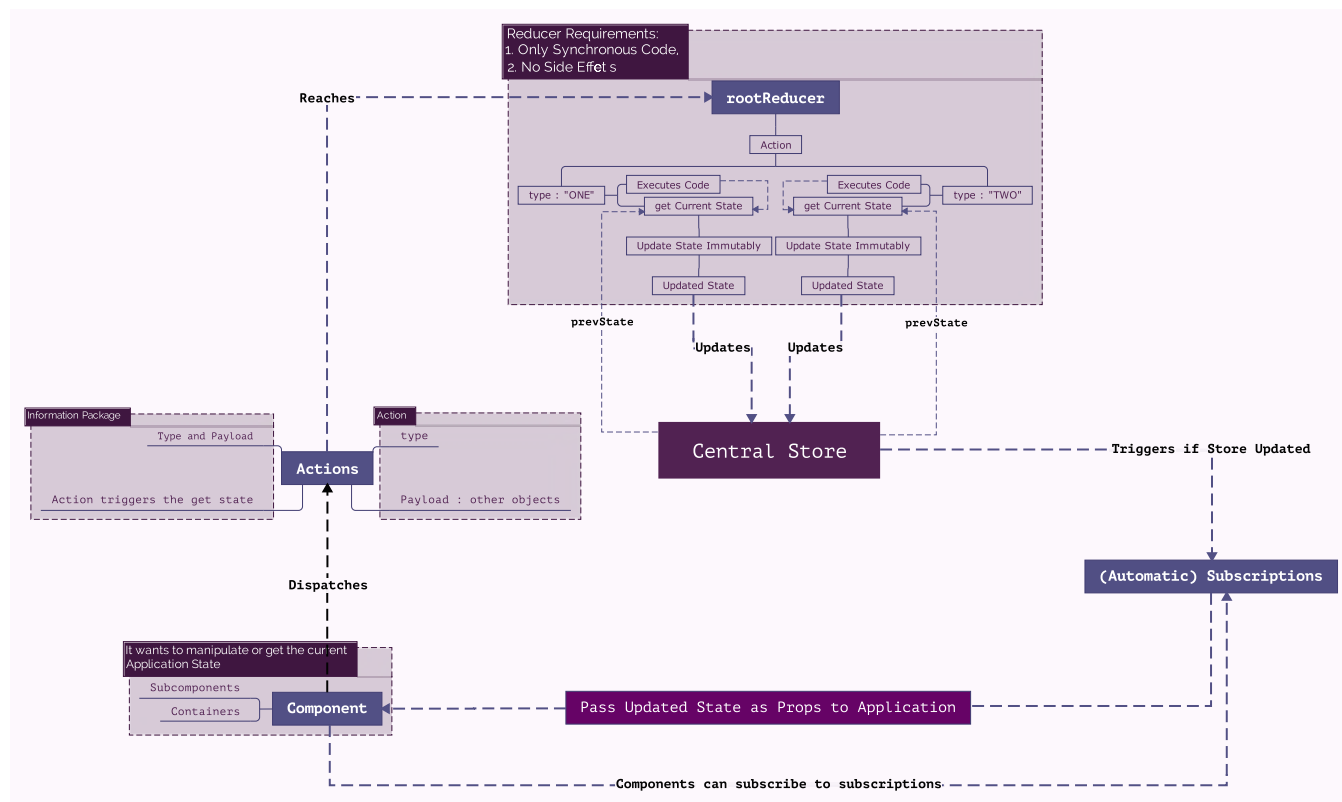


REDUX – Understanding it



Redux is a standalone package and it can work without React. To use redux, we need Redux

```
const redux = require ('redux');  
const createStore = redux.createStore;
```

Below are the details of different components of Redux.

Reducers

1. It is the only thing that is strongly connected to the store. It holds the logic when to update the state.
2. Action is the javascript object that has the type and other payload objects
3. The logic is binded to action.type. Example below.

```
const initialState = {  
  counter: 0  
}  
  
const rootReducer = (state = initialState, action) => {  
  switch (action.type){  
    case 'INC_COUNTER':  
      console.log(`In ${action.type} type`);  
      return {  
        ...state,  
        counter : state.counter + 1  
      }  
      break;  
    case 'ADD_COUNTER':  
      console.log(`In ${action.type} type`);  
      return {  
        ...state,  
        counter : state.counter + action.value  
      }  
      break;  
    default:
```

```

        return state;
    }
    return state;
}

```

A *reducer* (also called a *reducing function*) is a function that accepts an accumulation and a value and returns a new accumulation. They are used to reduce a collection of values down to a single value.

Reducers are not unique to Redux—they are a fundamental concept in functional programming. Even most non-functional languages, like JavaScript, have a built-in API for reducing. In JavaScript, it's `Array.prototype.reduce()`.

In Redux, the accumulated value is the state object, and the values being accumulated are actions. Reducers calculate a new state given the previous state and an action. They must be *pure functions*—functions that return the exact same output for given inputs. They should also be free of side-effects. This is what enables exciting features like hot reloading and time travel.

Reducers are the most important concept in Redux.

Store

1. it will take reducer as the parameter.
2. Since it takes reducer as a parameter, the reducer needs to be created and available before creating the store.

```

const store = createStore(rootReducer);

```

Subscriptions

1. It eliminates the need to asking `getState` to understand the state change
2. It will inform me whenever the state changes if subscribe to it.
3. Subscribe take function as argument
4. Subscription is setup right after the store is created so that we will get the details from initial state.

```

store.subscribe( () => {
    console.log(`You are subscribed and here is your detail`)
    console.log(store.getState());
})

```

Dispatch

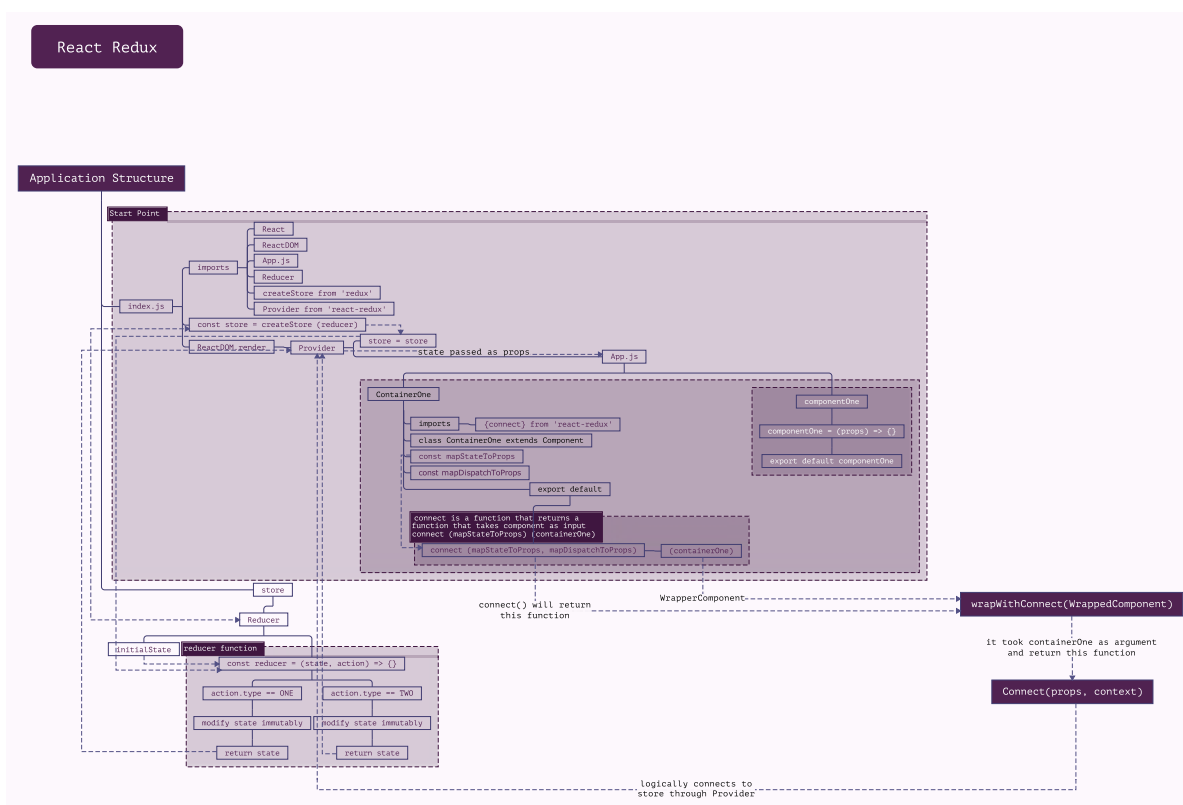
1. Dispatch Action – Code Dispatching an action
2. Action is an javascript object
3. 'type' property is must and important. The value is an unique identifier and the convention is to use all UPPER CASE

```

store.dispatch({type: 'INC_COUNTER'});
store.dispatch({type: 'ADD_COUNTER', value: 5});

```

Application conventions



Connecting the Redux to React

1. Install redux from npm as dependency

```
npm install --save redux
```

2. Most of the application will have their store created on the index.js file which will mount the root container App.js

```
/store/reducer.js
const initState = {
  counter: 0
}
const reducer = ( state = initState, action ) => {
  return state;
}
export default reducer;

./index.js
import { createStore } from 'redux';
import reducer from './store/reducer';
const store = createStore(reducer);
```

3. As mentioned above most scenarios, we will need a separate place to store the reducers. It is not mandatory but conventionally way for easy use.

Connecting the store to the React application

4. We need to install react-redux as the dependency to connect store to react

```
npm install --save react-redux
```

5. Import the Provider from react-redux and wrap it to the root component

```
import { Provider } from 'react-redux';
ReactDOM.render(<Provider><App/></Provider>, document.getElementById('App'));
```

5. Provider is the helper component which allows us to inject the store into the React Component. For hooking up the provider component with the store, need to setup the property that passes the created store.

the property name is store and we should pass the store to it

```
ReactDOM.render(<Provider store={store}><App/></Provider>,
document.getElementById('App'));
```

6. To get the state from the central store to the container or the component, we need to set the subscriptions in the respective component. We can use the feature provided by the react-redux package.

Still the containers/components will pass the state to its child components. The only thing that changes – components/containers will not maintain the state but the central store does.

7. We need to import 'connect' which is a function which will return the function as a higher order component which takes then the component as its input

```
'./containers/Counter/Counter.js'
import { connect } from 'react-redux';
class Counter extends Component {
  ...
}
export default connect()(Counter);
```

8. We pass two piece of information to the connect function

- which slice of the state that you want it in return in this container(since the central store maintain number of state that belongs to various components/containers)
 - We will get the slice of the **state from the central store as a props but not as a state** to our container because state is the thing which can be changed only internally but props doesn't change internally rather it is passed by the parent container. Since the idea is to manage the state to separate component, we are receiving those state only as props from central store.
- which action that I want to dispatch (since a bigger application will have lot of actions that manipulate the state)

Need to define the slice of the state that needed after the class definition

```
'./containers/Counter/Counter.js'
import { connect } from 'react-redux';
class Counter extends Component {
```

```

    render () {
        //The counter component will receive the state from centra store via
        connect as props hence accessing like this.
        return <CounterOutput value={this.props.ctr}
    }
}
//const name can be anything
const mapStateToProps = state => {
    return {
        //it returns the jsx object, we will map the counter property of the state
        to counter locally here
        ctr: state.counter
    };
}
//connect will return a function will take Counter component as its input and thus
giving access Counter to mapStateToProps.
export default connect(mapStateToProps)(Counter);

```

Dispatching actions from within the component

9. We can pass the second configuration mapDispatchToProps and pass it to connect function

```

class Counter extends Component {
    render () {
        //The counter component will receive the dispatch function through props
        hence accessing like this. The dispatch function will be same as we defined below.
        This has the two-way binding dispatch function which then be executed at reducers
        via action parameter by handling it through event here.
        return <CounterControl label="Increment" clicked=
        {this.props.incCounterHandler} />
    }
}
//mapDispatchToProps is a constant that return the Object which will have a key
mapped to dispatch function
const mapDispatchToProps = dispatch => {
    return {
        incCounterHandler: () => dispatch(type: 'INCREMENT')
    }
}
//By passing these two arguments, we are requesting both the slice of the state
and dispatching action.
export default connect(mapStateToProps, mapDispatchToProps)(Counter);
//If we just need to dispatch action but don't need a slice then, we can pass
null as the slice needed
export default connect(null, mapDispatchToProps)(Counter);

```

Outsourcing the action types.

10. It is best practise to outsource the action types to a separate file and importing them into reducers and the component which will dispatch actions. This will eliminate the TYPO that likely to occur and highlight it to us on the development mode itself

```

'./store/actions.js'
export const INCREMENT = 'INCREMENT';

'./store/reducer.js'
//import all the exported const from actions.js as actionTypes object.
import * as actionTypes from './actions';
const reducer = ( state, action) => {
  switch(action.type){
    case actionTypes.INCREMENT:
      DO SOMETHING
      return state
  }
}
export default reducer

'./containers/Counter/Counter.js'
//import all the exported const from actions.js as actionTypes object.
import * as actionTypes from './actions';
const mapDispatchToProps = dispatch => {
  return {
    incCounterHandler: () => dispatch({type: actionTypes.INCREMENT})
  }
}
export default connect(null, mapDispatchToProps)(Counter)

```

Having multiple reducers and combining multiple reducers

11. Redux gives us a helper function called *combineReducers* from 'react' to combine multiple reducers to one reducers and store will handle only one reducer
12. Conventionally we can have a separated folder reducers in store and have the reducers created within it.
13. combineReducers is a function which takes javascript object mapping our reducers that has different slices of our state as input and merges them into one state and one reducer.

```
combineReducers(reducers: ReducerMapObject): (state: any, action: AnyAction) => any
```

```

import counterReducer from './store/reducers/counter';
import logsReducer from './store/reducers/logs';
import { createStore, combineReducers} from 'redux';

//rootReducer is a new function of combineReducers that takes our other two
reducers as object input
const rootReducer = combineReducers({
  counter: counterReducer,
  logs: logsReducer
});

const store = createStore(rootReducer);

```

At the end, we will have only one state since we are combining all reducers however to avoid naming

conflicts between the reducers, redux add one level of nesting to the reducer keys that we define in the input. In above case, the counterReducer objects can be access through counter.objects & logsReducer can be accessed through logs.objects. Example

```

'./store/reducers/counter.js'
const initState = {
  counterValue: 100
}
const counterReducer = ( state = initState, action ) => {
  return state
}
export default CounterReducer

'./container/Counter/Counter.js'
class Counter extends Component {
  render() {
    return (
      //this.props since the state is passed as props via Provider
      //this.props.counter to access the counter object that has
      counterReducer as value in the rootReducer
      //this.props.counter.counterValue to access the counterValue set at
      the state maintained by counterReducer
      <p>{this.props.counter.counterValue}</p>
    )
  }
}
```

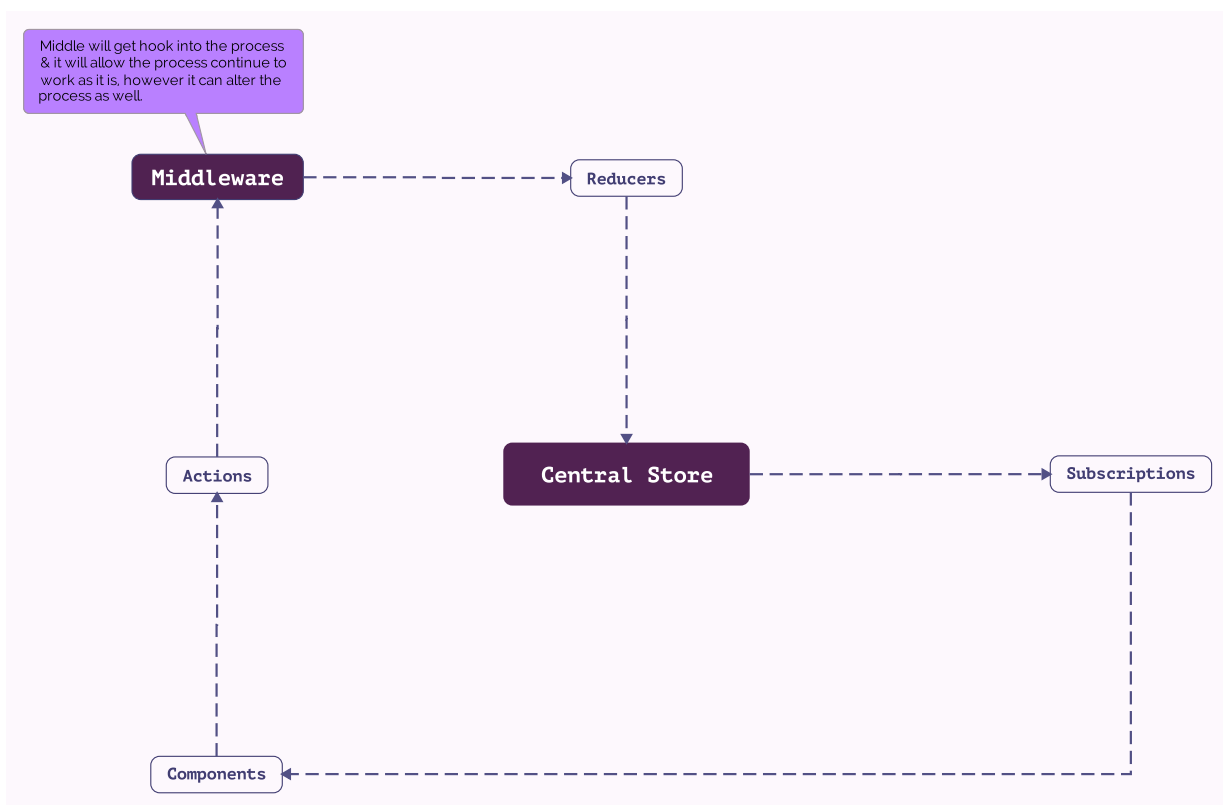
Understanding state types and determining when to use redux more sensibly.

14. You can use redux to store and manage your state for all kind of components and information however, using it sensibly will reduce the overload of setting up the redux and will help in speedy programming. Some of the state scenario and their necessity to use redux is briefed below.

Type	Example	Use Redux?	Determination
Local UI State	Show/Hide Backdrops	Mostly Handled within components	
Persistent State	Server Side DB info – All Users, All Posts	Stored on the server only slices (current user, current posts) which needs to be rendered will be managed by Redux	Redux is to manage the state as long as the application is live (refresh will kill)
Client State	Is User Authenticated? Filters to veiw specific posts	Managed via Redux	Cannot be maintained on server side as isAuth? will not always be true

Advanced Redux

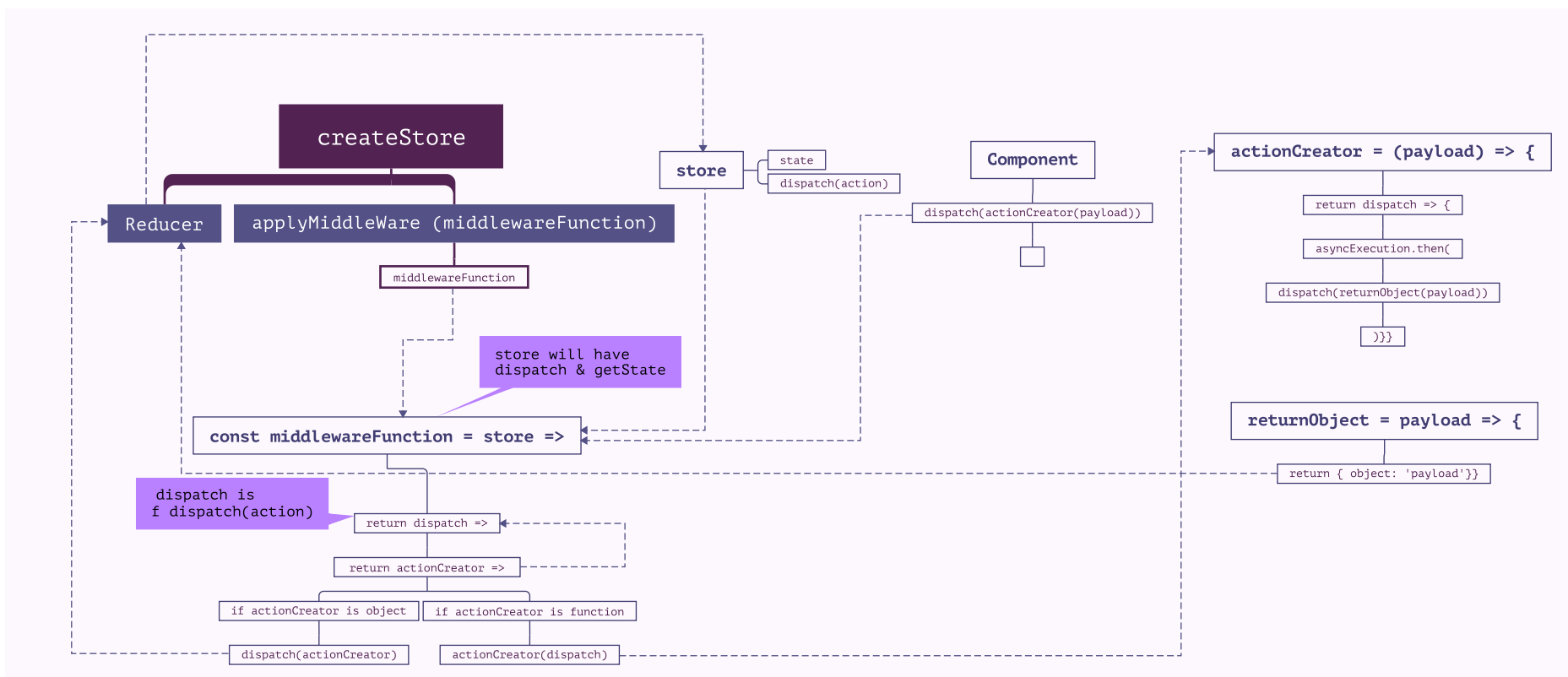
Applying Middleware



Middle will get hook into the process & it will allow the process continue to work as it is, however it can alter the process as well. In Redux, middlewares are applying between dispatch & reducers function where the actions will be dispatched from dispatchers & reach Reducer through middleware. It will be important to execute any asynchronous code.

Applying a simple self-created middleware

Middleware is predominantly a function, it will get store as an input. If we connect our middle to the store using function provided by redux, it will execute this function and give us the store



```

'@ ./src/index.js'
import { createStore, applyMiddleware } from 'redux';
// if we connect this middle using the function provided by redux, it will execute
// our middle ware function and give us the store,
// hence passing store as an argument to this function body
const logger = store => {
  console.log('[logger function] that gets store ', store);
  // we will return another function called next (name is upto you, since we are
  // telling it what to do next we name it as next)
  return next => {
    console.log('[function] that get next as argument ', next)
  }
}
  
```



```

        // the next function will return another function by giving the action
that we dispatch to it
        return action => {
            console.log('[function] that gets action as argument ', action)
            // from here we will have action to next as well as action as well as
store. So we can modify both before we giving it to reducer but need
            // to do it with caution as it will break the work-flow.
            const result = next(action);
            console.log('[Middleware] next state ', store.getState());
            // we are simply calling the next function with action as argument to
proceed with its normal workflow to reducer.
            return result;
        }
    }
}
const store = createStore(rootReducer, applyMiddleware(logger));

```

Asynchronous code

The asynchronous code is the code that will take some time to get determined due to server side delay or latency over network.

Reducers should not cause side effects since it has the features like hot reloading and time travel. We can execute the Asynchronous code with the help of action creators and middlewares.

Action Creators

Instead of using Action Identifiers, we can use the Action creators. Action creator is a function that returns an action object. Predominantly, instead of hardcoding the actions to dispatch function at the components/container, we can use action creators to return that action to dispatch.

```

'@ ./actions/actions.js'
export const INCREMENT = 'INCREMENT';
export const ADD_FIVE = 'ADD_FIVE';
export const increment = () => {
    return {
        type: INCREMENT
    }
}

export const logResult = (value) => {
    return {
        type: LOG_RESULT,
        value: value
    }
}

```

thunk middleware

redux-thunk is a library which adds middleware to our application which will allow us to write action creators that return a function instead of an action.

What does it do? just as below,

```
'@ ./index.js'
//THUNK alike custom middleware
// By default, the applyMiddleware wrapper will give the store to our middleware
const logger = store => {
  // store have dispatch (next argument here) function (initiated by our
  component which have action object) and getState function
  return dispatch => {
    // we are taking the action object parameter from dispatch and passing it
    to the function where we can manipulate it before it is getting passed to reducer
    action => {
      // if action is a function, then execute that function passing them
      the state & dispatch. You would've created an action creator that returns function
      to match this situation and returning the object there after performing
      asynchronous task. See below action.js code
      if (typeof action === 'function' ){
        return action(dispatch, store.getState);
      }
      // if action is an object then pass it to reducer through dispatch
      function
        dispatch(action);
      }
    }
  }
}

'@ ./actions/actions.js'
const asyncLogResult = (resultValue) => {
  return {
    type: LOG_RESULT,
    value: resultValue
  }
}

// this is the actionCreator function that receives value as argument passed by
the component to dispatch function and it returns function instead of object
export const logResult = (value) => {
  // we will receive the dispatch from the middle when it called this
  actionCreator function by passing state & dispatch. See above logger function.
  return dispatch => {
    setTimeout(() => {
      dispatch(asyncLogResult(value))
    }, 2000);
  }
}
```