**About Redux**

Redux is an application data flow architecture which can be used with any view library to manage the state of a web application. It was introduced by Dan Abramov. It is inspired greatly by functional programming language and FLUX architecture.

Developers choose Redux to handle the state changes in a common place with more consistency. Redux helps the independent components (which do not share parent child relationship) to interact with each other. When multiple components are trying to manipulate and display the same data, Redux helps the components to keep the data in sync.

In this course we will discuss the about how to handle events by triggering actions, how to modify the state using reducers, how does the store handles the entire state of the application. This course also explains on how to handle asynchronous calls using middleware in Redux. This course deals with how to bind React with Redux. This course also explains about the server side rendering in Redux.

**Why Redux**

Many applications are built using React because of its component architecture, usage of virtual DOM for DOM manipulation and the ability to think about any application in terms of small pure functions.

When the data flow becomes more complex we may want to display same data in multiple places and we may have lot of state changes which would be hard to manage. We may feel that it would be great if we can handle the state changes in a single place for more consistency. Redux helps in managing the state changes.

**Component initiating a change**

Whenever a component initiates a change, instead of each component updating its state property and communicating the same to other components each component update the store (where the complete application state is present) and all the other components gets the updates state from the store.

Redux is useful for applications that have complex data flows. If we need to handle interactions between two components which do not have parent-child relationship and both the components are trying to manipulate and display the same data then Redux offers an elegant solution to keep the data in sync.

Redux is preffered over Flux because of it's 3 main advantages

1. Hot reloading

2. State Immutability

3. Time travel debugging

**State Immutability**

We need to keep track of each version state object for debugging. So each version needs to be an entirely separate object so that we are not accidently changing.

Flux re-writes it's state with every action triggered.

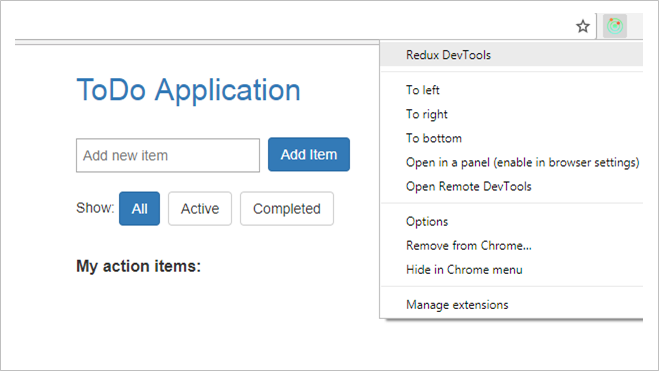
Redux solves the problem by creating a copy of the current state object and modifying the copy whenever an action is triggered.

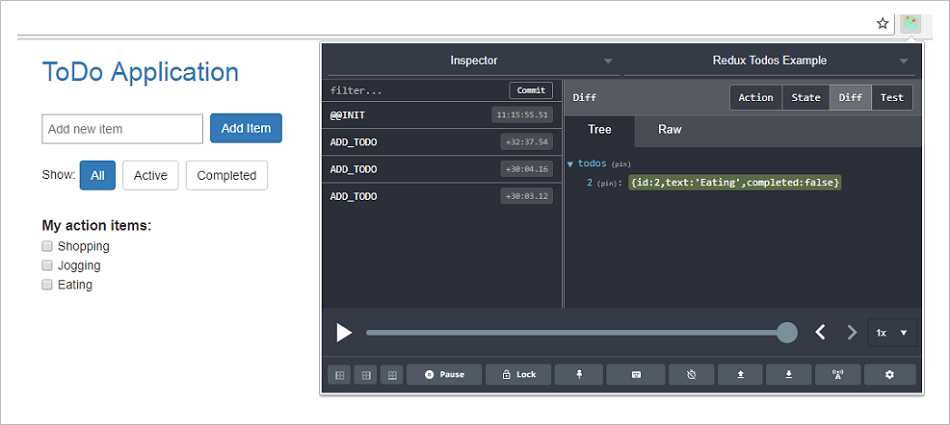
This helps in improving performance.

**Time Travel Debugging**

Time Travel Debugging: Time travel debugging is a powerful way to see exactly how your application state is changing over time. We can travel through time as you debug, so we can go back in history and see each specific state change. We can also undo specific state changes and observe how final state changes. We can even turn off individual actions that occurred so you observe what would be the state object if a specific action had not happened. We can play all the interactions back with the click of a button and even select the speed at which it plays back.

Time Travel debugging can be done using Redux developer tools which is available as an add-on to the browser. Once the add-on is installed in the browser, we can use it to debug our Redux application as shown below.

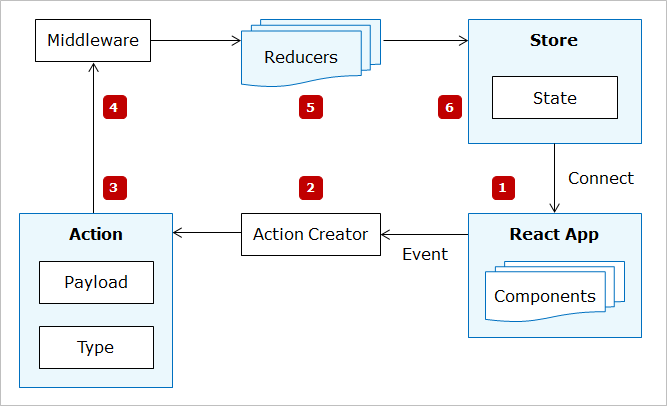




**What is Redux**

Redux is not a framework, it is an application data-flow architecture which is used for predicting state changes in JavaScript applications. Redux can be used with any other View library but mostly used with React.

**Redux Data Flow**



1. **React Components:** React components represents the UI rendered on the browser. Components would dispatch actions for events triggered within them and whenever state changes the component renders the current state by connecting to Redux store.
2. **Action Creators:** Action creators are functions which wraps the actual action object.
3. **Action:** Action is a plain JavaScript object. Action is a command to change the state when an event is triggered.
4. **Middleware:** Middleware is a mediator between the action and reducer. Its purpose is to intercept the actions before it reaches the reducer.
5. **Reducer:** Reducer changes the state of the application based on the action triggered. Reducer are functions which accepts action and current state and modify the current state by creating a copy of it based on the action.
6. **Store:** Store is responsible for managing the entire state of the application. State management is centralized in Redux. Action will be dispatched to store using dispatch method of store.

# Redux Data Flow explained with Login as an example

Let us observe the Redux data flow by considering Login as an example. Login form is represented by Login component.

1. When user clicks on the login button an action is dispatched.
2. Action reaches action creator which returns an action. Action object contains action type and payload information (if required)
3. Action reaches Reducer which modifies the state
4. If any operation needs to be performed before modifying state then action reaches middleware first and then reaches reducer
5. Reducer updates the modified state to the store
6. React components gets the updated state from the store

**State --- Why State**

State is plain JavaScript object which is used to handle data in React. State is reserved for interactivity in React.

In React individual components were associated with state, but in Redux the entire state of the application is contained in a single state object.

Unlike React, state is immutable in Redux. State can be changed only by dispatching actions in Redux.

**Actions and Action Creators**

In React, components were associated with state and the state changes happens within the component

In Redux since the state management is centralized, the components cannot modify the state directly. Hence Redux dispatches an action whenever there is an event triggered within a component.

**What is Action**

Action is a JavaScript object which describes an event. Action object must contain a type property which describes the type of the action triggered and optionally can have payload information. The payload information can be a number, a boolean value, a complex object, and any value that is serializable to JSON. The payload information cannot be a function or promises.

{

type:'LOGIN',

value:username

}

Action creators are the functions that return the action object.

function handleLogin(username)

{

return

{

type: LOGIN,

username

}

}

# 

# Need for action creators

Actions creators are called convenient functions. It is not mandatory to define action creators to return the action rather we can just use the action object for dispatching an action, but action creators helps us in achieving

* Maintainability. Suppose if we want to change the structure of the action object at the later point of time, with action creators we can change it at one place but with action objects, we need to change the structure wherever the action is triggered (if same action is used in multiple places).
* Abstraction and encapsulation because the component where the action is triggered need not know about the action creator structure.
* Testability as we can create mock versions for action creators. Hence it is always good to have action creators though it adds a little boilerplate to the code.

**Reducers**

* When an action is triggered within any component the state modification is done using Reducers in Redux.
* Reducers contain the logic for modifying the state in Redux. State is immutable in Redux, hence state cannot be directly modified by reducers. Reducers in Redux are responsible for copying the current state and modifying the copy and returning the new state.
* Before knowing how reducer modifies the state in Redux let us learn about immutability.

**Why Immutability**

The benefits of immutability is

**1. Clarity:**

If we are referencing state object in one part of the application and the state object changes in some dramatic way at an unexpected time, not only your application will stop working, but you may have a hard time figuring out what’s going wrong. Immutability gives clarity on who changed the value of the state and also it would be easy to track the state updates while debugging.

**2. Performance:**

Consider a state object which has many properties in it. If state is mutable, in order to track the state changes, each and every property has to be tracked by Redux which is an expensive operation. Since state is immutable every time state is modified a new state object is returned. Hence Redux just does a reference comparison. If the old state is not referencing to the same object in memory then it is evident that the state has been modified. Immutability increases performance. React-Redux provides lot of performance optimizations behind the scenes.

**3. Awesome debugging experience:**

Immutability helps in debugging. Time travel debugging is a powerful way to see exactly how your application state is changing over time. We can travel through time as you debug, so we can go back in history and see each specific state change. We can also undo specific state changes and observe how final state changes. We can even turn off individual actions that occurred so you observe what would be the state object if a specific action had not happened. We can play all the interactions back with the click of a button and even select the speed at which it plays back

**What is Immutability**

Immutability means objects cannot be changed. To change immutable objects we need to create a copy of it and modify the copy.

In Redux state is an immutable object. State object cannot be changed directly in Redux. Triggering actions are the only way to modify the state. Every time the state is modified a new object should be returned which represents the modifications that are made to state object.

# Handling immutability

There are many ways to handle immutability. Object.assign and spread operator for arrays are the most popular approach when working in ES6.

**Object.assign()**

Consider state object a shown below

state={status: 'logged out', value: 'guest',email:'',tel:''}

When Login action is triggered we want to change only the status and value properties. We can use Object.assign to create a copy and modify only few properties as shown below.

Object.assign({}, state, {

status: 'logged in',

value: action.value

})

The first parameter to Object.assign is the target which is a new empty object, then we are mixing the new object together with our existing state and also changing the status and value properties. So the result is effectively a clone of our state object but with the state modified after login.

**Spread(...) operator**

The spread operator is represented using 3 dots (...)

It copies own enumerable properties from a provided object onto a new object.

var obj1 = { foo: 'bar', x: 42 };

var obj2 = { foo: 'baz', y: 13 };

var clonedObj = { ...obj1 };

// Object { foo: "bar", x: 42 }

var mergedObj = { ...obj1, ...obj2 };

// Object { foo: "baz", x: 42, y: 13 }

**Why Reducers**

Reducers are used to achieve immutability in Redux. Reducers are pure functions which are responsible for handling the actions and modify the state of the application based on the action triggered.

Pure functions are functions which always returns the same result given the same parameters

Reducers are functions which accepts current state and an action object as arguments and will return the new state.

const auth = function(state = {status: 'logged out', value: 'guest'}, action)

{

switch (action.type)

{

case 'LOGIN':

return Object.assign({}, state, {

status: 'logged in',

value: action.value

})

}

}

Reducers would contain switch statement where it would check for the actions and whichever action matches the state is changed according to the action and a new copy of state is returned.

In the above code for login action the current state is modified by creating a copy of the current state using Object.assign() method.

**Forbidden in Reducers**

* 1. Mutating the arguments passed
  2. Performing side effects like API call, AJAX call etc
  3. Calling other non pure functions (Ex: date.now(), math.random())

**Loading the initial state**

Reducer accepts the current state and changes the state based on the action and updates the state to the store.

Initial state of the application should be passed to the Reducer.

Initial state of the todo application would look be an empty todo array which refers to the items in the Todo list and visibilityFilter whose value will be the default value SHOW\_ALL.

initialState={

todos:[],

visibilityFilter: visibilityFilters.SHOW\_ALL

}

This initial state object has to be passed to the reducer.

**Store -- Why store**

We know that in Redux the entire state of the application is present in a single state. Store is the container used to place the entire state of the application.

React Components can fetch the current state from the store. Actions are processed in the store by a reducer.

Creating store in Redux

Store can be created by using createStore method

let store= createStore(reducer)

Reducer is passed to the store. Since the state changes are done by reducer the reducer would update the store with the state changes done. Every time reducer changes the state it updates the store and store would contain the current state.

# 

**Store Methods**

1. **getState():** This method can be used to get the current state from the store.
2. **dispatch(action):** React components should use this method to dispatch an action whenever an event occurs within the component. This method dispatches an action and then the reducer take care of updating the state
3. **subscribe(listener):** Used for registering the listeners

We don't have any API's to change the data in the store. The only way to change the state present in the store is by dispatching actions. The store would handle actions using reducers.

**Combining Multiple Reducers**

To achieve modularity we can write multiple reducers in Redux.

Though we split the reducers as our application grows, ultimately we can pass only one reducer object to the createStore method. We can use combineReducers() method of Redux to combine multiple reducers in to a single reducing function and then pass it to createStore method.

const todoApp = combineReducers({

todos,

visibilityFilter

})

The todos reducer and visibilityFilter reducer are combined in to a single reducer todoApp and then the combined reducer is passed to the store.

var store = applyMiddleware(thunk)(createStore)(todoApp);

Whenever an action is triggered the action would reach the root reducer first and then all the reducers get the action from root reducer. Every reducer will check the action in the switch case. When the action matches the respective reducer will change the state and return the new state to root reducer. Whichever reducer doesn’t find a match will return the current state. Hence we will always have a default case in all the reducers which returns the current state without any modification.

The root reducer will combine the state from all the reducers and create a single new state object and will update the store with the new state.

import { combineReducers } from 'redux'

import todos from './todos'

import visibilityFilter from './visibilityFilter'

const todoApp = combineReducers({

  todos,

  visibilityFilter

})

export default todoApp

**Why Middleware**

Middleware need not be a pure function like reducers. It can cause side effects. So whatever functionality we could not put in to reducer because reducers are pure can be put in middleware

# Uses of middleware

1. It provides functionality between action and reducer. After action is triggered and before its reduced middleware can take action
2. Organize all changes to the database (ajax calls) in to one place
3. Middleware allows to pre-process actions. We can check for correct syntax, make sure they confirm to your standards, or edit them in some way like wrapping them in a function
4. Perfect for debugging an application
5. Allows us to keep actions clean so we need less boilerplate and less tests in our actions to make sure they are working correctly
6. API calls are happening through middleware so if there is a mistake we can check middleware first.

# Popular middleware Libraries

1. Redux Thunk
2. Redux Saga

**Asynchronous Operations with Middleware**

Middleware lets us to wrap the dispatch method of the store. The most common requirement to use middleware is to support asynchronous actions. Middleware lets us to dispatch async actions in addition to our regular actions.

Each middleware receives Store's dispatch and getState functions as named arguments, and returns a function.

createStore() method by default supports only synchronous data flow. In order to perform async actions we can use applyMiddleware() function

Redux thunk middleware allows the developer to write action creators which returns a function instead of an action object. The middleware can be used to delay the dispatch of an action, or dispatch the actions based on certain condition. The function which the action creator returns  receives the two arguments 1. dispatch method of store 2. getState method of store

var store = applyMiddleware(thunk)(createStore)(todoApp);

Use applyMiddleware method and pass thunk to it while creating the store.

thunk is available in the redux-thunk library.

import thunk from 'redux-thunk';

In the todo application you can observer the thunk function callAddToDo().

export const callAddToDo=text=>{

console.log("in callAddToDo");

return (dispatch)=>{

setTimeout(()=>{

dispatch(addTodo(text));

},3000)

}

}

The action creator CallIncrement is dispatching the onIncrement action after 3000 ms. Async operations can be written within the CallIncrement action creator.

Binding React Components with Redux

Since state is managed by the Redux store. In order to access and render the state properties, the React components need to connect to the Redux store.

We can write React components as:

Presentational components

Container components

**Presentational Components:** Presentational components are used to present the markup and styles.

**Container Components:** Container components are responsible for connecting to the Redux store.

Container components are used to fetch the updated state from redux store. Once container components get the state from Redux store they pass the updated state to presentational components as props.

Container components can be written by developer by subscribing to the store using store.subscribe() and the current state can be fetched from the store using store.getState() method.

But as already discussed React Redux has already implemented the necessary performance optimizations. Hence we generate container components using the methods provided by react-redux library, rather than writing the container components by hand.

**Container and Presentational Components**

# Implementing presentational component

Let us discuss about presentational and container components using few components of our Todo application.

The TodoList component is a presentational component

import React from 'react'

import PropTypes from 'prop-types'

import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (

<ul>

{todos.map(todo => (

<Todo key={todo.id} {...todo} onClick={() => onTodoClick(todo.id)} />

))}

</ul>

)

TodoList.propTypes = {

todos: PropTypes.arrayOf(

PropTypes.shape({

id: PropTypes.number.isRequired,

completed: PropTypes.bool.isRequired,

text: PropTypes.string.isRequired

}).isRequired

).isRequired,

onTodoClick: PropTypes.func.isRequired

}

export default TodoList

The todos and onTodoClick is passed as props to the TodoList component.

The ToDoList component loops through the Todolist and uses Todo child component to render each todo item.

import React from 'react'

import PropTypes from 'prop-types'

const Todo = ({ onClick, completed, text }) => (

<li

onClick={onClick}

style={{

textDecoration: completed ? 'line-through' : 'none'

}}

>

{text}

</li>

)

Todo.propTypes = {

onClick: PropTypes.func.isRequired,

completed: PropTypes.bool.isRequired,

text: PropTypes.string.isRequired

}

export default Todo

# 

# Implementing container component

The TodoList component should be hooked to Redux, hence we define a container component

To generate container component the react-redux library provides connect() function

To use connect we need to define two functions

mapStateToProps

mapDispatchToProps

These 2 functions are passed as arguments to connect function

**mapStateToProps:** This function helps in transforming the current state present in the redux store as props and then the same is passed to it's corresponding presentational component.

In the ToDo list example todos state property is required by the TodoList presentational component

Hence VisibleToDoList container component is generated and mapStateToProps function is defined as shown below

const mapStateToProps = state => {

return {

todos: getVisibleTodos(state.todos, state.visibilityFilter)

}

}

The mapStateToProps function accepts the current state as an argument. From the entire state present in the redux store we can map todos and visibilityFilter as props.

## **Passing Store to All the Components Within an Application**

In order to subscribe to the store and get the state from the store, the container components need access to the Reduce store.

We can pass the store as a props to all the container components, but it is not a feasible solution if the number of container components are more in number and the store needs to be passed to the presentational components as well.

Hence react-redux provides an in-built <Provider> component which provides access to the store to all the container components in an application. When rendering the root component to the DOM, the root component should be wrapped within the <Provider> component as shown below.

render(

<Provider store={store}>

<AppComp />

</Provider>,

document.getElementById('root')

)

Passing Store Using Provider in Todo Application

import React from 'react';

import { render } from 'react-dom';

import { Provider } from 'react-redux';

import App from './components/App';

import store from './store/store';

render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

## **Server Rendering**

We know that SPA gives best user experience as the websites wont constantly reload and load on the server is also reduced because most of the rendering happens at client side.

One of the major disadvantage of SPA is slow initial loading time. JavaScript would take some time to render the HTML which will make the SPA's to wait. The end user will have to wait for a long time for the initial page to load.

Server-Side Rendering solves this problem. Server will fetch the content required to render the initial HTML page and then send it to the client.

The client would get the initial page from the server. The first page is loaded from the server and the rest of the rendering after that happens at the client side using JavaScript. The initial page load would be faster when it is rendered from the server.

Server side rendering would involve the below steps:

1. fetch the content which is required to the initial page
2. render the initial HTML page using the content fetched
3. send the HTML to the client