Understanding the concept behind using reducer and actions using JavaScript

If we want to change a particular value by preserving all the properties then-

example

```
let stateObj={
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state){
    return {
        ...state, value:state.value+1
    }
}

stateObj=myReducer(stateObj);
console.log(stateObj);
stateObj=myReducer(stateObj);
console.log(stateObj);
stateObj=myReducer(stateObj);
console.log(stateObj);
```

example

```
let stateObj={
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state, action){
    console.log(action);
    return {
        ...state, value:state.value+1
    }
}

stateObj=myReducer(stateObj, {type:'change/modify'});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/modify'});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/modify'});
console.log(stateObj);
```

example

```
let stateObj={
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state, action){
    if(action.type==='change/incr')
    return {
        ...state, value:state.value+1
    }
    else if(action.type==='change/decr')
        return {
            ...state, value:state.value-1
        }
}

stateObj=myReducer(stateObj, {type:'change/incr'});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/decr'});
console.log(stateObj);
```

Example : if we don't get the appropriate action to perform

```
let stateObj={
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state, action){
    if(action.type==='change/incr')
    return {
        ...state, value:state.value+1
    }
    else if(action.type==='change/decr')
        return {
            ...state, value:state.value-1
        }
    return state;   // if we don't get the appropriate action to perform then return
the previous state
}

stateObj=myReducer(stateObj, {type:'change/incr'});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/decr'});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/dont_know'});
console.log(stateObj);
```

Example : Changing value through payload

```
let stateObj={
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state, action){
    if(action.type==='change/incr')
    return {
        ...state, value:state.value+action.payload
    }
    else if(action.type==='change/decr')
        return {
            ...state, value:state.value-action.payload
        }
    return state;   // if we don't get the appropriate action to perform then return
the previous state
}

stateObj=myReducer(stateObj, {type:'change/incr', payload:5});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/decr', payload:10});
console.log(stateObj);
stateObj=myReducer(stateObj, {type:'change/dont_know'});
console.log(stateObj);
```

Before proceeding with redux next practices, lets talk about Parcel js

Parcel.js

- Parcel.js is a modern, fast, and zero-configuration web application bundler.
- It is designed to make web development easier by handling the complexities of bundling, transpiling, and optimizing assets with minimal configuration.
- Parcel.js is a powerful tool that simplifies the process of building and bundling web applications.
- Its zero-configuration approach, fast performance, and built-in support for modern web features make it an attractive choice for developers looking for an easy-to-use bundler with robust capabilities.
- Whether you are working on a small project or a larger application, Parcel offers a straightforward and efficient way to manage and optimize your web assets.

Key Concepts and Features of Parcel.js

1. **Zero Configuration**
   - **Automatic Setup:** Parcel aims to work out-of-the-box with minimal setup. Unlike other bundlers like Webpack, which often require extensive configuration files (webpack.config.js), Parcel automatically handles most of the configuration for you based on the files in your project.
   - **Sensible Defaults:** It provides sensible defaults for a wide range of features, reducing the need for manual configuration.

2. **Fast Performance**
   - **Bundling Speed:** Parcel is known for its speed, thanks to features like multi-core processing and caching. It performs incremental builds, which means it only rebuilds what has changed, leading to faster build times during development.
   - **Parallel Processing:** Utilizes parallel processing to speed up compilation and bundling processes.

3. **Automatic Asset Management**
   - **Asset Types:** Parcel can handle a variety of asset types, including JavaScript, CSS, HTML, images, fonts, and more. It automatically processes these assets and optimizes them for production.
   - **Code Splitting:** Supports automatic code splitting, which helps in optimizing the loading of JavaScript by splitting the code into smaller chunks.

4. **Built-in Support for Modern JavaScript and CSS**
    - **JavaScript Transpiling:** Parcel supports modern JavaScript features out-of-the-box, including ES modules, async/await, and dynamic imports. It uses Babel internally to transpile modern JavaScript code into a format compatible with older browsers.
    - **CSS Processing:** Parcel supports CSS out-of-the-box, including CSS modules and preprocessors like Sass and Less, with no additional configuration required.

5. **Built-in Development Server**
    - **Live Reloading:** Parcel includes a development server with live reloading capabilities. This means that any changes you make to your code are automatically reflected in the browser without needing a manual refresh.
    - **Hot Module Replacement:** It provides hot module replacement (HMR) during development, allowing you to replace modules without a full page reload.

6. **Plugin System**
    - **Extensibility:** While Parcel works with minimal configuration, it also supports a plugin system for extending its capabilities. You can use plugins to add additional features or modify Parcel's behavior to suit your needs.

7. **Simplified Dependency Management**
    - **Module Resolution:** Parcel understands how to resolve dependencies and modules using ES module syntax. It can automatically resolve and bundle dependencies without requiring complex configuration.

8. **Built-in Optimizations**
    - **Minification:** Parcel performs automatic minification and optimization of JavaScript, CSS, and HTML files for production builds.
    - **Tree Shaking:** It includes tree shaking to eliminate unused code, helping to reduce the size of the final bundle.

How Parcel Fits into the Ecosystem

Parcel is part of a broader ecosystem of tools designed to enhance web development workflows:

- **Bundlers:** Parcel is often compared with other bundlers like Webpack, Rollup, and Browserify. While Webpack is highly configurable and powerful, it often requires extensive setup. Parcel aims to provide a more straightforward alternative with automatic configuration.
- **Build Tools:** Parcel complements other build tools and task runners, such as npm scripts and Gulp. It handles the bundling and processing of assets, while task runners can be used for additional build tasks.
- **Frameworks:** Parcel works well with popular frameworks and libraries like React, Vue.js, and Angular. It simplifies the setup process for these frameworks by handling the bundling and optimization of their assets.

Example Use Cases

- **Single Page Applications (SPAs):** Parcel is suitable for SPAs, providing fast development builds and efficient production optimizations.
- **Static Sites:** It can be used for static site generators, handling the bundling of assets and serving the static files.
- **Prototypes and Small Projects:** Due to its zero-config nature, Parcel is ideal for prototypes and small projects where ease of use is more important than advanced customization.

Example : Simple Parcel Project

To get started with Parcel, you generally need to follow these steps:

1. **Install Parcel:** Install Parcel as a development dependency in your project using npm or yarn.

   ```
   npm install --save-dev parcel
   ```

2. **Create an Entry Point:** Create an HTML file (e.g., index.html) that serves as the entry point for your application.

   index.html

   ```
   <!DOCTYPE html>
   <html lang="en">
   <head>  <meta charset="UTF-8">
     <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>Parcel Example</title>
   </head>
   <body>
     <h1>Hello Students, Welcome to Parcel!</h1>
           <script src="./index.js"></script>
   </body>
   </html>
   ```

3. **Create Your JavaScript File:** Create a JavaScript file (e.g., index.js) that will be bundled by Parcel.

   index.js

   ```
   console.log('Hello, Parcel!');
   ```

4. **Add Scripts to package.json:** Add a script in your package.json to run Parcel.

   ```
   {
     "scripts": {
       "start": "parcel index.html"
     }
   }
   ```

5. **Run Parcel:** Start the development server.

   ```
   npm start
   ```

   Parcel will automatically bundle your files and serve them at http://localhost:1234 by default.

Redux Store

- The Redux store is a central piece of the Redux architecture.
- It holds the entire state of your application, manages state updates, and provides a way for components to subscribe to state changes.
- Understanding the Redux store and its use cases can help you effectively manage complex state in JavaScript applications, particularly those built with React.
- The Redux store is a powerful tool for managing state in JavaScript applications, especially as they grow in complexity.
- It centralizes state management, ensures predictable state transitions, and offers advanced features like time travel debugging and middleware support.
- By using Redux, developers can build more maintainable and scalable applications.

What is a Redux Store?

The Redux store is an object that:

- **Holds the application state:** The entire state tree of your app is stored in one place, making it easier to track and manage.
- **Allows state updates:** It allows you to dispatch actions that describe changes to the state.
- **Registers listeners:** Components or middleware can subscribe to changes in the store and react when the state updates.
- **Exposes state:** It provides a method (getState) to retrieve the current state of the application.

Core Components of a Redux Store

- **State Tree:** The entire state of the application is stored as a single object tree.
- **Actions:** Plain JavaScript objects that describe what happened. They are the only source of information for the store.
- **Reducers:** Functions that take the current state and an action, and return a new state. They define how the state changes in response to actions.

Example: Setting Up a Redux Store

1. Install redux using 'npm install redux'
2. declare script as module inside index.html using statement

   `<script src="index.js" type="module"></script>`

3. use the JavaScript file code

   index.js

```
import { createStore } from "redux";

const stateObj={        // changes are not required in initial state
   value:10,
   name:'Neha Rathore',
   age:25
}
function myReducer(state=stateObj, action){   //passing the initial state (stateObj)
   if(action.type==='change/incr')
   return {
      ...state, value:state.value+action.payload
   }
   else if(action.type==='change/decr')
      return {
         ...state, value:state.value-action.payload
      }
   return state;   // if we don't get the appropriate action to perform then return
the previous state
}

const myReduxStore=createStore(myReducer);  //create a redux store
console.log(myReduxStore) // to get the redux store structure & details
console.log(myReduxStore.getState()) //to get the initial state

myReduxStore.dispatch({type: 'change/incr', payload:5})   // calling reducer using
dispatch & passing the action
console.log(myReduxStore.getState()) //to get the updated state
```

4. run the project using 'npm start'.

Explanation : const myReduxStore = createStore(myReducer)

1. **createStore Function**
   - The createStore function is a core method provided by Redux to create a Redux store.
   - This store holds the entire state tree of your application, manages state updates, and allows components or middleware to subscribe to state changes.
   - You need to call createStore to initialize the store that will manage the state of your application.
   - Without a store, there would be no centralized way to handle state management in Redux.

2. **myReducer Argument**
   - The argument myReducer is a function (often referred to as a reducer) that defines how the state of the application changes in response to actions dispatched to the store.
   - Redux is based on the idea of predictable state changes. The reducer function is where you describe how each action affects the state.
   - When an action is dispatched, the store calls the reducer with the current state and the action, and the reducer returns the new state.

3. **const myReduxStore = ...**
   - The const keyword is used to declare a constant variable, myReduxStore, that will hold the instance of the Redux store created by createStore.
   - You store the result of createStore in myReduxStore so that you can access this Redux store throughout your application. Components, middleware, and other parts of your app can interact with the store using this constant.

Use Cases of Redux Store

- **Global State Management**
  - Problem: In large applications, managing shared state between multiple components can become challenging.
  - Solution: The Redux store serves as a single source of truth for the entire application. Components can access and update the global state through the store.
  - Example: An e-commerce application where the shopping cart state needs to be accessed and updated from various parts of the app (e.g., product pages, cart summary, checkout).

- **Predictable State Updates**
  - Problem: Applications with complex state interactions can become unpredictable.
  - Solution: By using pure functions (reducers) to manage state transitions, the Redux store ensures that state changes are predictable and follow strict rules.
  - Example: A task management app where adding, deleting, or updating tasks must follow specific business logic.

- **Time Travel Debugging**
  - Problem: Debugging state changes can be difficult, especially in dynamic applications.
  - Solution: The Redux store, when used with the Redux DevTools, allows developers to track state changes over time, undo/redo actions, and inspect previous states.
  - Example: A form wizard where the user can navigate back and forth between steps without losing progress.

- **Middleware Integration**
  - Problem: Applications often need to handle side effects, such as API calls, logging, or analytics, in a manageable way.
  - Solution: The Redux store supports middleware, which can intercept actions and perform side effects before they reach the reducers.
  - Example: A blog application where actions like posting a comment trigger an API call and dispatch additional actions based on the response.

- **Cross-Component Communication**
  - Problem: Passing state and callbacks down through multiple layers of components (prop drilling) can be cumbersome.
  - Solution: The Redux store allows components to subscribe to specific parts of the state, reducing the need for prop drilling.
  - Example: A social media app where the user profile state is needed in multiple, deeply nested components, such as profile settings, posts, and notifications.

- **State Persistence**
  - Problem: User data can be lost on page reloads, leading to a poor user experience.
  - Solution: The Redux store state can be persisted to local storage or a database and rehydrated when the app initializes.
  - Example: A note-taking app where the notes state is stored in the Redux store and persisted to local storage, allowing users to retain their notes even after refreshing the page.

store.subscribe()

- In Redux, the store.subscribe method is used to listen for changes to the Redux store's state.
- When the state of the store changes, the callback function provided to store.subscribe is executed.
- This allows you to perform side effects or update parts of your application in response to state changes.

Syntax

```
const unsubscribe = store.subscribe(listener);
```

where, listener is a function that will be called every time an action is dispatched and the state of the store changes.

Returns: A function (unsubscribe) that can be called to stop listening to store updates. This is useful for cleanup purposes, particularly in environments like React components.

How store.subscribe works

1. **Initial Subscription:** When you call store.subscribe, you pass it a callback function (listener). This function will be invoked every time the Redux store's state changes.

2. **State Change Detection:** Redux uses a shallow comparison to detect changes in the state. The listener function will be called whenever a new action is dispatched and the state produced by the reducer differs from the previous state.

3. **Unsubscribe:** The function returned by store.subscribe is used to unsubscribe from state updates. This is important in scenarios where you no longer need to listen to state changes, such as when a component is unmounted in a React application.

Example

index.js

```
import { createStore } from "redux";

const stateObj={        // changes are not required in initial state
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state=stateObj, action){   //passing the initial state (stateObj)
    if(action.type==='change/incr')
    return {
        ...state, value:state.value+action.payload
    }
    else if(action.type==='change/decr')
        return {
            ...state, value:state.value-action.payload
        }
    return state;
}

const myReduxStore=createStore(myReducer);
console.log(myReduxStore);

myReduxStore.subscribe(()=>{    //executed everytime when an action is
dispatched or state changes
    console.log(myReduxStore.getState())
})

myReduxStore.dispatch({type: 'change/incr', payload:5})
myReduxStore.dispatch({type: 'change/decr', payload:8})
```

__REDUX_DEVTOOLS_EXTENSION__()

- The __REDUX_DEVTOOLS_EXTENSION__() is a function that allows you to integrate your Redux store with the Redux DevTools extension in your browser.
- The __REDUX_DEVTOOLS_EXTENSION__() function is a way to connect your Redux store to the Redux DevTools, enabling powerful debugging features in your browser.
- This extension provides powerful tools for debugging your application's state changes in real time.
- It's a helpful tool during development to visualize and track the changes in your application's state.

What Does __REDUX_DEVTOOLS_EXTENSION__() Do?
When you call __REDUX_DEVTOOLS_EXTENSION__() in your Redux store setup, it connects your store to the Redux DevTools. This allows you to:

- Monitor every action that is dispatched.
- View the state before and after each action.
- Time travel (undo or redo state changes).
- Inspect the action payloads and state trees.
- Export and import state for debugging.

Syntax:

```
const store=createStore(myReducer, __REDUX_DEVTOOLS_EXTENSION__());
```

The above syntax checks whether the Redux DevTools extension is installed or not. If it is, then it calls the function to connect the store to the DevTools otherwise you will get an error.

or

```
const store=createStore(myReducer, window.__REDUX_DEVTOOLS_EXTENSION__
&& window.__REDUX_DEVTOOLS_EXTENSION__())
```

The above syntax is used to check if the __REDUX_DEVTOOLS_EXTENSION__ function exists on the window object (meaning the Redux DevTools extension is installed). If it exists, it calls the function to connect the store to the DevTools. If the extension is not available, it returns undefined, which is safely ignored by createStore.

Note: Before using above syntax make sure that you have the Redux DevTools browser extension installed or not. For Chrome, you can install it from https://chromewebstore.google.com/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en

Example

```
import { createStore } from "redux";

const stateObj={
    value:10,
    name:'Neha Rathore',
    age:25
}

function myReducer(state=stateObj, action){
    if(action.type==='change/incr')
    return {
        ...state, value:state.value+action.payload
    }
    else if(action.type==='change/decr')
        return {
            ...state, value:state.value-action.payload
        }
    return state;
}

const myReduxStore=createStore(myReducer,
__REDUX_DEVTOOLS_EXTENSION__()); // To integrate your Redux store with the
                                   Redux DevTools extension in your browser

console.log(myReduxStore);

myReduxStore.subscribe(()=>{    dispatched or state changes
    console.log(myReduxStore.getState())
})

myReduxStore.dispatch({type: 'change/incr', payload:5})
myReduxStore.dispatch({type: 'change/decr', payload:8})
```

Example: Connecting UI & JavaScript with Redux

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Parcel Example</title>
</head>
<body>
  <h1>Hello Students, Welcome to Parcel!</h1>
  <script src="index.js" type="module"></script>
  <h2>Value : <span class="updatedValue"></span></h2>
  <button class="incrButton">increment</button>
  <button class="decrButton">decrement</button>
</body>
</html>
```

index.js

```
import { createStore } from "redux";

const incrButton=document.querySelector('.incrButton');
const decrButton=document.querySelector('.decrButton');
const updatedValue=document.querySelector('.updatedValue');

const stateObj={        // changes are not required in initial state
   value:10,
   name:'Neha Rathore',
   age:25
}

function myReducer(state=stateObj, action){     //passing the initial state (stateObj)
   if(action.type==='change/incr')
   return {
      ...state, value:state.value+action.payload
   }
```

```
    else if(action.type==='change/decr')
        return {
            ...state, value:state.value-action.payload
        }
    return state;   // if we don't get the appropriate action to perform then return
the previous state
}

const myReduxStore=createStore(myReducer,
__REDUX_DEVTOOLS_EXTENSION__());
console.log(myReduxStore) // to get the redux store structure & details
myReduxStore.subscribe(()=>{          //executed everytime when an action is
dispatched or state changes
    console.log(myReduxStore.getState())
    updatedValue.innerText=myReduxStore.getState().value;
})

updatedValue.innerText=myReduxStore.getState().value;

incrButton.addEventListener('click', ()=>
    {
        myReduxStore.dispatch({type: 'change/incr', payload:5})
    })
decrButton.addEventListener('click', ()=>
    {
        myReduxStore.dispatch({type: 'change/decr', payload:5})
    })
```

Note: while running the above code make sure that Parcel should be properly installed and following script should be added in your package.json to run Parcel.

```
{
"scripts": {
"start": "parcel index.html" }
}
```