1. mapStateToProps() and mapDispatchToProps():

The `mapStateToProps()` and `mapDispatchToProps()` functions are used in conjunction with the `connect()` function provided by the React Redux library to connect a React component to the Redux store. They allow you to define how the component accesses the state and dispatches actions.

- mapStateToProps(state, ownProps): This function is used to map the Redux store state to the component's props. It receives two arguments: the current state of the Redux store (`state`) and the component's own props (`ownProps`). It should return an object that contains the props derived from the state.

Example:

```
const mapStateToProps = (state, ownProps) => {
  return {
    counter: state.counter,
    username: state.user.username,
    ...ownProps, // Include the original props of the component
  };
};
```

- mapDispatchToProps(dispatch, ownProps): This function is used to map action creators to the component's props, allowing the component to dispatch actions to the Redux store. It receives two arguments: the `dispatch` function and the component's own props (`ownProps`). It should return an object that contains the props that will dispatch actions.

Example:

```
import { incrementCounter, setUser } from './actions';

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    increment: () => dispatch(incrementCounter()),
    updateUser: (user) => dispatch(setUser(user)),
    ...ownProps, // Include the original props of the component
  };
};
```

2. Key differences between mapStateToProps() and mapDispatchToProps():

- mapStateToProps() is used to map the state from the Redux store to the component's props. It allows the component to access the state values and use them within the component.
- mapDispatchToProps() is used to map action creators to the component's props. It enables the component to dispatch actions to the Redux store, triggering state updates.

In summary, `mapStateToProps()` connects the state to the props, while `mapDispatchToProps()` connects the actions to the props.

3. Middleware:

Middleware in Redux is a third-party extension point that allows you to add custom logic between dispatching an action and the moment it reaches the reducers. It provides a way to modify, log, or handle actions in a centralised manner.

The most commonly used middleware in Redux is Redux Thunk.

To create a custom middleware in Redux, you need to define a function that follows a specific structure and apply it to the Redux store during store creation. Here's an example of how you can create a custom middleware:

```
const customMiddleware = (store) => (next) => (action) => {
  // Perform any logic before the action reaches the reducers
  console.log('Custom middleware triggered:', action);

  // Pass the action to the next middleware or the reducers
  const result = next(action);

  // Perform any logic after the action has been processed by the
reducers
  console.log('Next state:', store.getState());

  // Return the result of the action processing
  return result;
};
```

In the above example, `customMiddleware` is a function that takes the `store` as an argument and returns another function that takes `next` as an argument. This inner function, in turn, returns another function that takes `action` as an argument. This is known as the currying pattern and allows middleware composition.

Inside the `customMiddleware` function, you can add any custom logic before and after the action reaches the reducers. In this example, it logs the action before it's processed and the resulting state after the action has been dispatched.

To apply the custom middleware to the Redux store, you need to include it in the `applyMiddleware()` function when creating the store:

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from './reducers';

const store = createStore(rootReducer,
applyMiddleware(customMiddleware));
```

In the above code, the `applyMiddleware()` function is used to apply the `customMiddleware` to the store during store creation.

By including the custom middleware, any dispatched action will pass through the defined middleware logic before reaching the reducers. This allows you to add custom behavior, such as logging, modifying actions, or handling side effects, at a centralized place in your Redux application.

4. Redux Thunk:

Redux Thunk is a middleware for Redux that enables asynchronous actions to be dispatched. It allows action creators to return functions instead of plain action objects, which is useful for handling asynchronous operations such as API calls.

To use Redux Thunk, you need to install the `redux-thunk` package and apply it as middleware when creating the Redux store.

Example of an action creator using Redux Thunk:

```
import axios from 'axios';

const fetchUser = () => {
  return (dispatch) => {
    dispatch({ type: 'FETCH_USER_REQUEST' });

    axios.get('/api/user')
      .then((response) => {
        dispatch({ type: 'FETCH_USER_SUCCESS', payload: response.data });
      })
      .catch((error) => {
        dispatch({ type: 'FETCH_USER_FAILURE', payload: error.message });
      });
```

```
  };
};
```

In this example, the `fetchUser()` action creator returns a function instead of a plain action object. Within that function, asynchronous logic is performed, such as making an API request using Axios. The dispatched actions notify the Redux store about the different stages of the asynchronous operation (request, success, or failure).

5. How to use connect() from React Redux:

The `connect()` function from React Redux is used to connect a React component to the Redux store. It creates a new connected component that can access

the Redux store's state and dispatch actions.

To use `connect()`, you need to define `mapStateToProps()` and `mapDispatchToProps()` functions and pass them as arguments to `connect()`.

Example usage of `connect()`:

```
import { connect } from 'react-redux';

// Component definition

const mapStateToProps = (state) => {
  return {
    counter: state.counter,
    username: state.user.username,
  };
};

const mapDispatchToProps = (dispatch) => {
  return {
    increment: () => dispatch({ type: 'INCREMENT' }),
    decrement: () => dispatch({ type: 'DECREMENT' }),
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

In this example, the component `MyComponent` is connected to the Redux store using `connect()`. The `mapStateToProps()` function maps the state to the component's props, while `mapDispatchToProps()` maps the actions to the component's props. The resulting

connected component will have access to the mapped props and be able to dispatch actions.

6. Redux DevTools:

Redux DevTools is a browser extension that provides a deb
ugging interface for Redux applications. It allows you to inspect and time-travel through the state changes and action dispatches in your Redux store. It greatly facilitates the debugging process and understanding of how state changes over time.

To use Redux DevTools, you need to install the extension for your preferred browser. Then, when creating the Redux store, you can enable Redux DevTools using the `window.__REDUX_DEVTOOLS_EXTENSION__` extension.

Example of creating a Redux store with Redux DevTools enabled:

```
import { createStore } from 'redux';
import rootReducer from './reducers';

const store = createStore(
  rootReducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
);

export default store;
```

With Redux DevTools enabled, you can open the browser's developer tools and navigate to the Redux DevTools tab. There, you can inspect the dispatched actions, the current state, and navigate through the state history to debug and analyze your Redux application.

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)