React & Redux

Combining react app with Redux

Our goals

- Understand what is Redux and when and how to use it
- Redux architecture
- How to connect React components to the store
- How to trigger state change
- How to split our reducers
- How to connect middlewares to our redux store
- How do we deal with async action

Understanding Redux is Hard

- Working with React and Redux means we are going to work in a different architecture that is a bit hard to understand
- This is why we are going to try to simplify the need and how to achieve it with Redux

Communicating between react components

- In previous lesson we talked about different approaches to how communicate between components
- One of the challenges was communicating between 2 components that are far apart in the component tree and it will be hard work to transfer the communication by lifting it to the state of the parent (the common parent is really high up the tree)
- One way we solved that problem is by using service and rxjs for that communication
- Another way of solving that problem is by using Redux

share object - Redux State

- So we have React component A and component B which want to communicate with each other
- Let’s describe an object that will be shared to all components that request to look to that object, We will call this object redux state
  - Don’t get confused with component internal state, this state is shared among multiple components
- But wait if we have a shared object that everyone can change with no rules and order it could lead to a lot of bugs
- The redux state is also —- IMMUTABLE
  - WAT???

State protector - Redux Store

- The redux state is encapsulated and hidden and we cannot directly access that state and manipulate it
- The state is held in Redux store
- With the store you can do the following:
  - Read the current state
  - Subscribe for changes of the state
  - Manipulate the state
- We can also attach middlewares to the store to enhance it’s abilities

components communication ex

- Remember the problem we are trying to solve?
- Let’s create component A and component B that can grab data from the shared state in the store
- First we need to create the store with our state
- The state will look like this:
  - {message: ‘hello from redux’}
- component A and component B will print that message

Create the store - EX

- We need to first install redux with npm
  - npm install redux —save
- We create the store with the createStore method
  - reducer - will be explained later
  - initialState
  - middlewares - will be covered later
- Let’s create our store with the state we want to share with our components

react-redux context - EX

- One of the communication patterns between components we talked about is via context
- Redux use context to share the store and state along our component tree
- Recall that to use context we have to wrap our tree with a Provider of that context that redux created
- That provider is located in a separate package called react-redux which you will have to install
- Recall that a provider requires a value property to pass along the tree
  - The context and value property is aliased as <Provider store={theStoreWeCreated} > … </Provider>
- Let’s wrap the context around the root component

Reading the context - EX

- Recall that to read the context in our components we have to subscribe to it via providing the static property **contextType** which we will import from react-redux (ReactReduxContext)
  - or use the context Consumer with function components
- Create the 2 components and make them read the redux state from the context

Change the state

- The only way to change the state is by calling
  - store.dispatch(action)
- An action describes what happened in our app
- An action must have a name which we will refer to type
  - The name should be unique among all the actions (need to figure a team convention here)
- An action can optionally have extra data which we will refer to payload

Action example

- An action that happens when we query our rest server to grab the todo tasks and the server returns a response to our ajax request
  - {type: ‘[todoReducer] GET TODOS’, payload: [{title: …}, {description …}, …]}
- While we are querying the server and the response did not return we want to show a loading sign, we might have an action like this:
  - {type: ‘[todoReducer] TOGGLE LOADING’, payload: true}

Reducer

- The store will get our action and will have to decide how that action effects the state
- It will use a reducer to do so
- The reducer is a pure function that will get the current state and the action and will have to return the next state will be
  - Be carful, don’t forget the state is immutable
- The reducer function contains a switch on the action type
- The reducer is the first argument in the createStore method we used to create the redux store
- Redux will call the reducer when it is first initiated. It will be called with an init action and with state as the initial state we placed in the createStore method
- The initial state is optional in the createStore which means redux will send undefined to the reducer when redux is initiating
- This means we can initiate the state in the reducer

Reducer - EX

- In component B create a form with text input
- When submitting the form we will dispatch an action to the store with the message the user typed
- Create a reducer that will get the action and will change the message in the state
- Component A should display the change in the message as well

Context problem

- We don’t want every component connected to the redux store to have access to the store and the entire state
- We want to encapsulate the state and expose only the relevant part for each component
- Also every component should re render only when the part of the state he is connected to is changing and not re render every state change

connecting to part of the state - EX

- Let’s try and use the connect HOC to connect component A directly to the message from the state and not to the entire store using the context
- first argument of connect is a function refered to mapStateToProps
  - This function will be called with the state and needs to return an object that will be passed to the props

Action creator - EX

- We would usually want to wrap our actions so we can reuse them in different places in our app
- In our TOGGLE action when we use it in different places we might want to toggle with a different message
- How can we solve this?
- Create an Action creator for our toggle action

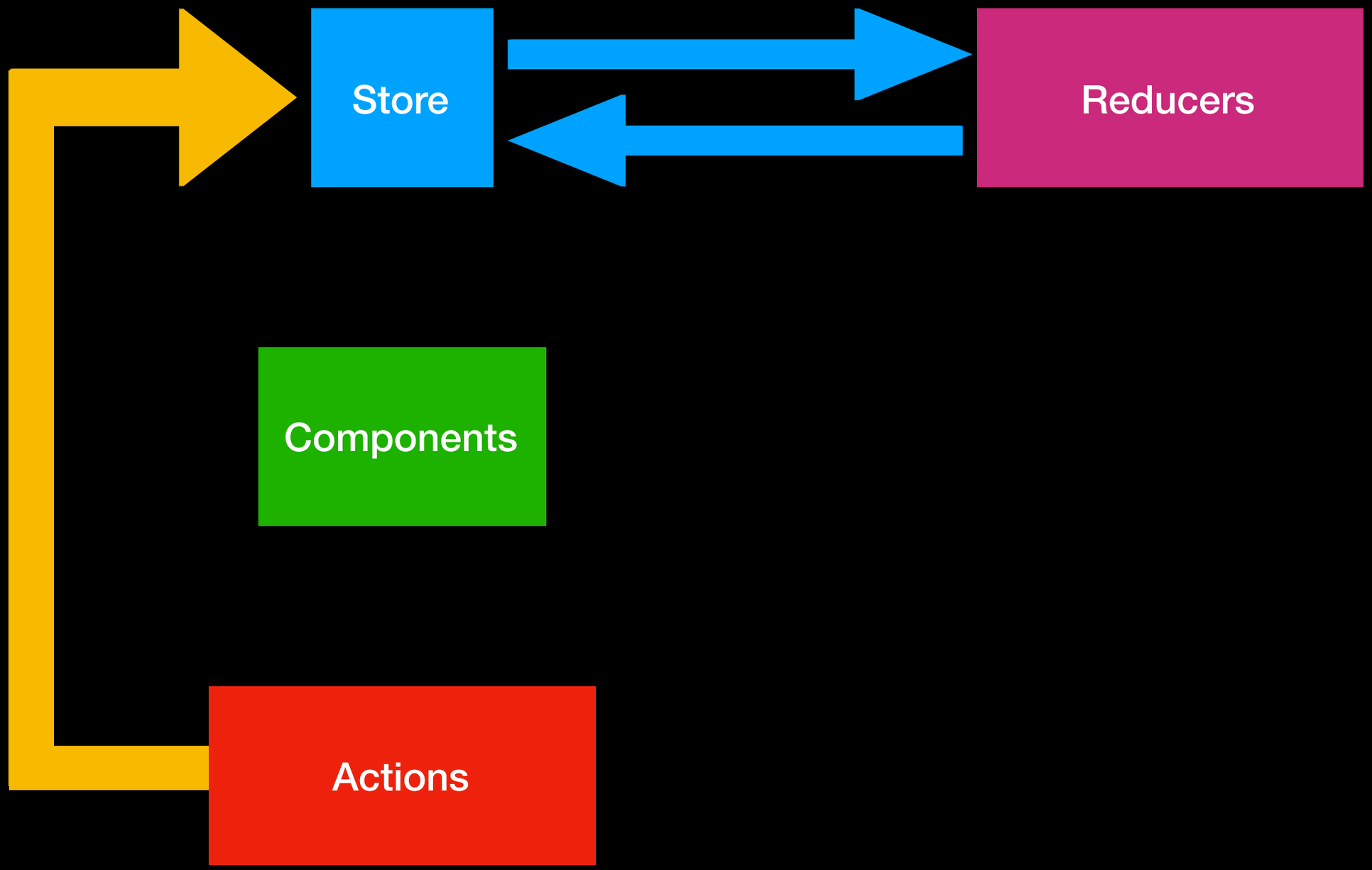
connecting dispatch - EX

- Component B can change the state by issuing an action
- In our current configuration component B can access the entire store to dispatch his action which means he will render upon every state change
- component B also has the power to dispatch any action he wants and it will be better to encapsulate and expose him only to the actions he needs
- connect can get a second argument which is a function called mapDispatchToProps
  - Will be called with the store.dispatch as argument and needs to return the actions which are mapped to the props
- use connect to expose component B to the action creator we created earlier

So what is Redux?

- Redux is a library to manage global application state
- Redux is framework agnostic and can be connected to react, angular, vue, etc.
- State change is managed in a predictable way using action and reducers
- What is a reducer, a reducer takes an initial value, an array and performs aggregation on the array to return a single value, the same doest redux, in what way?

Redux Architecture - Uni directional data flow



Smart / Dumb components and reusability

- When we learned about context one of the downsides we mentioned was us losing reusability of a component
- This is still the case when we connect a component to redux store
- We need to differentiate between smart components that can read the state and dispatch an action to the store and dumb components
- The smart components are often referred as containers
- For example we are implementing a Login form component and when we log in we want to place the user in the state
- The login form will be reused in our app and might be reused in many future app
- We need to consider how to build it in a reusable way

our state is currently like global var - problem

- Our app will grow larger
- As a result our state will grow
- So continuing to manage our state like we are doing means we will have a global variable containing all the redux state data to our components
- Managing our state is like managing the services in our app, we do not create a single service to serve all our components, rather we create a service to help us with the todo server query api, or our app settings api
- So our state should be splitted to sections

our state is currently like global var - problem

Bad State

```
{
    someSettingState: someValue,
    anotherSettingsState: Another Value
    someTodoState: …,
    anotherTodoState: …
}
```

Good State

```
{
    todo: {
        someTodoState: …,
        anotherTodoState: …
    },
    settings:  {
        someSettingsState: …,
        anotherSettingsState: …
    }
}
```

combineReducers - EX

- Using combineReducers we can split our state to sections where each section is controlled by a reducer
- The combine reducer will get an object with the state sections and the reducer that manage each section
- Each reducer will initialise his portion of the state
- Split our state to todo section and messages section which are managed by two reducers

enhancers

- The third argument of the createStore is store enhancers
- store enhancer is a function that will get a StoreCreator function like createStore and return a StoreCreator
- This let’s us create our own store or brind more power to the store
- The third argument of the create store is StoreEnhancer or a bunch of Store Enhancers
- You usually won’t create our own enhancer rather use some community ones

enhancer - applyMiddleware

- The first enhancer and most used one is applyMiddleware and is provided with Redux package
- This enhancer let's you connect middleware to your store
- A middleware will be called between the dispatch of the action and before sending that action to the reducer
- You can connect multiple middlewares
- When connecting the dispatch to our props using connect and mapDispatchToProps, we are not getting the original dispatch rather the one wrapped with all the middlewares
- With middlewares we can add additional functionality like
  - async actions
  - reporting
  - persistent state

applyMiddleware - ex

- As an example of the middleware usage, lets solve a problem we have with our actions, we often would like to dispatch async actions
- For example we want to query our todo server and place all the todo tasks in our state
- For async action we can use the redux-thunk middleware

redux-thunk

- As an example of the middleware usage, lets solve a problem we have with our actions, we often would like to dispatch async actions
- For example we want to query our todo server and place all the todo tasks in our state
- For async action we can use the redux-thunk middleware this middleware will examine the action and if the action is of type function, the middleware will run that action passing the store.dispatch
- We can also send an async function
- In that function we can query the server and when getting an answer we can dispatch the sync action

redux-thunk - EX

- apply the redux thunk middleware
- Create a new reducer called todoReducer our todo state will contain an array of tasks and we will add an action to set that array
- Create a redux-thunk action that will query the todo server
- create a react component that will dispatch that action and display the tasks from the state

Multiple enhancers

- If we want to add more than one enhancer to our store we will have to use compose
- compose will create a single enhancer from multiple enhancers
- We will now add an extra enhancer to our store

redux dev tools enhancer

- When working with redux we will need tools to debug our redux state
- We will want to know what is the current state and what action led to this state and maybe backtrack to the problem
- There is a browser extension that can connect to our store and display this information and to install it we need
  - Install the browser extension
  - Install redux-devtools-extension enhancer (devToolsEnhancer)

State best practices

- Normalize the state (not to many nesting)
- Remember that even nested objects needs to be cloned
- You can use immutable libraries to help you make sure you are keeping the state immutable
- Let’s examine some normalize recommendation:

State best practices - normalize

Bad

```
{
    users: [
        {doctors: [{name: '…', id: 1, nurses: [{…}, {…}]}, {…}, {… }]}
    ]
}
```

Good

```
{
    users: [
        {doctors: [1,2,3]}
    ],
    doctors: [
        {name: '…', id: 1}
    ],
    nurses: [
        {…}
    ]
}
```

Summary

- We now have to think about our private state and global app state
- When our component can solve a problem with their private state it is better to use the private state
- We often need to share state among multiple components and in this case Redux is a perfect fit