WEB3, Session 9

Redux

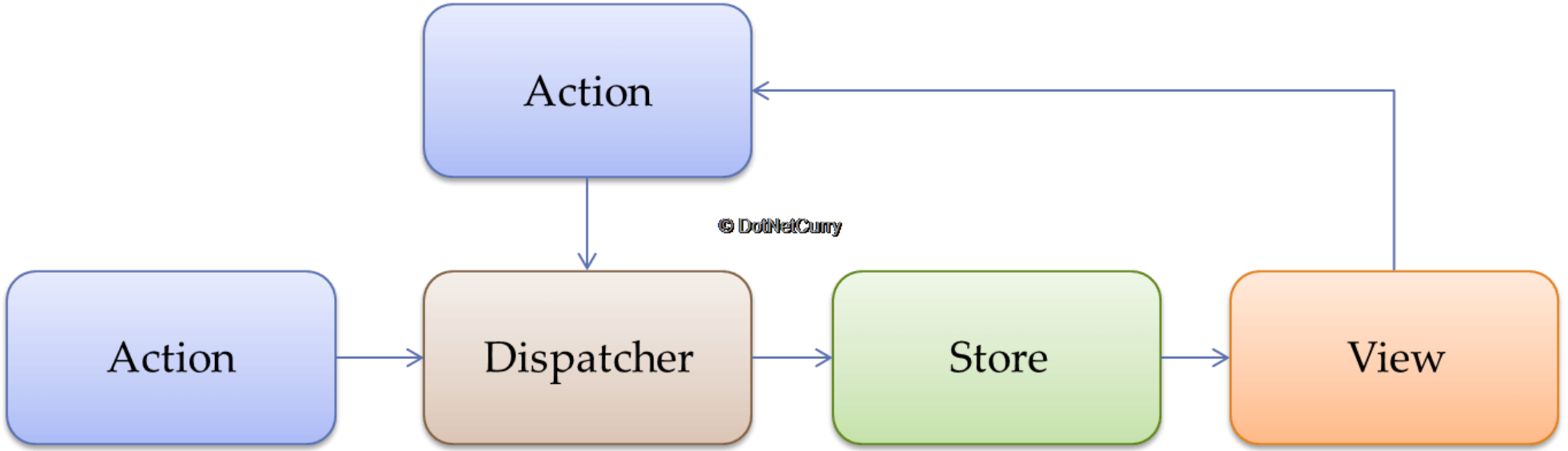
State Management

- Facebook's original motivation for React
- From many small interconnected state changes (MVC/MVVM)
- To one big independent state change (1-way data flow/flux pattern)
- React = Pure functions + non-functional hooks
- That means: State changes are *functionally computed*, but *non-functionally applied*

Local vs global state

- Local State
  - Defined in the components (useState hooks)
  - State is transferred to child components using properties
  - State is transferred to parent components using listeners
- Global State
  - The state is defined in a single place (Single Source of Truth)
  - This can be a global variable (or singleton pattern, if you prefer)

The original Flux pattern



The ReduceStore

What is a reducer?

- A pure function that creates the new state based on the old state and the action.
- Always creates a new copy of the state.
  - Optimize this in various ways (e.g. immutable.js).
  - (Free undos)
- state = reducer(state, action)
- Why "reducer"?
  - If you save all actions in an array (actions: Action[])
  - Current state is: actions.reduce(reducer, init_state)

The problem of Flux and state management

- The reducer function is unwieldy (and scales poorly)
- Some parts of the state are not relevant for the entire view
- There is no standard way of introducing side-effects (like server calls)
- Actions are poorly defined

Redux

- A way of creating a store with a reducer
  - Like what you would do yourself …
  - … expect more solid with more features
- Creating a store:
  - Create a reducer: (State, Action) => State
  - Make sure the 1st argument has a default parameter (the initial state) to get started.
  - Create the store with configureStore.
  - In theory, you don’t need anything else.
  - In practice, you want more.

Reducer

```
const initState = {
  sum: 0
}

function reduce({sum} = initState, action) {
  switch(action.type) {
    case 'add':
      const addend = action.payload
      return { sum: sum + addend}
    case 'reset':
      return initState
    default:
      return {sum}
  }
}
```

Store

```
export const store = configureStore({reducer: reduce})
```

Hooks

- Added to React to allow functional style same features as OO style
  - useState()
  - useEffect()
  - useContext()
  - useReducer()
- Allows side effects in functions
- Redux adds its own hooks

Provider

- The Redux equivalent of <Context.Provider>
- Provides store
  - Accessible through useSelector() and useDispatch()

Using a Provider

```
<Provider store = { store }>
  <h1>Adder</h1>
  <Adder/>
</Provider>
```

useSelector()

- Essentially a store.getState() singleton
- Allows to select the relevant part of the state

useDispatch()

- A store.dispatch singleton

Dispatch and Selector

```
const [addend, setAddend] = useState(0)
const sum = useSelector(s => s.sum)
const dispatcher = useDispatch()

const updateAddend = e => setAddend(parseInt(e.target.value))
const dispatchAdd = () => dispatcher({type: 'add', payload: addend})
const dispatchReset = () => dispatcher({type: 'reset'})
```

The JSX

```
return (<div>
  <div>Number to add:
    <input type='number' value={addend} onChange={updateAddend}/>
  </div>
  <div>Sum: {sum}</div>
  <div>
    <button onClick={dispatchAdd}>Add</button>
    <button onClick={dispatchReset}>Reset</button>
  </div>
</div>)
```

Slices

- Real states are complex
- Real reducers are big an unwieldy
- We can split them into *slices*
- A slice is a part of the state with its own reducer
- The state becomes a combination of the slices: { user: …, product: …, shopping: …, payment: …}
- The combined reducer can be built with combineReducer() or configureStore()

createSlice

- A method for each action instead of a big switch
- Create slice creates
  - the reduce function
  - An initial state
  - Action creators for each action

Creating a reducer

```
const lobbyReducers = {
  init(_: Game[], action: PayloadAction<Game[]>): Game[] {
    return action.payload
  },
  newGame(state: Game[], action: PayloadAction<Game>): Game[] {
    return [...state, action.payload]
  },
  joinGame(state: Game[], action: PayloadAction<Game>): Game[] {
    return state.filter(g => g.gameNumber !== action.payload.gameNumber)
  }
}
```

Creating a Slice

```
export const lobbySlice = createSlice({
    name: 'lobby',
    initialState: [] as Game[],
    reducers: lobbyReducers
})
```

Creating a store from slices

```
export const store = configureStore<State>({
  reducer: {game: gameSlice.reducer, lobby: lobbySlice.reducer }
})
```

Side effects

- All GUIs have some sort of side effects
  - Calling servers
  - Reacting to incoming events
  - Timers
- It's a common problem in 1-way flow where to handle side effects
  - View?
  - Action creation?
  - Reducer?
- Redux answer is to use a *Thunk*

Thunk

- A thunk is used as a pseudo-action with side effects
- A thunk is a function:
- async function thunk(dispatch, getState) {
  …
  }

- A thunk calls the server, then dispatches an action based on the result
- You can dispatch a thunk, if you are using ThunkMiddleware

Thunk dispatch

A thunk

```
async function concedeThunk(dispatch: Dispatch, getState: GetState) {
    const state = getState()
    const {mode, player, game: {gameNumber}} = state.game
    if (mode === 'playing') {
        const game = await api.concede(gameNumber, otherPlayer(player))
        dispatch(gameSlice.actions.setGame({player, game}))
    }
}
```

Dispatching a thunk (jsx/tsx)

```
<button onClick = {() => dispatch(concedeThunk)}>Concede game</button>
```

Parameterizing thunks

```
function makeMoveThunk(x: number, y: number): Thunk {
  return async function(dispatch: Dispatch, getState: GetState) {
    const state = getState()
    const {mode, game: { gameNumber }, player} = state.game
    if (mode === 'playing') {
      const payload = await api.createMove(gameNumber, {x, y, player})
      dispatch(gameSlice.actions.makeMove(payload))
    }
  }
}
```

Enhancers, Middleware, and Thunks

- Enhancer: is a function that wraps around createStore to do more.
- Middleware: An enhancer that only enhances dispatch.
- ThunkMiddleware: An enhancer that can handle thunks and actions
- configureStore() automatically adds ThunkMiddleware to the store.