

20.02.2023

Redux 2

Valentins Jegorovs

- Homework update from me
- Homework questions?

Topics for today:

- Multiple sub-store objects
- Middlewares
- redux-thunk & side-effects
- Action creators & typings
- reselect memoized & composable selectors

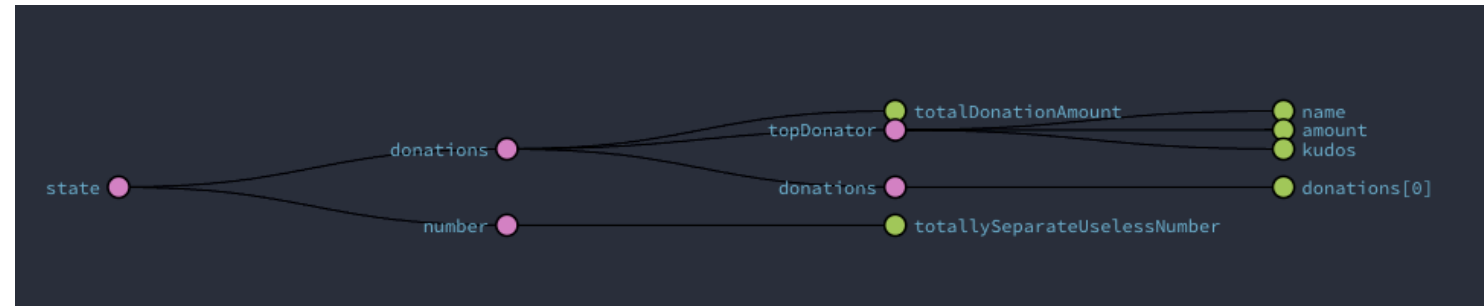
Growing your State

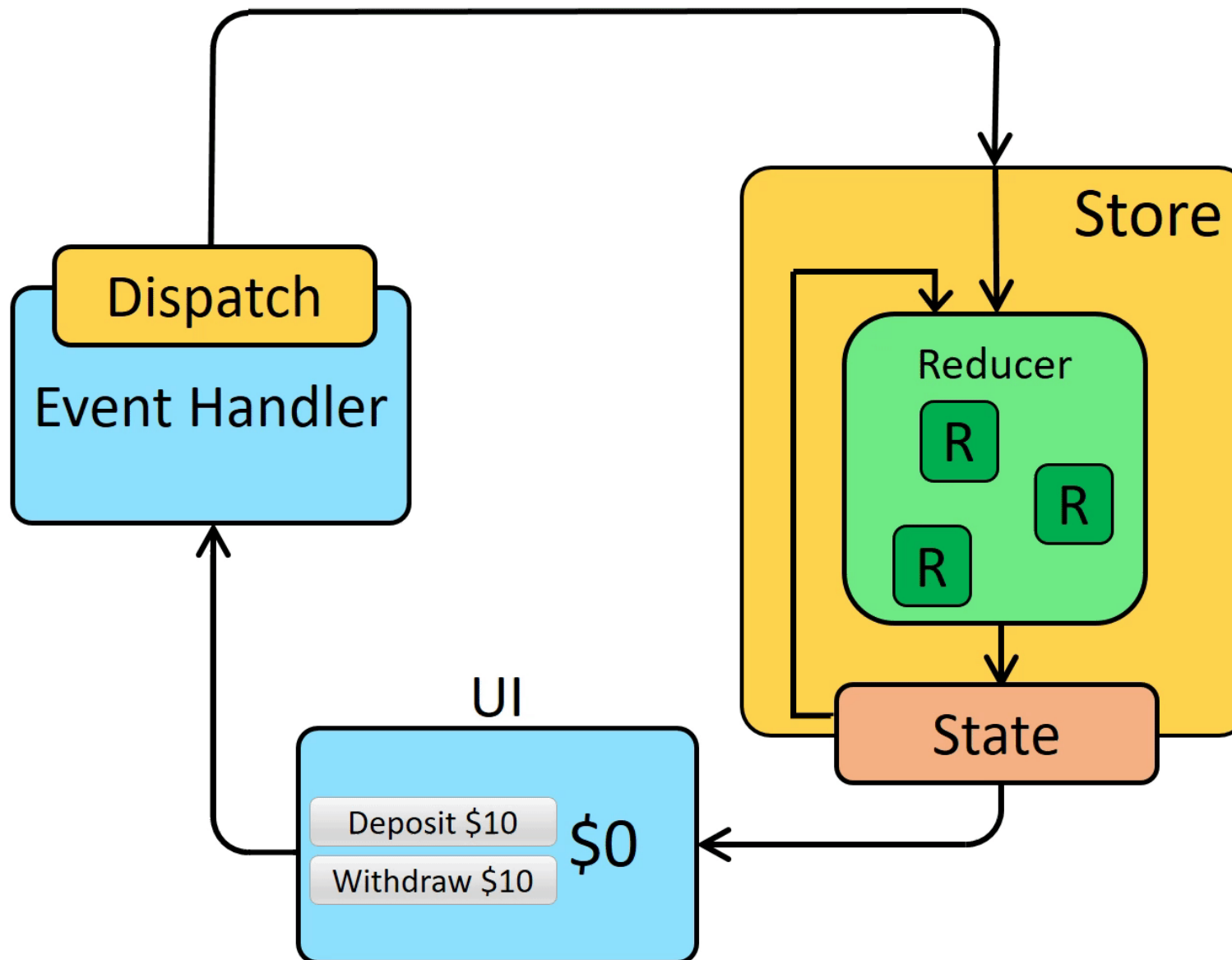
As state grows, so do the respective reducers

Keep them maintainable and separate concerns - use *combineReducers()*

```

const rootReducer = combineReducers({
  donations: donationsReducer,
  number: numbersReducer,
})
  
```





Middleware

By itself, a Redux store doesn't know anything about async logic. It only knows how to synchronously dispatch actions, update the state by calling the root reducer function, and notify the UI that something has changed.

Redux middleware can do ***anything***

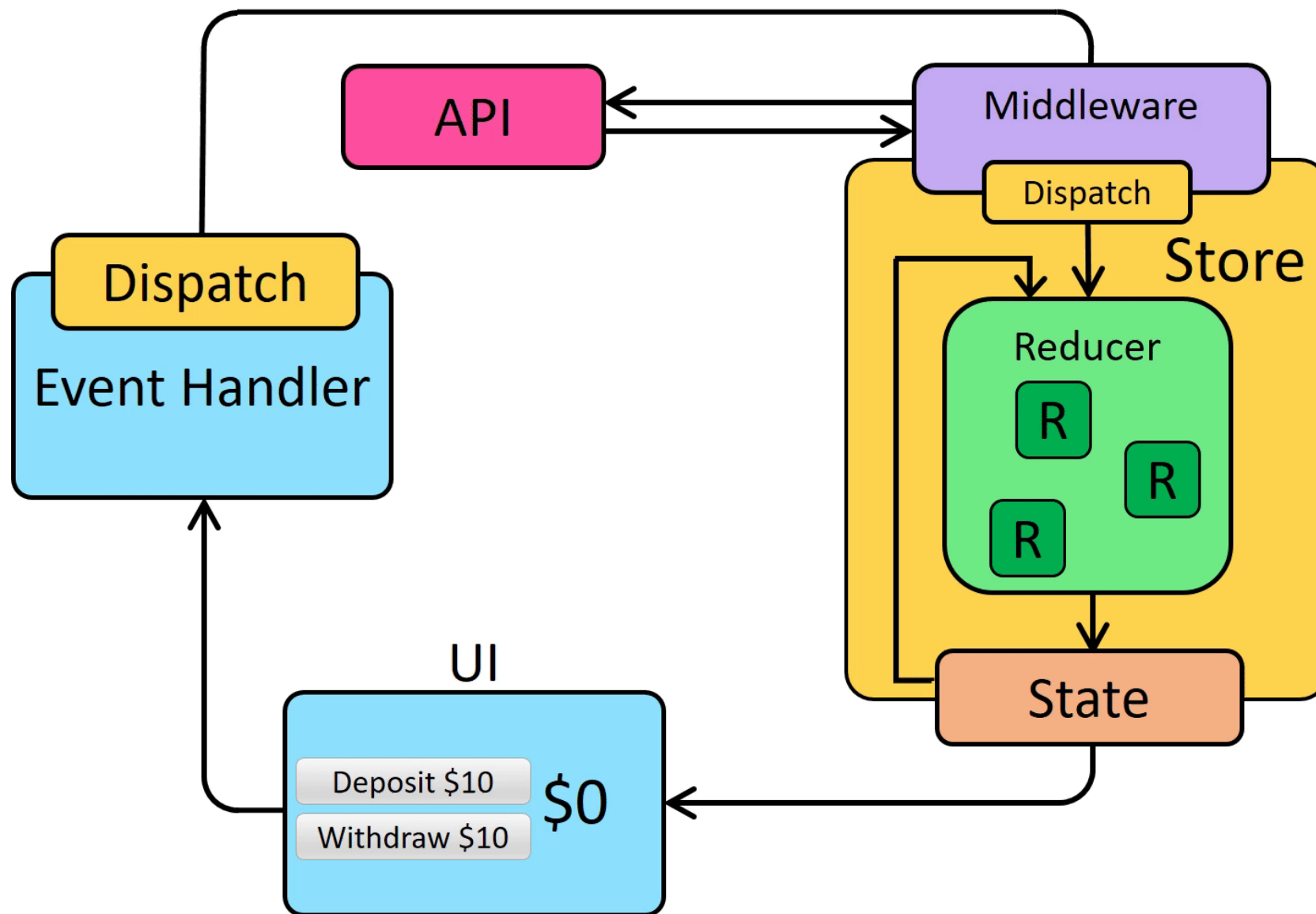
Provides a third-party extension point between dispatching an action, and the moment it reaches the reducer

```
({ getState, dispatch }) => next => action.
```

```
export const exampleMiddleware: Middleware<{}, RootState> =
  (store: MiddlewareAPI<Dispatch, RootState>) => (next) => (action) => {
    // You logic here

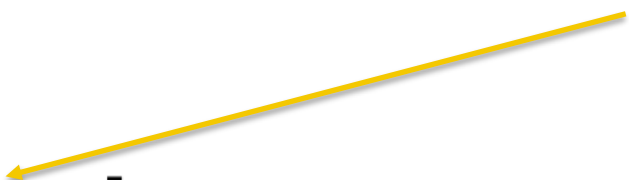
    // In the end, pass the action to other middlewares
    // next in line. Or in the end - reducers
    return next(action);
  }
```

Now, possible to pass something that *isn't* a plain action object to dispatch, as long as a middleware intercepts that value and doesn't let it reach the reducers.



Middleware creation

⚠ Middleware order matters!



```
const middlewares = [  
  delayedActionMiddleware,  
  logger,  
  thunk.withExtraArgument({extraArgument: {test: 5763}})  
]  
  
const store = createStore(  
  rootReducer,  
  applyMiddleware(...middlewares)  
)
```

Source: <https://redux.js.org/tutorials/essentials/part-2-app-structure#writing-async-logic-with-thunks>

redux-thunk

“**thunks**” are a pattern of writing functions with logic inside that can interact with a Redux store's dispatch and getState methods.

Requires the *redux-thunk* middleware.

Thunks are a standard approach for writing async logic in Redux apps ... they can be used for a variety of tasks, and can contain both synchronous and asynchronous logic.

Allows us to write additional Redux-related logic separate from a UI layer.

Use cases:

- ✓ Moving complex logic out of components
- ✓ Making async requests or other async logic
- ✓ Writing logic that needs to dispatch multiple actions in a row or over time
- ✓ Writing logic that needs access to getState to make decisions or include other state values in an action

```
const thunkFunction: Thunk =
  (dispatch: ThunkDispatch<any, any, any>, getState) => {
    // logic here that can dispatch actions or read state
  }
```

```

export type DonationsActionType = {
  type: "donate",
  payload: { name: string, amount: number }
} | {
  type: "addThanks",
  payload: { recipient: string }
};

export function donationsReducer(
  state: DonationsState = initialState,
  action: DonationsActionType
): DonationsState {
  switch (action.type) {
    case "donate": ...
    case "addThanks": ...
  }

  return state;
}

dispatch( action: {
  type: "addThanks",
  payload: {
    recipient: name,
  }
}
))
    
```

```

export type NumberActionType = ReturnType<
  typeof incrementUselessNumber
  // | typeof decrementUselessNumber
  // | typeof someOtherActionCreatorFunction
>;

export const INCREMENT_USELESS_NUMBER = "numbers/INCREMENT_USELESS_NUMBER";

export function incrementUselessNumber() {
  return {
    type: INCREMENT_USELESS_NUMBER,
  } as const;
}

export function numbersReducer(
  state: NumbersState = initialState,
  action: NumberActionType
): NumbersState {
  switch (action.type) {
    case INCREMENT_USELESS_NUMBER: ...
  }

  return state;
}

dispatch(incrementUselessNumber());
    
```

reselect

Selectors in general -
encapsulation and reusability for
your State shape

Redux store subscribe() causes all
selectors to be re-run on **any**
update

Reselect allows memoizing the
result of selector functions and re-
runs them only when the required
part of state changes

- Selectors can compute derived data, allowing Redux to store the minimal possible state.
- Selectors are efficient. A selector is not recomputed unless one of its arguments changes.
- Selectors are composable. They can be used as input to other selectors.

```
const exampleComposedSelector = createSelector(
  selectorFunction1,
  selectorFunction2,
  selectorFunctionN,
  (s1, s2, sN) => {
    // Derive your data
    return s1 + s2 + sN;
  }
)
```

Link: <https://github.com/reduxjs/reselect>

More reading: <https://blog.isquaredsoftware.com/2017/12/idiomatic-redux-using-reselect-selectors/>

