An Introduction

@manoj_nama

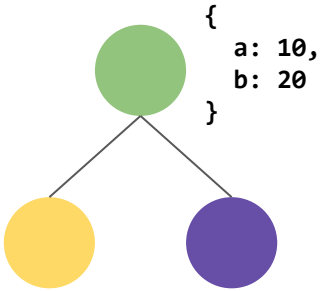
Current Scenario

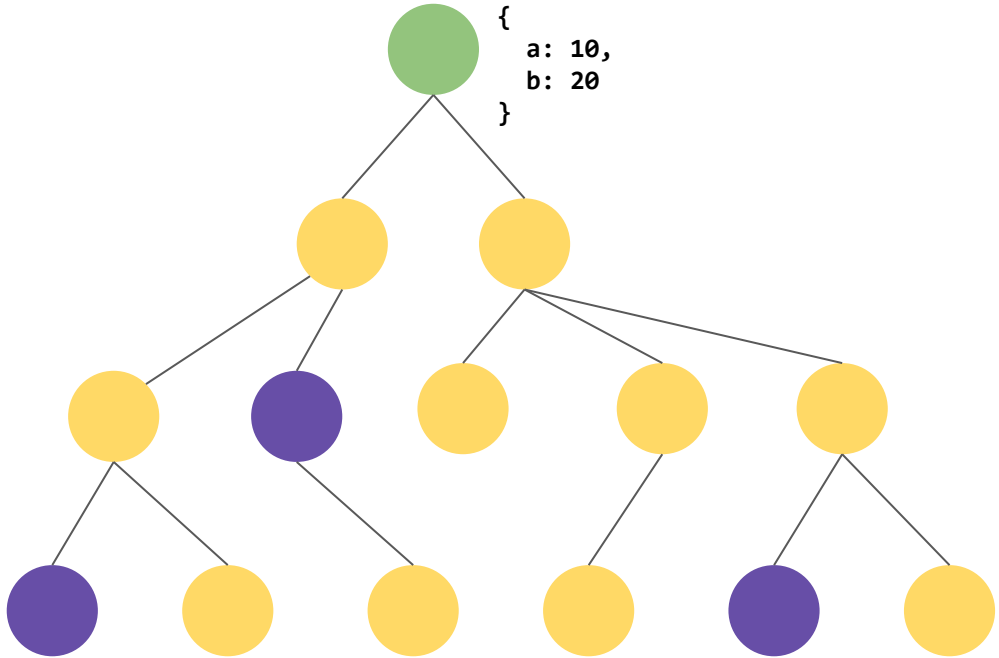
- Storing Data?
- Change Listeners?
- Separate parts of an App? How to decide where data should reside?
- Debugging?

What is the Problem?

How do you **pass** *data* between your components in a *Large* React App?

What is the Problem?

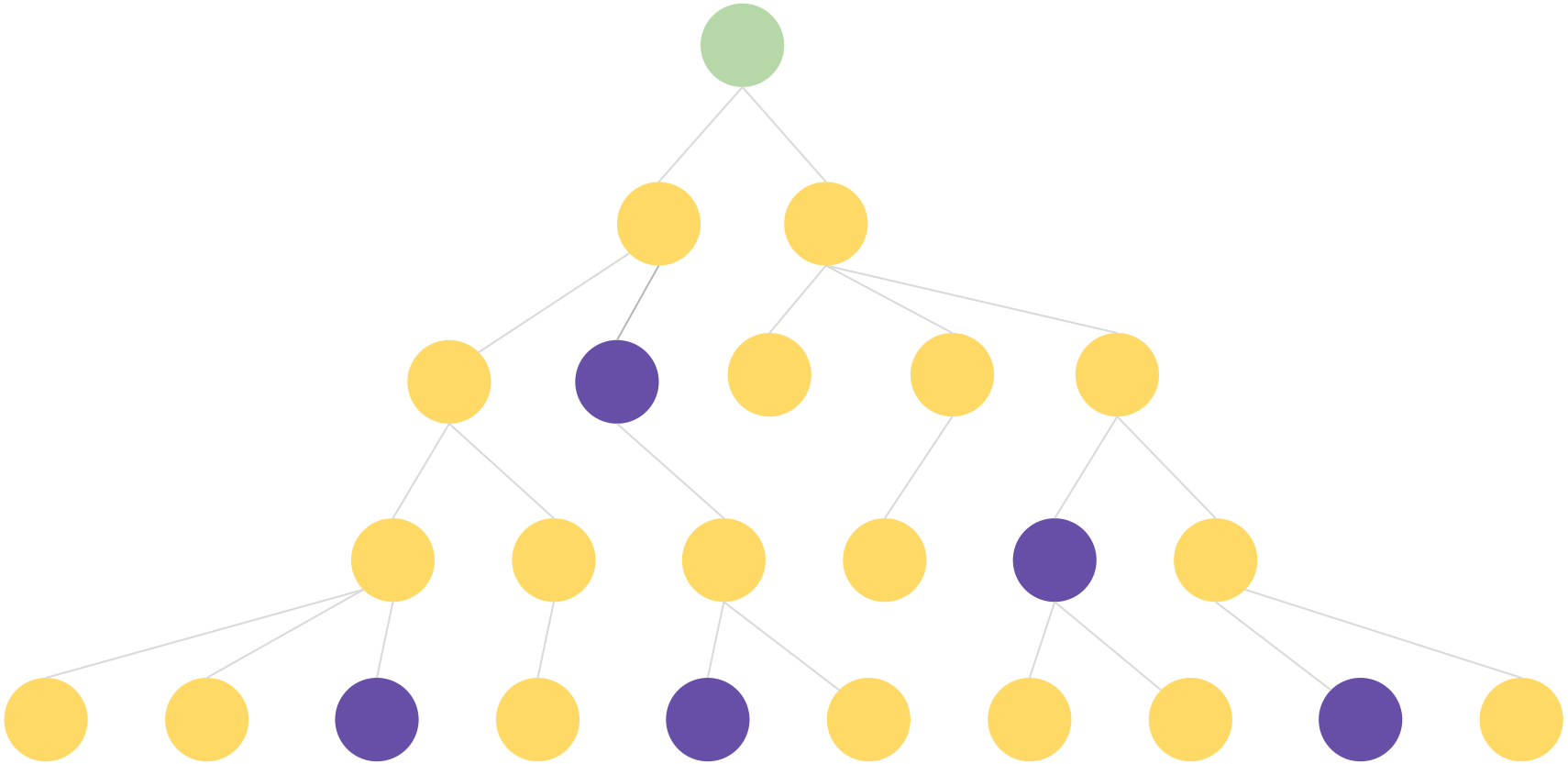




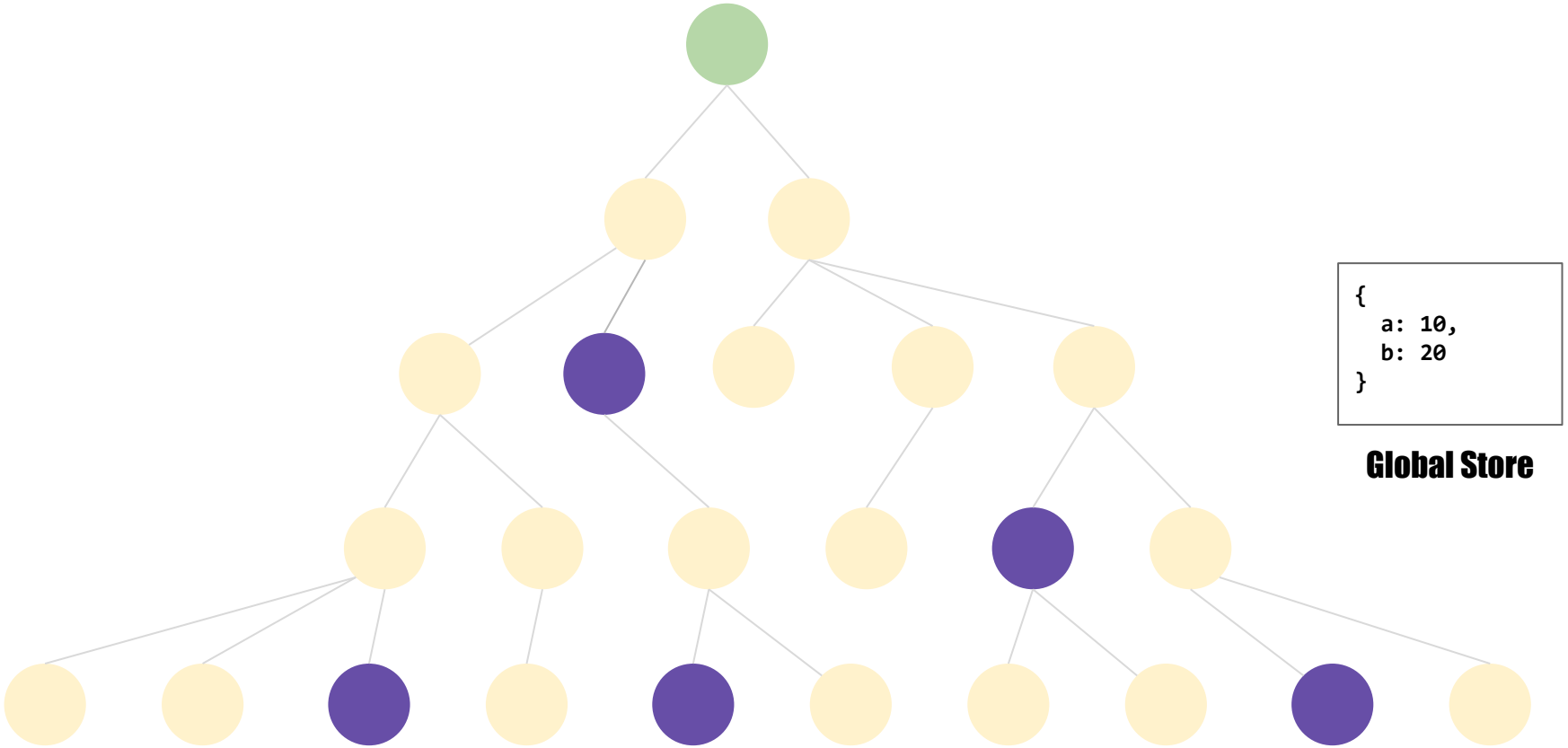
What is the Problem?

Imagine a **100** *levels for this*?

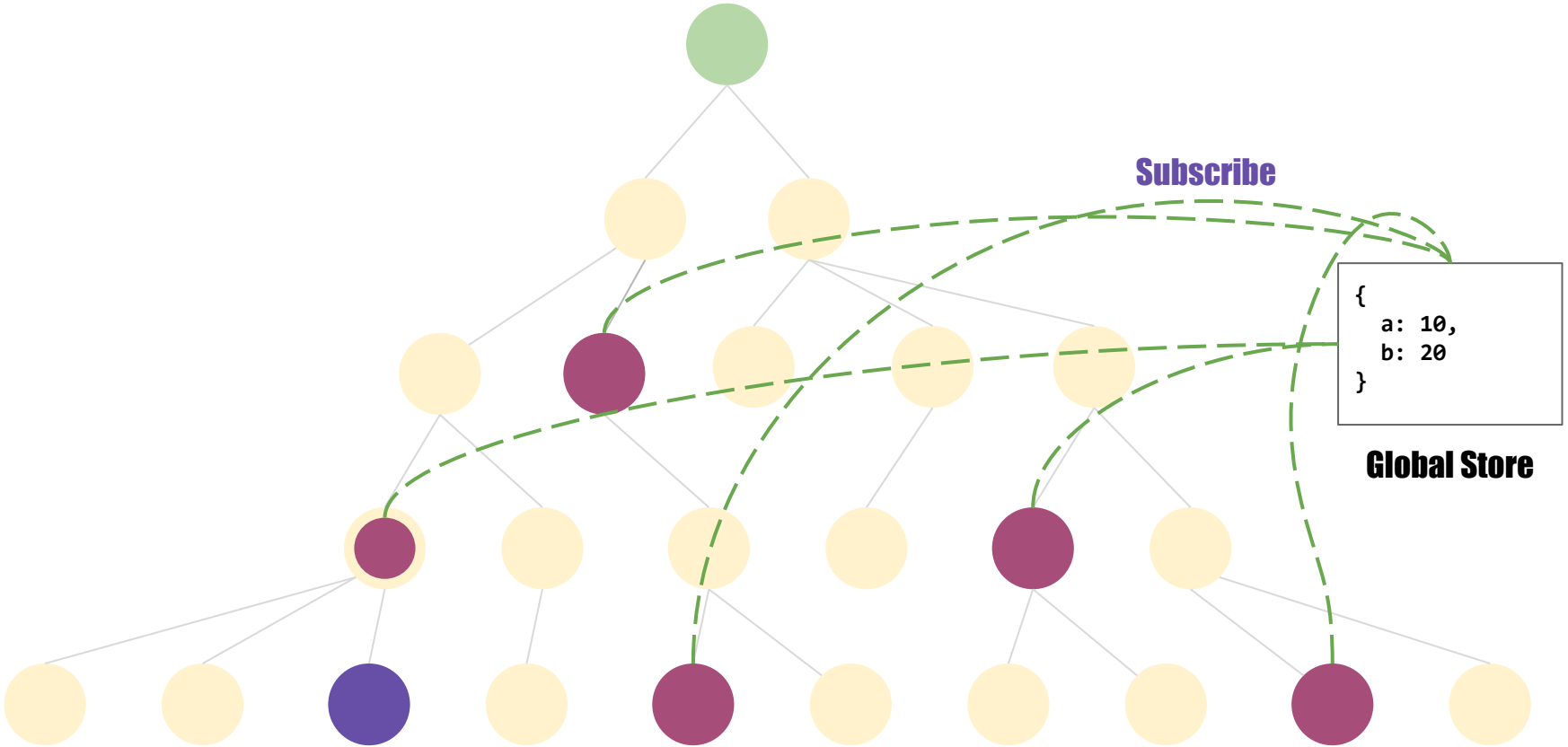
How Redux solves this?



How Redux solves this?



How Redux solves this?



What is Redux?

It is a **predictable** *state* **container** for JavaScript apps.

- Official Docs

What is Redux?

What the **hell** does that *mean*?

What is Redux?

It is a **predictable** *state* **container** for JavaScript apps.

- Official Docs

What is Redux?

Separate **Business** and **Presentation**

Logic.

React for Views, *Redux* for Data

Understanding Redux

- It is a Glorified Event-Emitter
- It fires *events* when the **store** has changed
- Requires us to keep our data flow **Uni-directional**
- Can be used with Any of the front-end languages, including *Angular*, *Backbone*, *React* and many *more*.
- **Redux** does not have *anything to do* with **React**

Components of Redux

There are **3** basic/essential ***components*** to keep in mind when using Redux

- Store
- Reducers
- Actions

Store

A **Global** Object, Holds your **entire** *application* state. To *update* any part of app, **change** the store.

Store

```
{
    loading: true,
    items: [{...}, {...}],
    user: { email: "...", name: "..." },
    products: [
        { id: 1, ... },
        { id: 2, ... },
        { id: 3, ... },
    ]
}
// normalized state?
```

Action

A **plain** JavaScript Object, specifies what to do. Fire an **action** when the *store* needs to be *updated*.

Action

```
{
    type: "FETCH_USERS",  // required
    data: {
        offset: 50,
        limit: 10,
        query: "mike"
    }
}
```

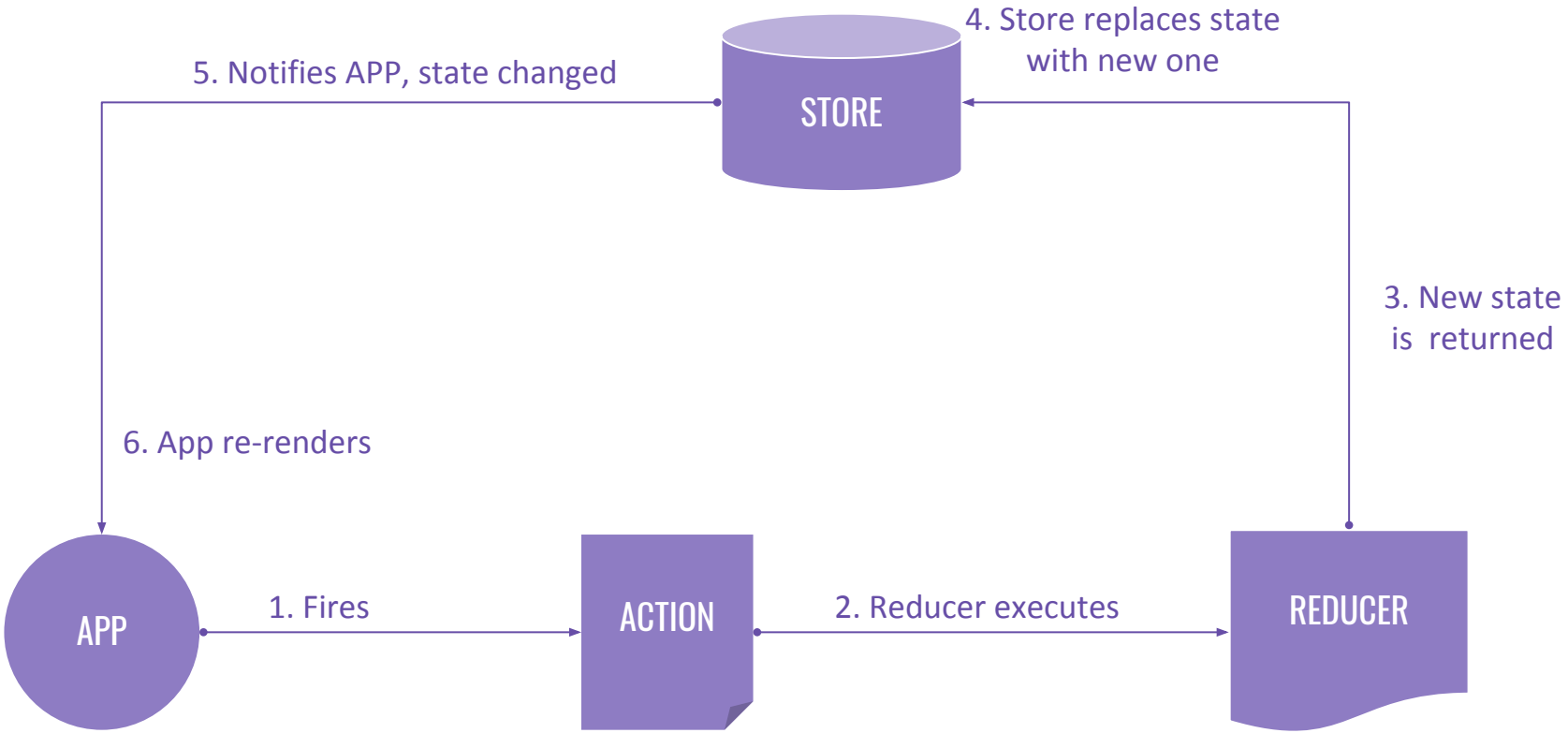
Reducer

A **pure** function, takes an **action** and **state**, *returns* a **new** state.

Reducer

```
const reducer = (state, action) => {
    switch(action.type) {
        case "FETCH_USERS_COMPLETE": {
            return {
                users: [...state.users, ...action.users],
                loading: false
            };
        }
        case "...": { ... },
        default: { return state; }
    }
}
```

Data Flow


4. Store replaces state with new one
5. Notifies APP, state changed
STORE
3. New state is returned
6. App re-renders
APP
1. Fires
ACTION
2. Reducer executes
REDUCER

Hands-On

Let's see all of this in **Action**.

Multiple Reducers

- We can have multiple reducers which can act on the state specified to that reducer
- All the reducers can **act** on any *Action*, but they have **access** to manipulate only their state.
- We can add multiple reducers by using `combineReducers` utility from **redux**
- `import { combineReducers } from 'redux'`
- We specify the state tree we want with each of the reducer

Multiple Reducers

```
const reducers = combineReducers({
    users: userReducer,
    products: productsReducer
});
// each reducer will have a state tree as its initial state
// userReducer is not aware about existence of products
// reducer and vice-versa

// create the store with combined reducers
const store = createStore(reducers);
```

Middlewares

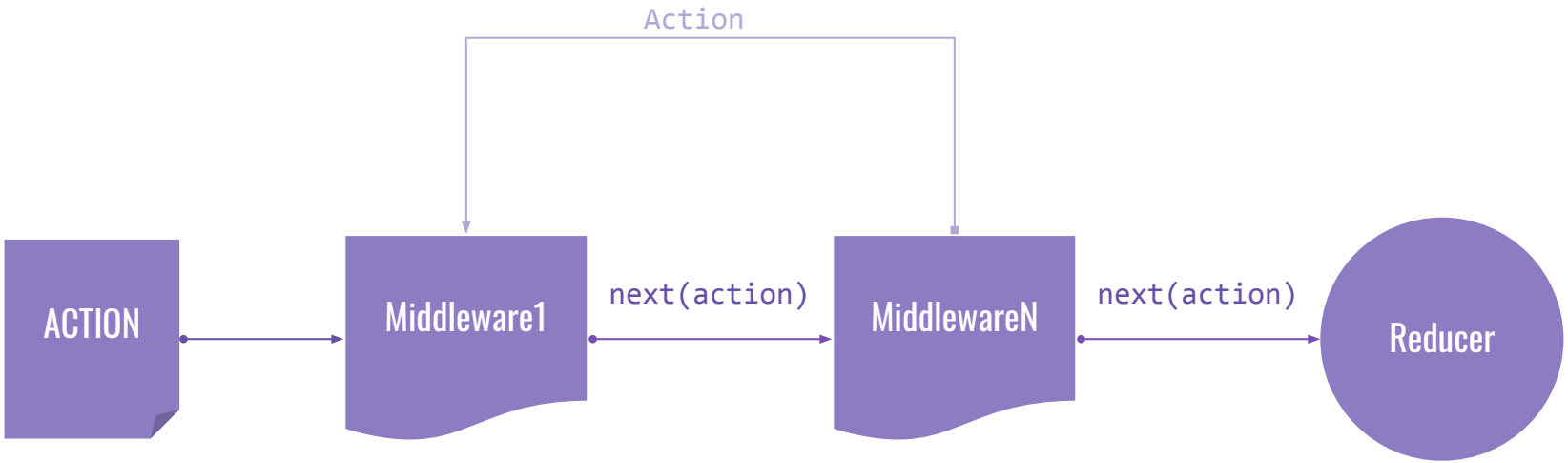
- Are simple functions intercepting the actions **before** reaching to the reducers
- They can **manipulate** actions and can **block** any actions to be forwarded to the reducer
- Middleware have the power to do async actions, i.e API calls etc. Since they have control over actions to be forwarded
- These are a chain of **thunks**, i.e functions returning functions
- `import { applyMiddleware } from 'redux'`

Middlewares

```
const applyMiddleware = (store) => (next) => (action) {
    // manipulate action here
    ...
    // forward action to next middleware(s) or to a reducer
    next(action)
};

To use middlewares, use applyMiddleware utility from redux
const middlewares = applyMiddleware(...list of middlewares)
const store = createStore(reducers, middlewares);
```

How redux-thunk works?



Async Actions

- Actions are synchronous, i.e they happen instantly
- For things like API calls or network requests, these need to be used in a different way
- We use `redux-thunk` to implement *async* actions
- These are multiple actions fired over a period of a Network request
- I.e Request fired, Request succeeded, Request failed
- `redux-thunk` provides a middleware for this called `thunkMiddleware`

Async Actions

```
const asyncAction = () => {
    return (dispatch) => { // this is store's dispatch method
        dispatch(apiCallStarted()); // call started
        fetch('http://rest.learncode.academy/api/ttn/users')
            .then(response => response.json())
            .then(data => {
                dispatch(apiCallSuccess(data));  // success
            })
            .catch(err => {
                dispatch(apiCallFailed(err));    // failure
            });
    }
};
```

Hands-On

Demo.

React+Redux

Combining React with Redux

React+Redux

- Two things we need to connect React with Redux
- *Wrapping* our App with a **HOC** called `Provider`
- Provider *listens* for store changes and **notifies** the App (i.e **`subscribe`**)
- Use the `connect` utility to connect store with the components
- It controls **what** & **how** the component will receive the state data.
- Any component attached with connect utility has access to the store's state

Provider HOC

```
import { Provider } from 'react-redux';
import { render } from 'react-dom';
import App from './our/root/App';
import store from './our/create.store';

render(
    <Provider store={store}>
        <App />
    </Provider>,
    rootEl
);
```

Connect

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

class App extends Component {
    ...
}

const mapStateToProps = state => state;

const AppContainer = connect(mapStateToProps)(App);
export default AppContainer
```

Container Components

- Container component are the one which have access to the `Redux` state, or Are having any business logic
- They only pass the required data to child components
- Container components are the ones with `connect` applied
- We need a strategy for how many levels we need to pass data. After a couple of levels, introduce a `connect`ed component

Presentational Components

- Components which are there **only** to *present* data to the user.
- They **Don't** have any *business logic*, or any *complex computations*
- They get the data as *via* **props** and just display them.
- They are simple components, they may also be just simple functions

Functional components

```
const Row = ({data, from, props}) => {
    return (
        <p>This is a row. {data}</p>
    )
}
```

Connect Functions

- Connect function takes some functions as arguments
- These functions determine how we want to subscribe to store changes
- There are multiple scenarios where these come in handy
- `mapStateToProps`
- `mapDispatchToProps`
- Existence of each of these functions will tell what the component will receive

Connect

```
// Don’t Subscribe to store changes
// inject only dispatch to App component

const Root = connect()(App);
```

Connect

```
// Subscribe to store changes
// inject entire state and dispatch to App
const mapStateToProps = (state) => {
    return state;
}

const Root = connect(mapStateToProps)(App);
```

Connect

```
// Subscribe to store changes
// inject users and dispatch to App
const mapStateToProps = (state) => {
    return state.users;
}

const Root = connect(mapStateToProps)(App);
```

Connect

```
// Don't Subscribe to store changes
// inject fetchUsers action creator to App
const mapDispatchToProps = (dispatch) => {
    return {getUser: (id) => dispatch(fetchUsers(id))};
}

const Root = connect(null, mapDispatchToProps)(App);
```

Connect

```
    // Subscribe to store changes
    // inject users and dispatch to App
    // inject fetchUsers action creator to App
    const mapStateToProps = (state) => {
        return state.users;
    }
    const mapDispatchToProps = (dispatch) => {
        return {getUser: (id) => dispatch(fetchUsers(id))};
    }

const Root = connect(mapStateToProps, mapDispatchToProps)(App);
```