



An Introduction

Current Scenario

- Passing Data?
- Storing Data?
- Global scope limits?
- Change Listeners?
- Separate parts of an App?
- Debugging?
- Controlling re-render

What is Redux?

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

- *Official Docs*

What is Redux?

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

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"  
  }  
}
```

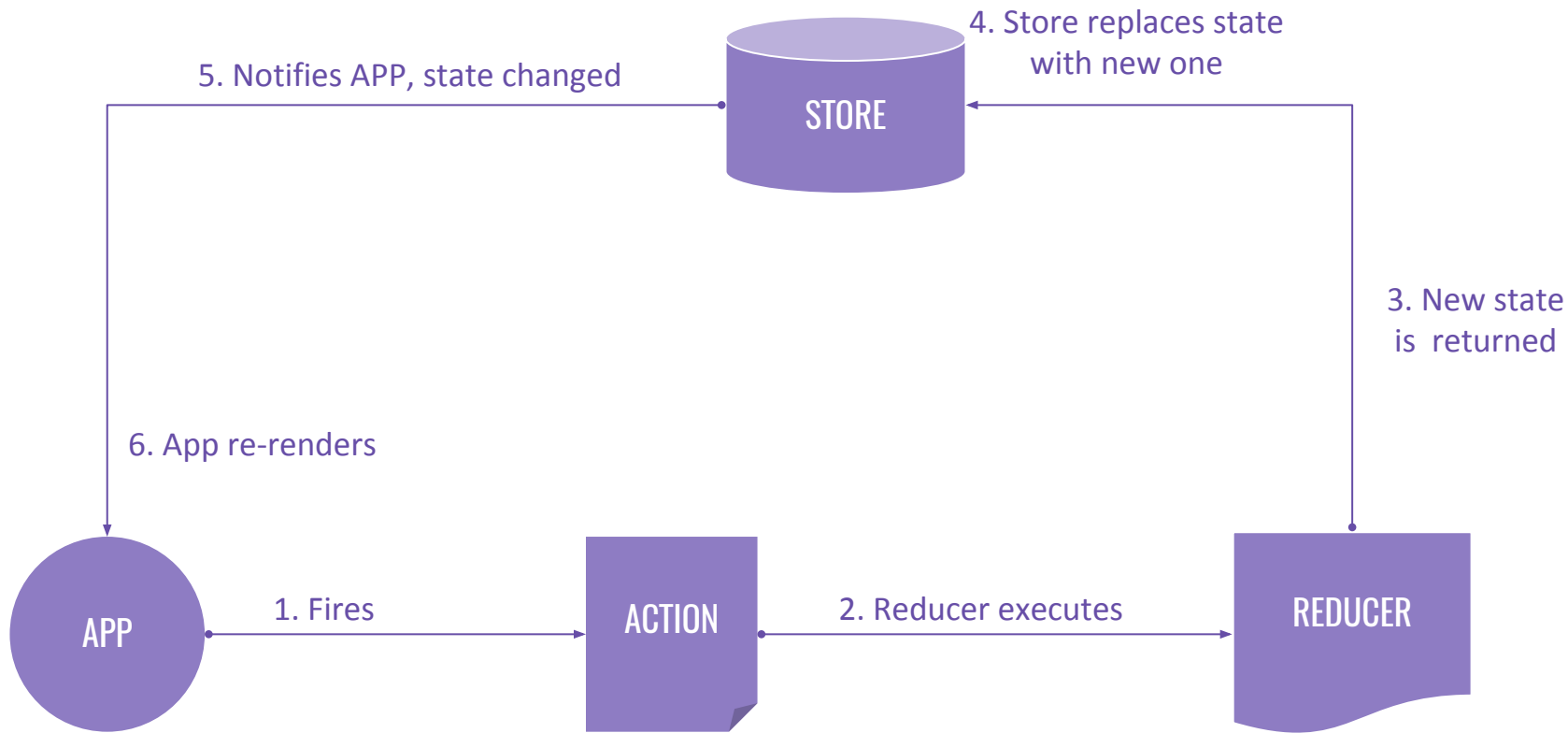
Reducer

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

Reducer

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

Data Flow



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 aMiddleware = (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 middleware = applyMiddleware(...list of middlewares)  
const store = createStore(reducers, middleware);
```

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.

Directory Structure

Organizing your Code.

Directory Structure

- Type Based
- Feature Based

Type Based

```
/src
  /actions
    - (async.actions.js, actions.js)
  /reducers
    - (x.reducer.js, y.reducer.js, ...others)
  /store
    - (create.store.js)
  /components
    - (Header.js, Footer.js, Rating.js)
  /containers
    - (Home.js, About.js)
  /config
    - (constants.js, global.config.js)
App.js
```


Feature Based

```
/src
|  /feature1
|  |   - feature1.actions.js      // Actions/Async Actions
|  |   - feature1.reducer.js      // Reducer
|  |   - feature1.constants.js    // Specific constants
|  |   - feature1.scss            // Styles
|  |   - Feature1.js              // Component
|  /store
|  |   - (create.store.js)
|  /config
|  |   - (constants.js, global.config.js)
App.js
```

Hands-On

Let's Organize our code.

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 React from 'react';
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 components are the ones 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 **connected** 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 sometimes called **PURE** components, as they are just simple functions
- They **Don't** have any *lifecycle* events and thus are *Faster* to process.

PURE components

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