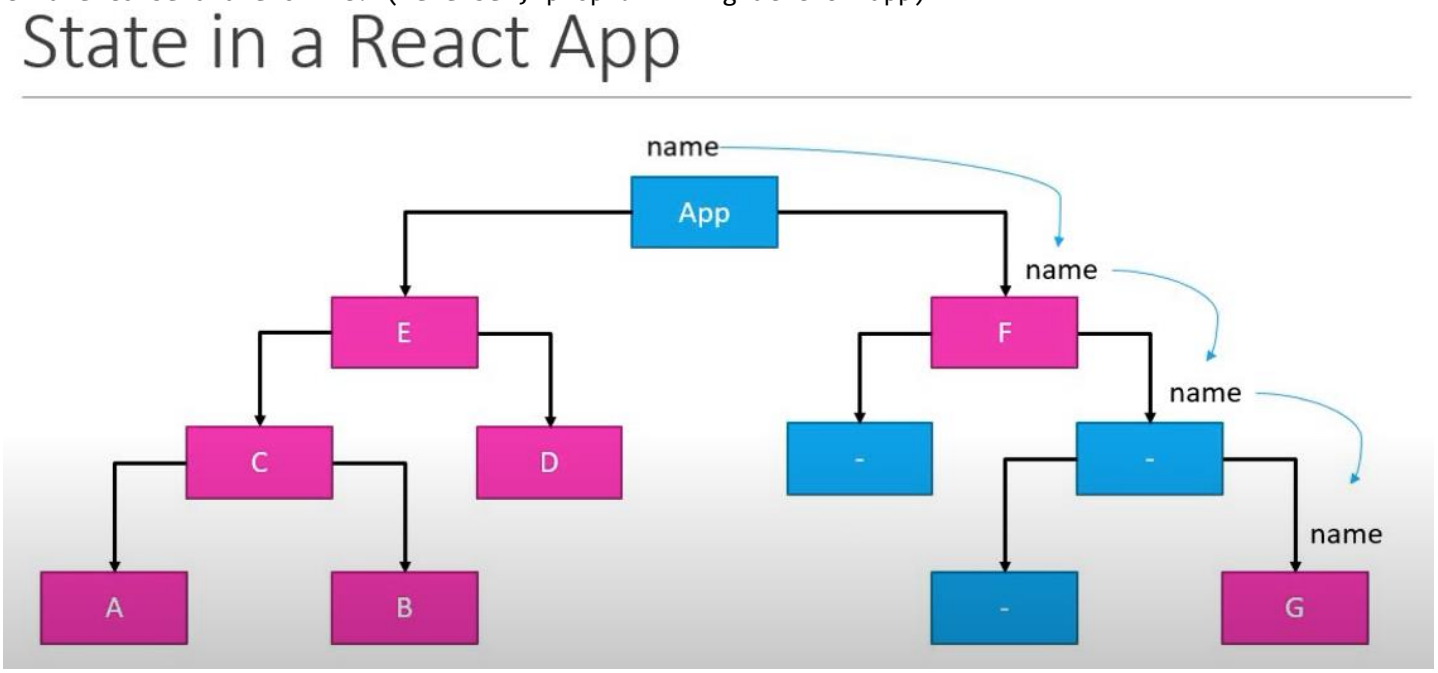
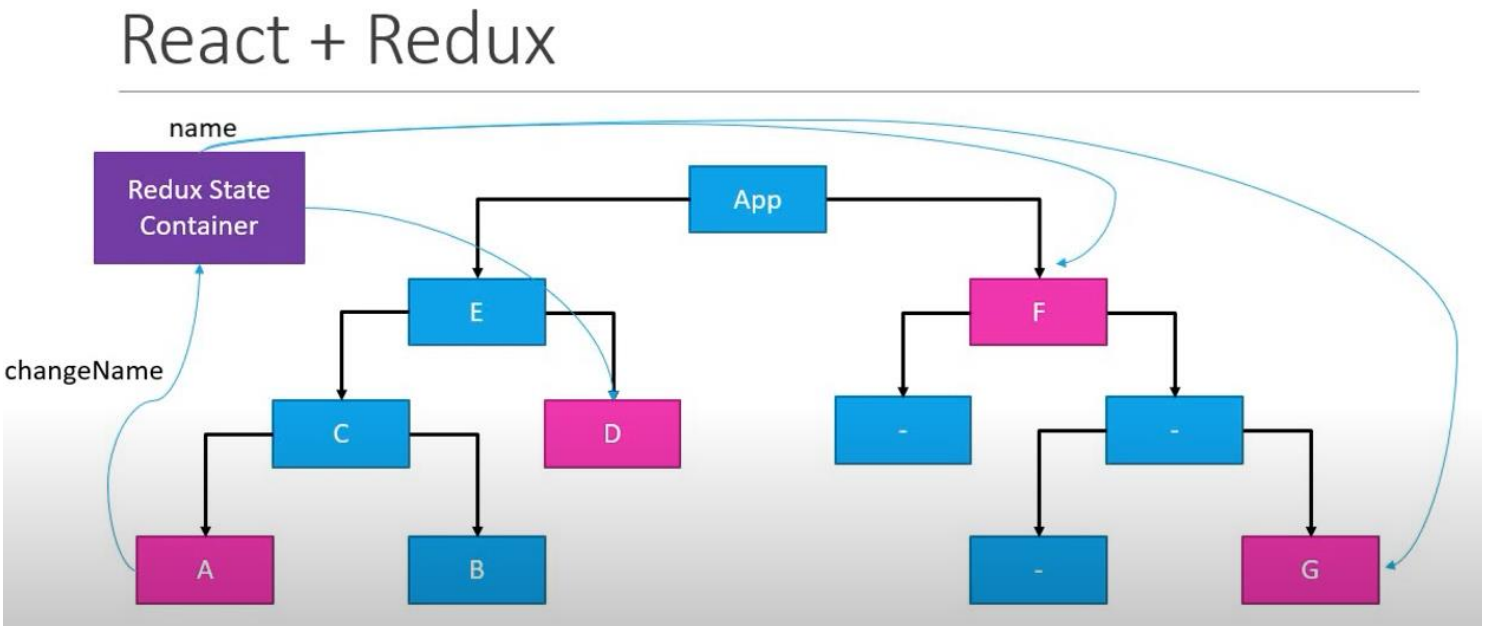
Redux

Redux is a predictable state container for JavaScript apps.
Redux not tied to React, can be used with Angular, Vue or even vanilla JS.
Redux stores the state of the application. State of an App represented by all individual components of that App. Redux manages the applications state.

Managing the state of a react app cam be trouble-some. In A we have an input where user can type in their name. Now if we want to display it on B we need to send it C, but if we want to display in D we need to send it to E. What about if we want to display in G? We need to lift the state of the component and provide it as props. That means many intermediate components are don`t really need it, but have to be aware of it. (remember, prop-drilling at CRUD app)



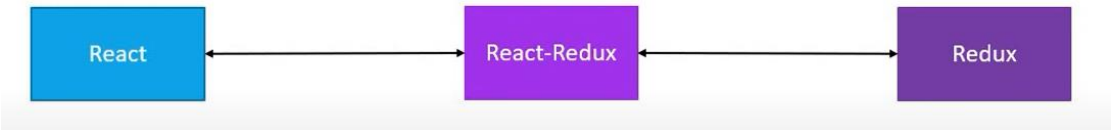
And here is why Redux can help us, no nonsense:



Altough in React we can use useContext and useReducer but Redux was relesead before we had them available. Regarless there are a tons of pros to use Redux.

React – redux

React UI library, Redux state management library. They work independently from each other. We need a React-Redux pacakadge. React-Redux bind them together.



Dev env setup

go repo fold
npm init –yes
npm install redux
create index.js (console.log("its on))
node index (should return "its on")

Three core concepts

Cake Shop

Entities	Activities
Shop – Stores cakes on a shelf	Customer – Buy a cake
Shopkeeper – At the front of the store	Shopkeeper – Remove a cake from the shelf
Customer – At the store entrance	– Receipt to keep track

Store holding the state of the application, **Reducer** will carry out the transition of the **Action**.

Cake Shop Scenario	Redux	Purpose
Shop	Store	Holds the state of your application
Intention to BUY_CAKE	Action	Describes what happened
Shopkeeper	Reducer	Ties the store and actions together

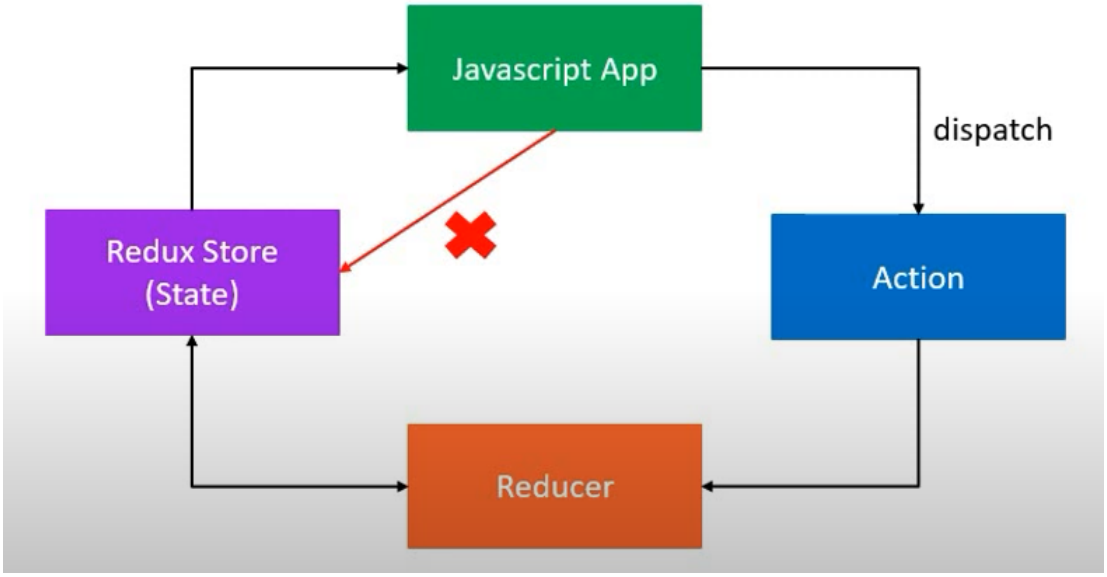
1. First principle – The state of the whole application is stored in and object tree withing a single store

2. Second principle - The only way to change the state is to emit and action, and object describing what happened (we need to let Redux know of the action)

3. Third principle – To specify how the state tree is transformed by actions, you write pure reducers (previousState, action) => newState

```
const reducer = (state, action) => {
  switch (action.type) {
    case BUY_CAKE: return {
      numOfCakes: state.numOfCakes - 1
    }
  }
}
```

We have a JS appliction, our application always subsribed to the Redux Store. The app not directly update the store. Its dispacth and action which gets handled by the Reducer. And as the App subscribed to the store it will get updated.



Actions

The only way we can App can interact with the store. Carry some information to redux store. It's plain JS objects with the "type" propety that indicates the type of action being performed. Its tipically defined as a string constants.

```
const BUY_CAKE = "BUY_CAKE"

{
    type: BUY_CAKE
}
```

Action-creator

Action is and object with type property. Action-creator is a function that returns an action.

```
const BUY_CAKE = "BUY_CAKE"

function buyCake() {
    return {
        type: BUY_CAKE,
        info: "first redux action"
    }
}
```

Reducer

Reducers specify howe the app`s state changed in response to the actions sent to the store. It's a function that accepts state and action as arguments, and returns the next state of the application. (previousState, action) => newState

```
const initalState = {
    numOfCakes: 10
}

const reducer = (state = initalState, action) => {
    switch(action.type) {
        case BUY_CAKE: return {
            ...state,
            numOfCakes: state.numOfCakes - 1
        }

        default: return state
    }
}
```

Redux Store

One store for the entire application. Responsibilites:

- Holds application state
- Allows access to state via **getState()**
- Allows state to be updated via **dispatch(action)**
- Register listeners via **subscribe(listener)**
- Handles unregistering of listeners via the function returned by **subscribe(listener)**

Holds application state

Creating store (usually w is ES6 import method)
We have our initial state returned by reducer
Which passed on to the createStore

```
const redux = require("redux")
const createStore = redux.createStore

const BUY_CAKE = "BUY_CAKE"

function buyCake() {
    return {
        type: BUY_CAKE,
        info: "first redux action"
    }
}

const initalState = {
    numOfCakes: 10
}

const reducer = (state = initalState, action) => {
    switch(action.type) {
        case BUY_CAKE: return {
            ...state,
            numOfCakes: state.numOfCakes - 1
        }

        default: return state
    }
}

const store =  createStore(reducer)
```

Allows access to state via **getState()**

As we not performed any state transition yet, this console log should return the applications initial state

```
console.log("Inital state", store.getState())
```

Register listeners via **subscribe (listener)**

Subscribe method accept a function
keep things simple we use and arrow function with console.log

```
store.subscribe(() => console.log("Updated state", store.getState()))
```

Allows state to be updated via **dispatch(action)**
store.dispatch accept and action as param,
we could include the action itself but we have
an action-creator buyCake()

```
store.dispatch(buyCake())
```

Handles unregistering of listeners via the function returned by **subscribe(listener)**

```
const unsubscribe = store.subscribe(() => console.log("Updated state", store.getState()))
store.dispatch(buyCake())
unsubscribe()
```

All completed, here is in full
Create store => Declare initial state AND reducer => Define action and action-creators => Subscribe to the store => Dispatch action to update the store => Unsibscribe to the changes

index.js > reducer

```
const redux = require("redux")
const createStore = redux.createStore

const BUY_CAKE = "BUY_CAKE"

function buyCake() {
    return {
        type: BUY_CAKE,
        info: "first redux action"
    }
}

const initalState = {
    numOfCakes: 10
}

const reducer = (state = initalState, action) => {
    switch(action.type) {
        case BUY_CAKE: return {
            ...state,
            numOfCakes: state.numOfCakes - 1
        }

        default: return state
    }
}

const store =  createStore(reducer)
console.log("Inital state", store.getState())
const unsubscribe = store.subscribe(() => console.log("Updated state", store.getState()))
store.dispatch(buyCake())
store.dispatch(buyCake())
store.dispatch(buyCake())
unsubscribe()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\rohad\Documents\GitHub\Redux-tutorial> node index
Inital state { numOfCakes: 10 }
Updated state { numOfCakes: 9 }
Updated state { numOfCakes: 8 }
Updated state { numOfCakes: 7 }
```

In theory we could pass the object itself to the dispatch method, but later on if we need to add more params or get used several places, we will need to change it everywhere, hence better to use action-creator

Multiple reducers

This is just and example, but moving forward is better not to use a single object and a single reducer. It will be hard to navigate and debug

index.js > unsubscribe

```
const redux = require("redux")
const createStore = redux.createStore

const BUY_CAKE = "BUY_CAKE"
const BUY_ICECREAM = "BUY ICECREAM"

function buyCake() {
    return {
        type: BUY_CAKE,
    }
}

function buyIceCream() {
    return {
        type: BUY_ICECREAM,
    }
}

const initalState = {
    numOfCakes: 10,
    numOfIceCreams: 20
}

const reducer = (state = initalState, action) => {
    switch(action.type) {
        case BUY_CAKE: return {
            ...state,
            numOfCakes: state.numOfCakes - 1
        }
        case BUY_ICECREAM: return {
            ...state,
            numOfIceCreams: state.numOfIceCreams - 1
        }

        default: return state
    }
}

const store =  createStore(reducer)
console.log("Inital state", store.getState())
const unsubscribe = store.subscribe(() => console.log("Updated state", store.getState()))
store.dispatch(buyCake())
store.dispatch(buyCake())
store.dispatch(buyIceCream())
store.dispatch(buyIceCream())
unsubscribe()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\rohad\Documents\GitHub\Redux-tutorial> node index
Inital state { numOfCakes: 10, numOfIceCreams: 20 }
Updated state { numOfCakes: 9, numOfIceCreams: 20 }
Updated state { numOfCakes: 8, numOfIceCreams: 20 }
Updated state { numOfCakes: 8, numOfIceCreams: 19 }
Updated state { numOfCakes: 8, numOfIceCreams: 18 }
```

#

Combine reducers

Redux provide a method to combine reducers. We will need this as const store = createStore(reducer) only accept one.

```
const redux = require("redux")
const createStore = redux.createStore
const combineReducers = redux.combineReducers

const BUY_CAKE = "BUY_CAKE"
const BUY_ICECREAM = "BUY ICECREAM"

function buyCake() {
    return {
        type: BUY_CAKE,
    }
}

function buyIceCream() {
    return {
        type: BUY_ICECREAM,
    }
}

const initalCakeState = {
    numOfCakes: 10,
}

const initalIceCreamState = {
    numOfCakes: 20,
}
X
const cakeReducer = (state = initalCakeState, action) => {
    switch(action.type) {
        case BUY_CAKE: return {
            ...state,
            numOfCakes: state.numOfCakes - 1
        }

        default: return state
    }
}

const iceCreamReducer = (state = initalIceCreamState, action) => {
    switch(action.type) {
        case BUY_ICECREAM: return {
            ...state,
            numOfIceCreams: state.numOfIceCreams - 1
        }

        default: return state
    }
}

const rootReducer = combineReducers({
    cake: cakeReducer,
    iceCream: iceCreamReducer
})
const store =  createStore(rootReducer)
console.log("Inital state", store.getState())
const unsubscribe = store.subscribe(() => console.log("Updated state", store.getState()))
store.dispatch(buyCake())
store.dispatch(buyCake())
store.dispatch(buyIceCream())
store.dispatch(buyIceCream())
unsubscribe()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

    at Module.load (node:internal/modules/cjs/loader:1080:32)
PS C:\Users\ychad\Documents\GitHub\Redux-tutorial> node index
Inital state { cake: { numOfCakes: 10 }, IceCream: { numOfCakes: 20 } }
Updated state { cake: { numOfCakes: 9 }, IceCream: { numOfCakes: 20 } }
Updated state { cake: { numOfCakes: 8 }, IceCream: { numOfCakes: 20 } }
Updated state {
  cake: { numOfCakes: 8 },
  IceCream: { numOfCakes: 20, numOfIceCreams: NaN }
}
Updated state {
  cake: { numOfCakes: 8 },
  IceCream: { numOfCakes: 20, numOfIceCreams: NaN }
}
```

Middleware

- The suggested way to extend Redux with custom functionality
- Provides a third-party extension point between dispatching an action and the moment it reaches the reducer
- Use middleware for loggin, crash reporting, performing asynchronous tasks etc

npm install redux-logger

```
const redux = require("redux")
const reduxLogger = require("redux-logger")

const createStore = redux.createStore
const combineReducers = redux.combineReducers
const logger = reduxLogger.createLogger()
```

The redux library provides a function called apply middleware

```
const applyMiddleware = redux.applyMiddleware
```

At createStore function we pass on a second parameter which is the applyMiddleware function
We only using one middleware at the moment but we could have many more

```
const store =  createStore(rootReducer, applyMiddleware(logger))
```

We remove the console.log as now we have the logger middleware

```
const store =  createStore(rootReducer, applyMiddleware(logger))
console.log("Inital state", store.getState())
const unsubscribe = store.subscribe(() => console.log("Updated state", store.getState()))
store.dispatch(buyCake())
store.dispatch(buyCake())
store.dispatch(buyIceCream())
store.dispatch(buyIceCream())
unsubscribe()
```

```
const store =  createStore(rootReducer, applyMiddleware(logger))
console.log("Inital state", store.getState())
const unsubscribe = store.subscribe(() => {})
store.dispatch(buyCake())
store.dispatch(buyCake())
store.dispatch(buyIceCream())
store.dispatch(buyIceCream())
unsubscribe()
```

If we run node index.js this will be the output
We have our initial state logged by us but the rest is the logger showing
prev state => action => next state

```
PS C:\Users\rohad\Documents\GitHub\Redux-tutorial> node index
Inital state { cake: { numOfCakes: 10 }, iceCream: { numOfCakes: 20 } }
 action BUY_CAKE @ 16:16:37.108
   prev state { cake: { numOfCakes: 10 }, iceCream: { numOfCakes: 20 } }
   action     { type: 'BUY_CAKE' }
   next state { cake: { numOfCakes: 9 }, iceCream: { numOfCakes: 20 } }
 action BUY_CAKE @ 16:16:37.111
   prev state { cake: { numOfCakes: 9 }, iceCream: { numOfCakes: 20 } }
   action     { type: 'BUY_CAKE' }
   next state { cake: { numOfCakes: 8 }, iceCream: { numOfCakes: 20 } }
 action BUY ICECREAM @ 16:16:37.111
   prev state { cake: { numOfCakes: 8 }, iceCream: { numOfCakes: 20 } }
   action     { type: 'BUY ICECREAM' }
   next state {
    cake: { numOfCakes: 8 },
    iceCream: { numOfCakes: 20, numOfIceCreams: NaN }
  }
 action BUY ICECREAM @ 16:16:37.112
   prev state {
    cake: { numOfCakes: 8 },
    iceCream: { numOfCakes: 20, numOfIceCreams: NaN }
  }
   action     { type: 'BUY ICECREAM' }
   next state {
    cake: { numOfCakes: 8 },
    iceCream: { numOfCakes: 20, numOfIceCreams: NaN }
  }
```

Asynchronous actions

So far the application using Syncrnous actions, but later on it will needed making API calls etc.

State

```
state = {
  loading: true,
  data: [ ],
  error: ''
}
```

loading - Display a loading spinner in your component
data - List of users
error – Display error to the user

Actions

FETCH_USERS_REQUEST – Fetch list of users
FETCH_USERS_SUCCESS – Fetched successfully
FETCH_USERS_FAILURE – Error fetching the data

Reducers

case: **FETCH_USERS_REQUEST**
loading: true

case: **FETCH_USERS_SUCCESS**
loading: false
users: data (from API)

case: **FETCH_USERS_FAILURE**
loading: false
error: error (from API)

State and actions via action creators

```
const initalState = {
    loading: false,
    users: [],
    error: ""
}

const FETCH_USERS_REQUEST = "FETCH_USERS_REQUEST"
const FETCH_USERS_SUCCESS = "FETCH_USERS_SUCCESS"
const FETCH_USERS_FAILURE = "FETCH_USERS_FAILURE"

const fetchUsersRequest = () => {
    return {
        type: FETCH_USERS_REQUEST
    }
}

const fetchUsersSuccess = users => {
    return {
        type: FETCH_USERS_SUCCESS,
        payload: users
    }
}

const fetchUsersFailure = error => {
    return {
        type: FETCH_USERS_FAILURE,
        payload: error
    }
}
```

Reducer

```
const reducer = (state = initalState, action) => {
  switch (action.type) {
    case FETCH_USERS_REQUEST:
      return {
        ...state,
        loading: true,
      };
    case FETCH_USERS_SUCCESS:
      return {
        loading: false,
        users: action.payload,
        error: "",
      };
    case FETCH_USERS_FAILURE:
      return {
        loading: false,
        users: [],
        error: action.payload,
      };
  }
};
```

```
const redux = require("redux");
const createStore = redux.createStore;
...
const store = createStore(reducer);
```

Async action creators

axios – Request to an API endpoint
redux-thunk – Define async action creators (middleware library)

npm install axios redux-thunk

```
const redux = require("redux")
const createStore = redux.createStore
const applyMiddleware = redux.applyMiddleware
const thunkMiddleware = require("redux-thunk").default
const axios = require("axios")
```

```
const store = createStore(reducer, applyMiddleware(thunkMiddleware));
```

redux-thunk make us able to return a **function via an action creator** and **not just an action object**

```
const fetchUsers = () => {
  return function (dispatch) {
    dispatch(fetchUsersRequest())
    axios
      .get('https://jsonplaceholder.typicode.com/users')
      .then(response => {
        // response.data is the users
        const users = response.data.map(user => user.id)
        dispatch(fetchUsersSuccess(users))
      })
      .catch(error => {
        // error.message is the error message
        dispatch(fetchUsersFailure(error.message))
      })
  }
}
```

Continues next page in full …

Creating an action creator fetchUsers => using redux-thunk we can return a function => which we dispatch to fetchUrserRequest() so loading will be true => using axios to get the api response => dispatch to either fetchUserSuccess or fetchUsersFailure

Here it is in full:

```
const redux = require("redux");
const createStore = redux.createStore;
const applyMiddleware = redux.applyMiddleware;
const thunkMiddleware = require("redux-thunk").default;
const axios = require("axios");

const initalState = {
  loading: false,
  users: [],
  error: "",
};

const FETCH_USERS_REQUEST = "FETCH_USERS_REQUEST";
const FETCH_USERS_SUCCESS = "FETCH_USERS_SUCCESS";
const FETCH_USERS_FAILURE = "FETCH_USERS_FAILURE";

const fetchUsersRequest = () => {
  return {
    type: FETCH_USERS_REQUEST,
  };
};

const fetchUsersSuccess = (users) => {
  return {
    type: FETCH_USERS_SUCCESS,
    payload: users,
  };
};

const fetchUsersFailure = (error) => {
  return {
    type: FETCH_USERS_FAILURE,
    payload: error,
  };
};
```

```
const fetchUsers = () => {
  return function (dispatch) {
    dispatch(fetchUsersRequest())
    axios
      .get('https://jsonplaceholder.typicode.com/users')
      .then(response => {
        // response.data is the users
        const users = response.data.map(user => user.id)
        dispatch(fetchUsersSuccess(users))
      })
      .catch(error => {
        // error.message is the error message
        dispatch(fetchUsersFailure(error.message))
      })
  }
}

const reducer = (state = initalState, action) => {
  switch (action.type) {
    case FETCH_USERS_REQUEST:
      return {
        ...state,
        loading: true,
      };
    case FETCH_USERS_SUCCESS:
      return {
        loading: false,
        users: action.payload,
        error: "",
      };
    case FETCH_USERS_FAILURE:
      return {
        loading: false,
        users: [],
        error: action.payload,
      };
  }
};

const store = createStore(reducer, applyMiddleware(thunkMiddleware));
store.subscribe(() => {
  console.log(store.getState());
});
store.dispatch(fetchUsers());
```

Success return:

```
PS C:\Users\rohad\Documents\GitHub\Redux-tutorial> node asyncActions.js
{ loading: true, users: [], error: '' }
{
  loading: false,
  users: [
    1, 2, 3, 4,  5,
    6, 7, 8, 9, 10
  ],
  error: ''
}
PS C:\Users\rohad\Documents\GitHub\Redux-tutorial>
```

Failure return:

```
PS C:\Users\rohad\Documents\GitHub\Redux-tutorial> node asyncActions.js
{ loading: true, users: [], error: '' }
{
  loading: false,
  users: [],
  error: 'Request failed with status code 404'
}
```