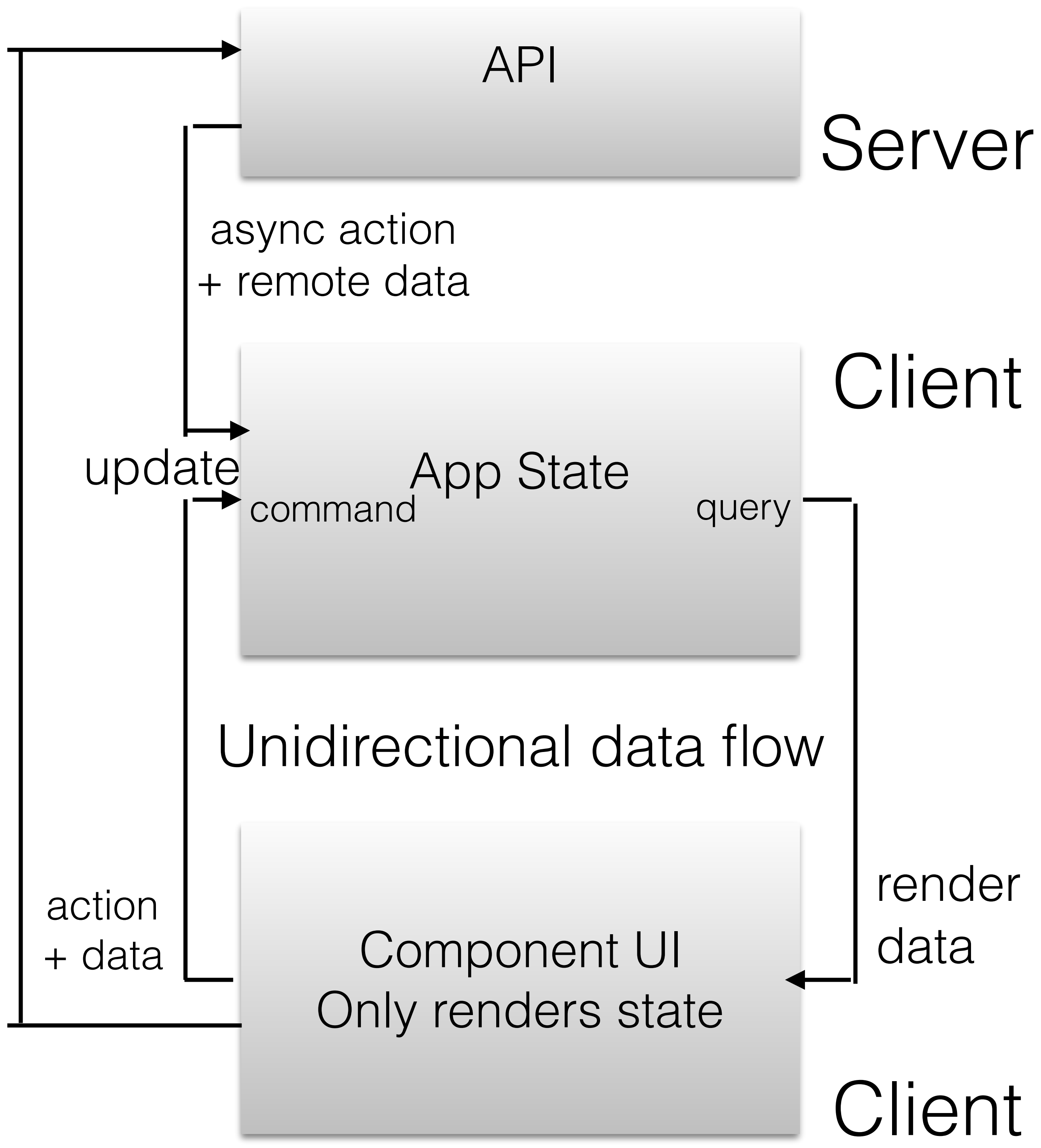
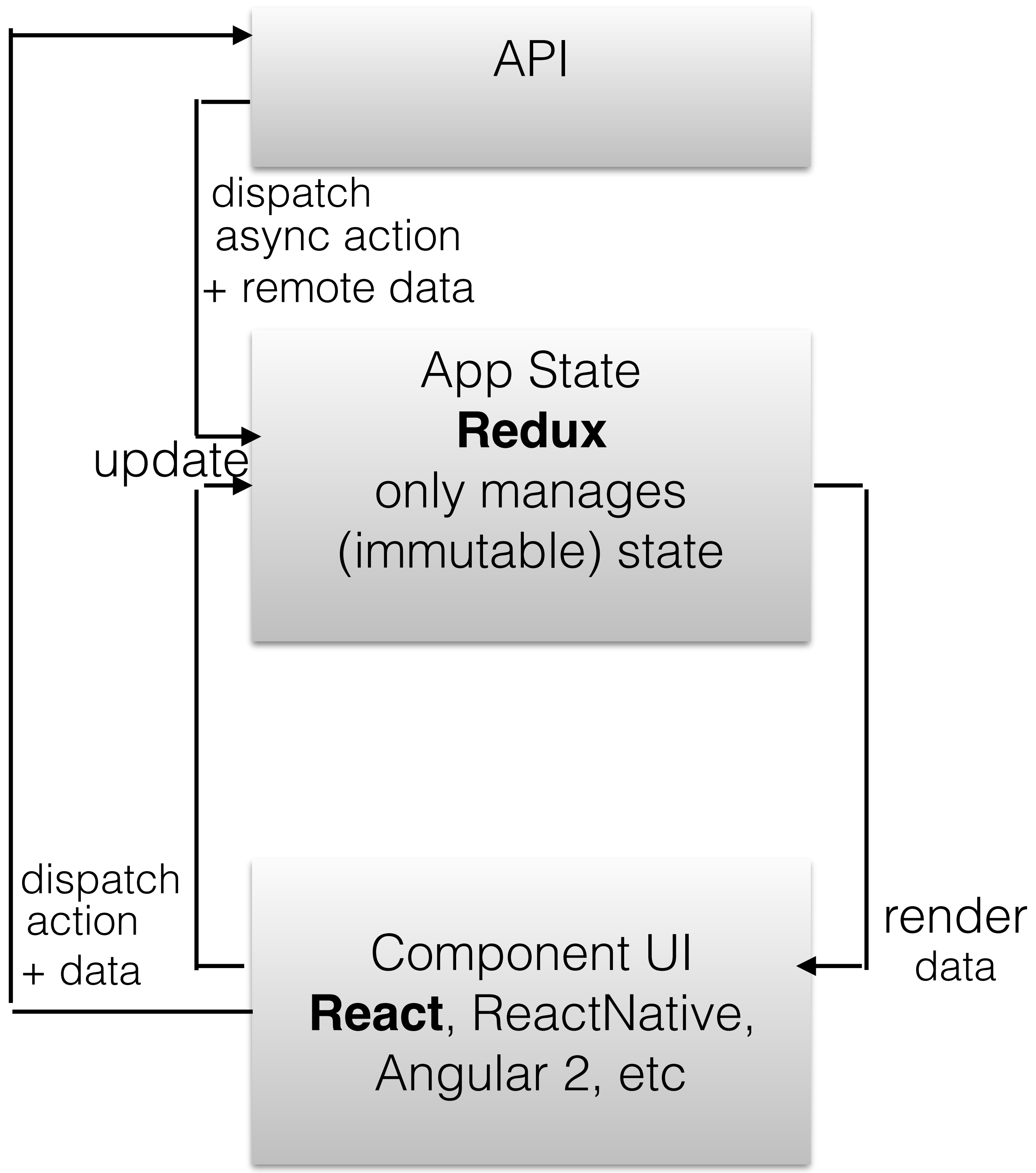
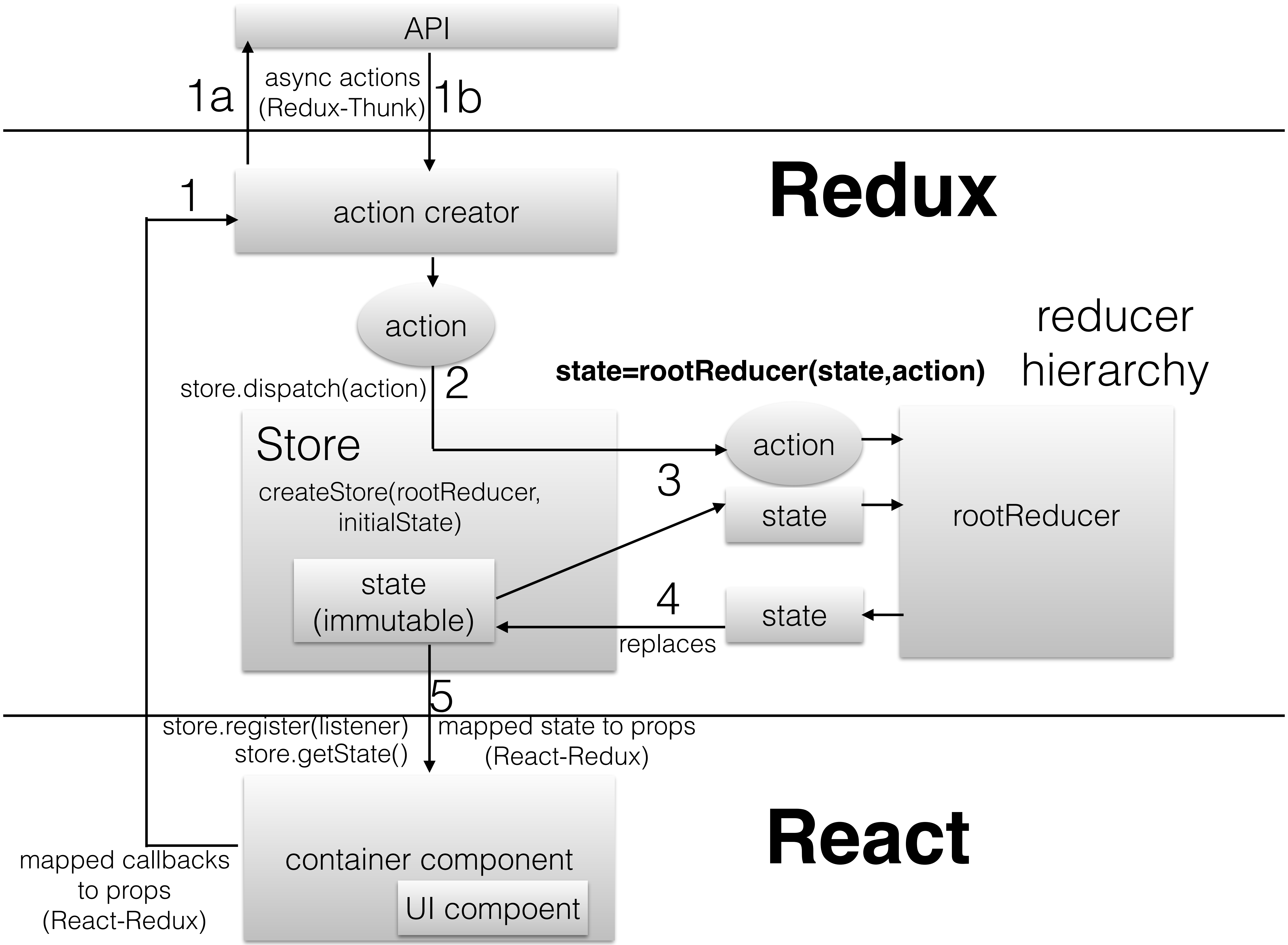
SPA application architecture



Redux SPA application architecture




API
1a
async actions
(Redux-Thunk)
1b
Redux
1
action creator
action
reducer
hierarchy
state=rootReducer(state,action)
store.dispatch(action)
2
Store
createStore(rootReducer,
initialState)
action
3
state
rootReducer
state
(immutable)
4
state
replaces
5
store.register(listener)
store.getState()
mapped state to props
(React-Redux)
React
mapped callbacks
to props
(React-Redux)
container component
UI compoent

Redux Action

Plain old javascript object

```
{
    type:RATE_COURSE
}
```

```
{
    type:RATE_COURSE,
    rating:1
}
```

```
{
    type:RATE_COURSE,
    rating:2,
    author:{name:'areg'}
}
```

Redux Action Creator

Plain old javascript function that returns an action

```
rateCouse(rating){
    return {
            type:RATE_COURSE,
            rating:rating
            };
}
```

Called within UI callback:

```
action = rateCouse(1)
```

Redux Application State

Plain old javascript object

Holds all application data required by the UI

It’s a state tree that will match the reducer hierarchy tree

```
{
    blogs:[{title:’a’},{title:’b’}],
    author:{name: ‘areg’},
    sidebar:{
                tags:[‘a’,b’]
                rating:10
            }
}
```

Redux Store

Container for application state

Receives actions from UI or network

Sends actions and current state to rootReducer

Gets new state back from rootReducer

(should only create a singleton store for entire application)

Creating the store

```
store = createStore(rootReducer)
store = createStore(rootReducer,initialState)
```

All reducers combined create the initial state of the store

Accessing the store state

```
state = store.getState()
```

```
state.blogs[0]
state.author
state.author.name
state.sidebar.tags[]
state.sidebar.rating
```

Dispatching an action to store

UI callback calls action creator than calls dispatch:

```
 action = addAuthorActionCreator(author)
      store.dispatch(action)
```

The dispatched action will be sent by the store to the rootReducer which in turn will be passed down to all reducers in the hierarchy to build the new state of the application

Reducer

Plain old javascript function

state=reducer(state,action)

new_state=reducer(curr_state={},action)

All actions are passed to all reducers

Three strict rules for writing reducers:

Rule 1:
the input state should have a default value for the initial state in case no initial state is specified for the store

Rule 2:
if the reducer does not handle an action type it should return the state passed to it un modified

Rule 3:
if the reducer handles the action the current state should not be modified. Instead it should be copied to a new object and the new object can be modified and returned as the new state

Reducer example

```
blogReducer(state=[],action){
    if(action.type=== ADD_BLOG)
        return […state,Object.Assign({},action.blog)];
    return state;
}
```

Called by redux store

```
state = blogReducer(state,action)
```

How the store uses reducer hierarchy to build the store state

create store passing in root reducer

```
     store = createStore(rootReducer,initialState={})
```

whenever an action is dispatched to the store
the store calls the root reducer:

```
     state = rootReducer(state,action)
```

reducer hierarchy

reducer hierarchy is normally declared in reducers/index.js

reducers are composable via redux's CombineReducer function

```
rootReducer = CR({
    blogs:blogReducer,
    author:authorReducer,
    sidebar:CR({
                tags:tagsReducer
                rating:ratingReducer
            })
});
```

CombineReducer simply returns a special reducer function that basically calls each reducer in the object passed to it, passing that reducer the action and part of the state that was passed to it, and sets the result of the call to the reducer to the object key that the reducer is assigned to.

Reducer tree relationship to store state tree
The reducer tree builds the state tree

```
rootReducer = CR({
    blogs:blogReducer,
    author:authorReducer,
    sidebar:CR({
                tags:tagsReducer
                rating:ratingReducer
          })
});
```

```
state = {
    blogs:[{title:’a’},{title:’b’}],
    author:{name: ‘areg’},
    sidebar:{
                tags:[‘a’,b’]
                rating:10
          }
}
```

Through CombineReducers the store passes each reducer in the reducer tree the dispatched action along with the data from the matching key of the state tree that the reducer is assigned to. The result of the each reducer call is set back to the matching key as the new state data for that key.

The object wrapped in CR is set to the key mapped to CR with the exception of the top level CR which is set to the state returned by store.getState().

Each reducer in the hierarchy only GETS the part of the state tree that is assigned to the same key that the reducer is assigned to.

```
add_blog_action = {type:add_blog,
                  blog:{title:’b’}}

curr_state = {
    blogs:[{title:’a’}],
    author:{name: ‘areg’},

}

rootReducer = CR({
    blogs:blogReducer,
    author:authorReducer,

});

new_state= rootReducer(curr_state,add_blog_action)

    [{title:’a’},{title:’b’}]= blogReducer([{title:’a’}],action)

    {name: ‘areg’}= authorReducer({name: ‘areg’},action)
```

```
curr_state = {
     blogs:[{title:'a'}],
     author:{name: 'areg'},

}
```

```
rootReducer = CR({
     blogs:blogReducer,
     author:authorReducer,

});
```

```
{type:add_blog,
 blog:{title:'b'}}
```

```
new_state= rootReducer(curr_state,add_blog_action)
```

```
new_state = {
    blogs: blogReducer(curr_state.blogs,action),
    author: authorReducer(curr_state.author,action)
}
```

```
new_state = {
    blogs: blogReducer([{title:'a'}],action),
    author: authorReducer({name: 'areg'},action)
}
```

```
new_state = {
    blogs:[{title:'a'},{title:'b'}],
    author:{name: 'areg'},
}
```

Summary

-Redux allows you to separate handling application logic from the application UI for extremely maintainable code

-All application logic that creates the new application state resides in reducers

-A reducer is simply a function that takes in the current state and an action and returns a new state without mutating the current state

-You don't need a reducer tree. You can do all logic in only a single root reducer but splitting it up will help manage complexity. For instance the reducer tree can produce a state tree that maps directly to the react component hierarchy

-The immutable nature of redux allows for efficiency in determining state changes and allows complex features such as undo\redo and time travel debugging with zero code