EOS Todo Dapp - Free RAM Model and State Management - EOS + Redux = <3

leordev (46) (/@leordev) 在 eos (/trending/eos) • 20天前

```
Dispatching Action from Block 169 - Trx fbf21b0f17fbc05c98c71162cc928a1eaadac5f4ad90b11548c0d3a27d1655cd:
 { type: 'edittodo',
  account: 'todo',
  authorization: [ { actor: 'eosio', permission: 'active' } ],
  data: { id: 2, text: 'win some battles' } }


>>> updated state:
 { todos:
   [ { id: 1, text: 'feed monsters', author: 'eosio', completed: true },
     { id: 2,
       text: 'win some battles',
       author: 'eosio',
       completed: false } ] }
```

You probably heard about the benefits of working with immutable states. That's the whole base of the blockchain, in ugly words, a big immutable database (decentralized of course). I'm not entering in details of the advantages of immutability, it's already changing the world and you can see it in our lovely EOS blockchain.

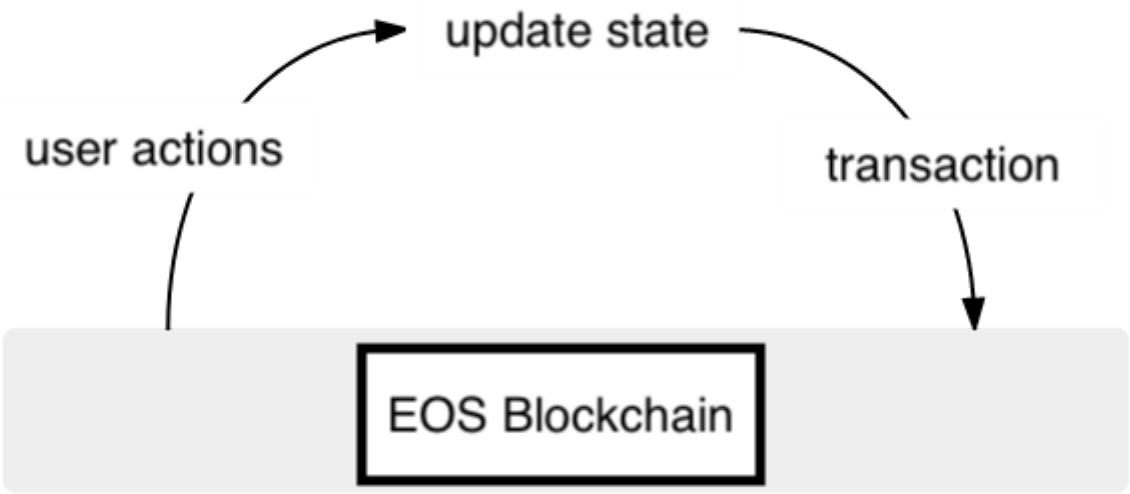
Now, inspired by the amazing DecentTwitter FREE RAM Dapp built by @kesarito (/@kesarito) and also by our outrageous EOS RAM price, I was studying and playing with Free RAM models and what should go inside EOS `multi_index` tables and what does not.

Functional Programming FTW

Working a lot in the last years with functional languages like Elixir and Elm, you have something in common: Immutability. The state is immutable and you actually use functions to update the state. You are not really updating a state, you are always creating a new one as it's immutable.

I would like to pick The Elm Architecture™ in this example, as it is the perfect fit for browsers (which I think is the best use case in how to handle a million things running in parallel - and that's what EOS will achieve soon). So this is the Elm Model-View-Update Architecture:

Does it not look like the **EOS Blockchain**?



My dirty analogy of The Elm Architecture to EOS Blockchain Action/State update flow

Explaining my flow:

1. Actions: an user (or oracle) sign and push an action
2. Update State: this is our smart contract validating and updating the tables **(OPTIONAL!)**

3. Transaction: if your smart contract ran fine a new transaction is generated and added to the next block
4. Everything above on top of EOS Blockchain which is generating blocks (with or without new transactions)

Yeah ok, maybe I'm just forcing something here but you will understand what I want to get if you follow me along.

Optional RAM usage and introducing Redux to handle Data

In the above EOS Blockchain actions flow step 2 I said that update table is optional. And indeed it is, check `DecentTwitter` contract actions now:

```
    void tweet(std::string msg) {}
    void reply(std::string id, std::string msg) {}
    void avatar(std::string msg) {}
```

Fantastic! It just has empty body actions. It does not update any contract table, actually this contract has no table at all. You know what it means? There's no RAM cost. Free RAM Dapp!

The question here is: how to extract/generate state and handle the action data for more complex cases like an edit action? That's where I came up with Redux idea to handle the state. Redux was heavily inspired on Elm Architecture and that's why I introduced the whole Elm story above.

So what we want to do to have a safe and well-managed immutable state application from the blockchain is:

1. Listen to EOS blockchain actions relevant to our dapp, usually our contract actions only
2. Every time this new action arrives we dispatch a Redux action
3. Redux will handle the state update based on the action

I think nothing like a real example to illustrate the case. Let's create a FREE RAM **TO-DO List** Dapp! (cliche...)

EOS To-Do List Dapp

This Dapp should allow us to handle our tasks in a to-do list fashion!

EOS Smart Contract

This is a *twenty-ish* lines contract, super simple:

```
#include <eosiolib/eosio.hpp>

using namespace eosio;
using std::string;

class todo : public eosio::contract {
  public:
      todo(account_name self)
      :eosio::contract(self)
      {}

      void addtodo(uint64_t id, string text) {}
      void edittodo(uint64_t id, string text) {}
      void toggletodo(uint64_t id) {}

};

EOSIO_ABI( todo, (addtodo)(edittodo)(toggletodo) )
```

The contract has three simple empty body actions:

1. addtodo - allows you to create a new task on your todo list
2. edittodo - allows you to edit a task description on your todo list
3. toggletodo - allows you to mark a task as completed or unmark the completed flag

Node.js Backend EOS Blockchain Listener with Redux store management

Here I will explain how Redux will work and how I came up with this integration. If you can't understand it at all, please check https://redux.js.org/ - the Introduction and Basics chapter, real quick!

First step: setup the actions we want to listen from EOS Blockchain, in my case I deployed the above todo contract in an account called `todo` in my single local node:

```
// actions mapping
const ACTION_ADD_TODO = 'addtodo'
const ACTION_EDIT_TODO = 'edittodo'
const ACTION_TOGGLE_TODO = 'toggletodo'

const CONTRACT_ACTIONS = [
  {
    account: 'todo',
    actions: [ACTION_ADD_TODO, ACTION_EDIT_TODO, ACTION_TOGGLE_T
  }
]
```

Second step: setup our initial state, how we want our store to look like. In our case it's just a simple `todo` list:

```
// state
const initialState = {
  todos: []
}
```

From the above state we will listen the EOS Blockchain actions to decide how to store and manage the data.

Third step: setup the application reducer. Reducer (what originates the Redux name) is just a function that takes state and action as arguments, and returns the **next state** of the app. (Note the next state here, it's very important. Remember we are immutable, we don't change anything.)

This is the application reducer code:

```
// reducer
const appReducer = (state = initialState, action) => {
  switch (action.type) {
    case ACTION_ADD_TODO:
      return addTodoReducer(state, action)
    case ACTION_EDIT_TODO:
      return editTodoReducer(state, action)
    case ACTION_TOGGLE_TODO:
      return toggleTodoReducer(state, action)
    default:
      // return the current state
      // if the action is unknown
      return state
  }
}
```

So in the above code we check the action type (action name from EOS) to decide which reducer we will call to update our state, if it's an unknown/not-mapped action it just ignores and returns the same state.

Add Todo reducer function - it checks if the id was never used and add to our todo list. Each todo has the fields `id`, `text`, `author` and `completed`:

```
const addTodoReducer = (state, action) => {

  // check and do not add new todo if this todo id
  // already exists
  if (state.todos.filter(todo => todo.id === action.data.id).le
    return state

  const newTodo = {
    id: action.data.id,
    text: action.data.text,
    author: action.authorization[0].actor,
    completed: false
  }

  const newTodos = [ ...state.todos, newTodo ]

  return { ...state, todos: newTodos }
}
```

It's important to note above that we **NEVER** change the current state, we always create and return a new one. We do that because we are, again, immutable. It also allows us to have cool features like time-traveler debug, undo actions etc.

Edit todo reducer - here we simply update the text of a todo, ONLY IF the actor for this action is the same as the one that created this todo task:

```
const editTodoReducer = (state, action) => {
  const updatedTodos = state.todos.map(todo => {
    if (todo.id === action.data.id &&
      todo.author === action.authorization[0].actor) {
      return {
        ...todo,
        text: action.data.text  // update text
      }
    } else {
      return todo
    }
  })

  return { ...state, todos: updatedTodos }
}
```

Toggle todo reducer - same as edit todo, it verifies if the actor is the same as the author of the task, this time it just toggles the `completed` boolean field:

```
const toggleTodoReducer = (state, action) => {
  const updatedTodos = state.todos.map(todo => {
    if (todo.id === action.data.id &&
      todo.author === action.authorization[0].actor) {
      return {
        ...todo,
        completed: !todo.completed  // toggle boolean
      }
    } else {
      return todo
    }
  })

  return { ...state, todos: updatedTodos }
}
```

Now the only thing left is to initialize the store! Here's the simple code:

```
// initialize redux store
const store = createStore(appReducer)

// Log the initial state
console.log('>>> initial state: \n', store.getState(), '\n\n')

// Every time the state changes, log it
store.subscribe(() =>
  console.log('>>> updated state: \n', store.getState(), '\n\n'
)
```

Actually only the first couple lines of the code is relevant. I'm just logging the initial state which prints this:

```
>>> initial state:
 { todos: [] }
```

And subscribing the state which prints the full state each time that we update the state. The cool thing about Redux subscription is that you could use it to serve websockets to dapp users or write to your database, and so on. The possibilities are endless.

Finally the last step is to dispatch the Redux actions every time that we have a relevant action. The blockchain listener that I built is not relevant, because you probably have yours already done, but what I'm basically doing is listening to each new block in the chain, checking if it has any transaction inside this block, selecting all the actions of it and finally filtering and dispatching each one to our reducer:

```
const filterAndDispatchAction = (newAction, trxId, block) => {
  const action = {
    type: newAction.name,
    account: newAction.account,
    authorization: newAction.authorization,
    data: newAction.data
  }

  const subscribed = CONTRACT_ACTIONS.find(item => (
    item.account === action.account &&
      item.actions.indexOf(action.type) >= 0
  ))

  if (subscribed) {
    console.log(`\nDispatching Action from Block ${block} - Trx
      action, '\n\n')
    store.dispatch(action)
  }

}
```

Remember the initial `CONTRACT_ACTIONS` we created in the beginning of this? Yes, it's finally being used to filter the relevant actions that you want.

Also the `store` variable that contains the Redux state handler, is being used to dispatch the action received from the blockchain to our Redux store. This is how everything connects. If everything goes right you will see the following log in the console:

```
Dispatching Action from Block 81 - Trx a57a690978b78b18a3a5b286
 { type: 'addtodo',
  account: 'todo',
  authorization: [ { actor: 'eosio', permission: 'active' } ],
  data: { id: 1, text: 'I need to add a database' } }


>>> updated state:
 { todos:
   [ { id: 1,
       text: 'I need to add a database',
       author: 'eosio',
       completed: false } ] }
```

Playing Todo Dapp using our backend Node.js Redux store application

The whole code is in this repository: https://github.com/leordev/eos-redux-todo

You can use the blockchain listener idea, it's super simple there and probably needs some polishing. Also I should refactor this code into different files, I just kept it simple with everything inside index.js

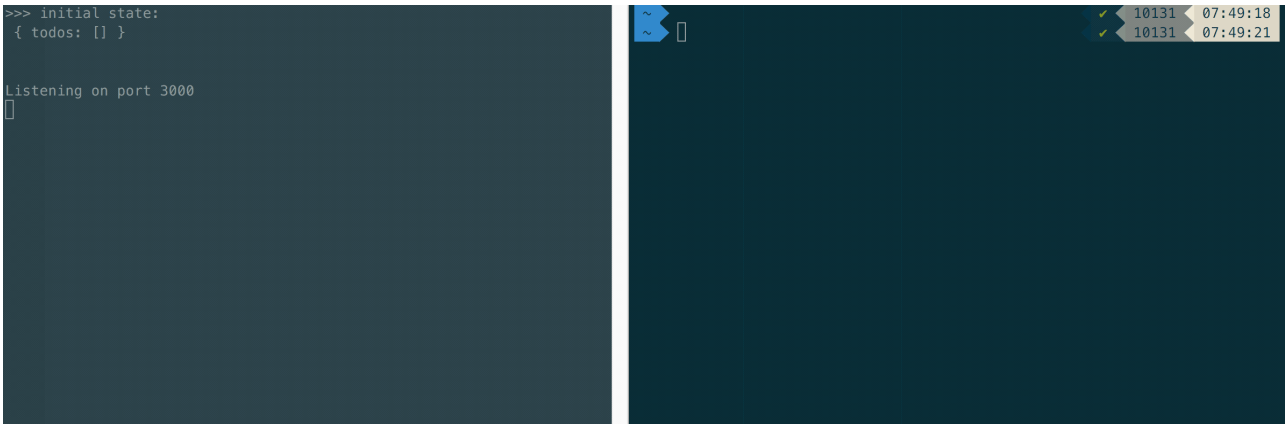
Instructions to Deploy the Contract

```
cleos create account eosio todo OWNERPUBKEY ACTIVEPUBKEY
cd eos-redux-todo/todo
eosiocpp -o todo.wast todo.cpp
eosiocpp -g todo.abi todo.cpp
cleos set contract todo ../todo
```

Instructions to Init the Nodejs Redux Store Application

```
cd eos-redux-todo
npm install
node index.js
```

Demoing



Wrapping it Up

That's it folks, that's the best way that I found to handle store management from EOS Blockchain Actions, using Redux which is a very popular and solid tool created by Dan Abramov from Facebook team. This is a nice library and has a lot of functionalities out-of-the-box like the subscription, it's easy to implement undo and time-traveling states. It's very mature and you have nice tooling around it like redux-logger and redux-devtools .

The 100% free RAM model is an interesting approach but I think it has some flaws. E.g. the way I'm filtering the editing/toggling in the todo record by author, I think it should be blocked in the EOS chain not allowing the transaction to be created, but actually we don't have a table inside the contract to match the todo ids with the respective authors. So if we have other apps consuming this contract it could wrongly consider that the todo was edited by a bad actor if it does not take care of this very same filter rule that we did in our application.

I would love to hear your feedbacks in how you are managing this and your thoughts in this approach. After we have solid progress in this Redux integration and some use cases with standardized actions I could wrap it up in a JS library and integrate with eosjs. That would be nice, just not sure if it makes sense yet. See you all!

free-ram-model-and-state-management-eos-redux-3)

排序: Trending

natp (25) **(/@natp)** · 20天前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@natp/re-leordev-eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3-20180707t124625748z) [-]

Thank you for this article. I think it will help developers understand that RAM is not the end all be all and there are ways around it.

$0.00 | 回复

mobicfly (25) **(/@mobicfly)** · 19天前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@mobicfly/re-leordev-eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3-20180708t165159657z) [-]

thanks for the tutorial, after deploy the contract, I encountered an error:
cleos push action todo addtodo '[1, "estset"]' -p eosio
Error 3050000: action exception

but no further details about it. I am running this on jungle testnet. is there any possible reason for this? thanks!

$0.00 | 回复

> **leordev** (46) **(/@leordev)** · 17天前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@leordev/re-mobicfly-re-leordev-eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3-20180709t215547103z) [-]
>
> I don't think you will have access to `eosio` account in jungle. change `-p eosio` for `-p youraccount`
>
> $0.00 | 回复

johnwilliamson (39) **(/@johnwilliamson)** · 18天前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@johnwilliamson/re-leordev-eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3-20180709t122559667z) [-]

Bloody genius, it feels so obvious now! Makes so much sense.

$0.00 | 回复

steemitboard (62) **(/@steemitboard)** · 16天前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@steemitboard/steemitboard-notify-leordev-20180711t143726000z) [-]

Congratulations @leordev (/@leordev)! You have completed the following achievement on Steemit and have been rewarded with new badge(s) :

Award for the number of upvotes received

Click on the badge to view your Board of Honor.

If you no longer want to receive notifications, reply to this comment with the word `STOP`

Do not miss the last post from @steemitboard (/@steemitboard):
SteemitBoard World Cup Contest - Croatia vs England (https://steemit.com/steemitboard/@steemitboard/steemitboard-world-cup-contest-croatia-vs-england)

Participate in the SteemitBoard World Cup Contest (https://steemit.com/steemitboard/@steemitboard/steemitboard-world-cup-contest-collect-badges-and-win-free-sbd)!
Collect World Cup badges and win free SBD
Support the Gold Sponsors of the contest: @good-karma and @lukestokes

---

Do you like SteemitBoard's project (https://steemit.com/@steemitboard)? Then **Vote for its witness** and **get one more award**!

$0.00 | 回复

lukestokes (73) **(/@lukestokes)** · 15天前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@lukestokes/re-leordev-eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3-20180711t200950835z) [-]

Fantastic post! I love that you educated while also giving all the details in a tutorial style. Very nicely done. Following for more.

$0.00 | 回复

dabble (41) **(/@dabble)** · 6小时前 (/eos/@leordev/eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3#@dabble/re-leordev-eos-todo-dapp-free-ram-model-and-state-management-eos-redux-3-20180727t014059608z) [-]

Great! I learned many things from this article.

$0.00 | 回复