

Redux, ngRedux, ~~& Sagas~~

A Gentle Introduction for Angular Developers

Jesse Warden | NationJS | September 16th, 2016

Who

- Jesse Warden
- I do JavaScript, Little Bit of Node

What We'll Cover

- What & Why of Redux
- Redux
- ngRedux

2 Things

- Pure Functions
- Redux Solves the Bubbling Problem

What & Why of Redux

- What: Predictable State Container
- Why: helps create consistent & predictable applications, easier to test

Er... wat

- Er, What: who's changing my data, when, and where in a predictable fashion

Er... Why?

- Er, Why: more hints as to where the defect are... and code you write is easier to unit test, easier to reason about in a codebase you didn't write

Predictable How

- comes from Functional Programming
- pure functions / avoiding side effects


```
function getX()  
{  
    return 1 + 2;  
}
```

```
log("getX:", getX()); // 3  
log("getX:", getX()); // 3
```

```
function getX()  
{  
    return 1 + 2 + data;  
}
```

```
var data = 'cow';  
function getX()  
{  
    return 1 + 2 + data;  
}
```

```
log("getX:", getX()); // '3cow'  
data = 1;  
log("getX:", getX()); // 4
```



```
var data = 'cow';  
function getX(data)  
{  
    return 1 + 2 + data;  
}
```

```
log("getX:", getX('cow')); // '3cow'  
data = 1;  
log("getX:", getX('cow')); // '3cow'
```

State Waaat

- "Who is the logged in user right now?"

State

- the data your front-end application shows and edits
- Where you put things
- Often called mutable state; you can change something
- you can't change 1
- 1 is 1, yo

Examples of State

- window state

```
log("window.cow first:", window.cow); // undefined
```

```
window.cow = 'moo';
```

```
log("window.cow after setting:", window.cow); // moo
```

```
window.cow = new Date();
```

```
log("window.cow after setting 2nd:", window.cow); // [object Date]
```


Examples of State

- Angular Factory state

```
// psuedo code from Angular 1  
function MyController($scope)  
{  
    $scope.cow = 'moo';  
}
```

```
expect(controller.cow).toEqual('moo') // true
```

```
function MyController()  
{  
    var vm = this;  
    vm.cow = 'moo';  
}
```

```
expect(controller.cow).toEqual('moo') // true
```

```
// psuedo code from Angular 1 / ES5, userModel.factory.js  
angular.module('com.jessewarden.main')  
  .factory('userModel', userModel);
```

```
function userModel()  
{  
  return {  
    firstName: 'Jesse',  
    lastName: 'Warden'  
  };  
}
```

```
// ES6  
export default function userModel()  
{  
  return {  
    firstName: 'Jesse',  
    lastName: 'Warden'  
  };  
}
```

Examples of Changing State

- set a variable
- have a function do it
- `Object.assign`


```
// setting variables  
var data = 'cow';  
  
expect(data).toBe('cow');  
  
data = 'moo';  
  
expect(data).toBe('moo');  
  
log('Tests passed.');
```

```
// setting variables via functions  
var data = 'cow';  
  
function setData(value)  
{  
    data = value;  
}  
  
expect(data).toBe('cow');  
  
setData('moo');  
  
expect(data).toBe('moo');  
  
log('Tests passed.');
```

// Object.assign normal way through objects

```
var data = {  
  value: 'cow'  
};
```

```
expect(data.value).toBe('cow');
```

```
var newData = Object.assign({}, data, {value: 'moo'});
```

```
expect(newData.value).toBe('moo');
```

```
expect(data !== newData).toBe(true);
```

Ok, we get predictable state

- predictable: pure functions
- state: my data in t3h RAM

The Bubble Problem

- When you start refactoring imperative code to use pure functions, you run into the bubble problem: mutable state bubbles up and out state.
- Someone, SOMEWHERE, has to eventually store the state by using a `var` vs. `const`.
- If you abstract it into a safe container, you've created Redux.

The Bubble Problem

- Imagine you can't ever use var, only const

Refactor Imperative

```
function getNotes(accountNumber)
{
    return new Promise((success, failure)=>
    {
        if(predicates.validAccountNumber(accountNumber) === false)
        {
            return failure(new Error('accountNumber is invalid.'));
        }
        oracle.getConnection()
            .then((connection)=>
            {
                connection.execute(
                    getNotesqueryString(accountNumber),
                    [],
                    function(err, result)
                    {
                        if(err)
                        {
                            console.error(err.message);
                            oracle.release(connection);
                            return failure(err);
                        }
                        oracle.release(connection);
                        try
                        {
                            success(queryResultToNoteList(result));
                        }
                        catch(parseError)
                        {
                            failure(new Error("Failed to parse query results:" + parseError.toString()));
                        }
                    }
                ));
            })
        .catch(failure);
    });
}
```

```
// and integration test for it  
it('gets a list of notes', (done)=>  
{  
  note.getNotes(1)  
    .then((result)=>  
    {  
      expect(result).toHaveLength(1);  
      done();  
    })  
    .catch(done);  
});
```


// How do you test oracle.getConnection? Sinon and/or insane mocks.

```
function getConnection()
{
  return new Promise(function(success, failure)
  {
    oracleAPI.getConnection(
    {
      user      : config.user,
      password  : config.password,
      connectionString : config.connectString
    },
    function(err, connection)
    {
      if(err)
      {
        console.error(err.message);
        return failure(err);
      }
      success(connection);
    });
  });
}
```

```
// How do you test bad password? Let's refactor.  
function getConnection(oracleAPI, config)  
{  
  return new Promise(function(success, failure)  
  {  
    oracleAPI.getConnection(  
    {  
      user : config.user,  
      password : config.password,  
      connectString : config.connectString  
    },  
    function(err, connection)  
    {  
      if(err)  
      {  
        console.error(err.message);  
        return failure(err);  
      }  
      success(connection);  
    }  
  ));  
});  
}
```

```
// and unit test
var mock = {
  getConnection: (config, callback)=>
  {
    callback(undefined, {});
  }
};
it('gives you a connection with valid creds', function()
{
  return oracle.getConnection(mock, oracle.defaultConfig());
});
```


// now, let's refactor getNotes

```
function getNotes(connection, accountNumber)
{
  return new Promise((success, failure)=>
  {
    if(predicates.validAccountNumber(accountNumber) === false)
    {
      return failure(new Error('accountNumber is invalid.'));
    }
    connection.execute(
      getNotesQueryString(accountNumber),
      [],
      function(err, result)
      {
        if (err)
        {
          console.error(err.message);
          return failure(err);
        }
        // console.log("note results:", result);
        try
        {
          success(queryResultToNoteList(result));
        }
        catch(parseError)
        {
          failure(new Error("Failed to parse query results:" + parseError.toString()));
        }
      });
  });
}
```

// and the unit test

```
var mock = {  
  execute: (queryStr, injections, callback)=>  
  {  
    callback(undefined, {});  
  }  
};  
it('returns notes', function()  
{  
  return note.getNotes(mock, 1);  
});
```

Redux

- Data for entire app in a single object. You only change it by dispatching actions with your new value. To actually change the data, you use pure functions.
- Data for entire app spread out over multiple classes. You change through method calls. To change data, you'd use getter/setters, or \$watchers.
- Made by Facebook & Dan Abramov
- @dan_abramov on Twitter

Initial State

- our data "right now"
- our data or domain model
- starts as a basic Object...
- eventually tree gets pretty big and specialized

```
// initial state  
var defaultState = {  
  loading: false,  
  loginError: undefined,  
  user: readUserFromLocalStorage(window.localStorage)  
};
```



```
var defaultState = {
  login: {
    loading: false,
    error: undefined,
    user: readUserFromLocalStorage(window.localStorage)
  },
  search: {
    text: "",
    field: "name",
    loading: false,
    results: undefined,
    error: undefined
  },
  newUserRole: {
    loading: false,
    error: undefined,
    roles: []
  },
  account: {
    loading: false,
    error: undefined,
    details: undefined,
    notes: undefined,
    hoverNote: undefined
  },
  roles: {
    loading: false,
    error: undefined,
    loadingAllUsersAndRoles: false,
    loadingAllUsersAndRolesError: undefined,
    users: undefined,
    roles: undefined,
    history: []
  }
};
```

Actions

- what happened / what do you want to change?
- Either an object...
- ... or a pure function that makes an Object

```
// actions & action creators  
{  
  type: "SEARCH",  
  text: "some search text"  
}
```



```
// mo bettah  
const SEARCH = "SERCH";  
  
function searchAction(text)  
{  
  return {  
    type: SEARCH,  
    text: text  
  };  
}
```

```
// for edit's, don't do this...  
{  
    type: "EDIT_USER",  
    user: userObject  
}
```

```
// ... do this  
{  
    type: "EDIT_USER",  
    userID: userObject.id  
}
```

Reducers

- change your data in response to what happened
- pure as possible
- like `Array.reduce`
- `_.reduce`

```
var searchResults = [  
  {  
    result: true,  
    database: 'oracle',  
    data: [...]  
  },  
  {  
    result: true,  
    database: 'mainframe',  
    data: [...]  
  }  
];
```

// what I want

```
var searchResults = [...];
```



```
// how do I get there?  
var searchResults = [  
  {  
    result: true,  
    database: 'oracle',  
    data: [1, 2, 3]  
  },  
  {  
    result: true,  
    database: 'mainframe',  
    data: [4, 5, 6]  
  },  
  {  
    result: false,  
    database: 'mongo'  
  }  
];
```

```
var reducedResults = _.reduce(searchResults, (arr, item)=>
{
  if(item.result)
  {
    arr = arr.concat(item.data);
  }
  return arr;
}, []);
log("reducedResults:", reducedResults); // [1, 2, 3, 4, 5, 6]
```



```
// action, intent to change first name
{
  type: "EDIT_USER",
  userID: userObject.id,
  firstName: 'Jesse'
}

var user = {
  firstName: 'Jessie',
  lastName: 'Warden'
};
expect(user.firstName).toBe('Jessie');
expect(user.lastName).toBe('Warden');

user = Object.assign({}, user, {firstName: 'Jesse'});

expect(user.firstName).toBe('Jesse');
expect(user.lastName).toBe('Warden');

log("Tests passed.");
```

```
export function search(state=defaultState, action)
{
  switch(action.type)
  {
    case SEARCH:
      return Object.assign({}, state, {
        loading: true,
        text: action.text,
        error: undefined
      });

    case SEARCH_ERROR:
      return Object.assign({}, state, {
        loading: false,
        error: action.searchError
      });

    case SEARCH_RESULT:
      return Object.assign({}, state, {
        loading: false,
        error: undefined,
        results: action.searchResults
      });

    default:
      return state;
  }
}
```



```
export function search(state=defaultState, action)
{
  switch(action.type)
  {
    case SEARCH:
      return searchReducer(state, action);

    case SEARCH_ERROR:
      return searchError(state, action);

    case SEARCH_RESULT:
      return searchResult(state, action);

    default:
      return state;
  }
}
```

```
function searchReducer(state, action)
{
  return Object.assign({}, state, {
    loading: true,
    text: action.text,
    error: undefined
  });
}
```

```
export const SEARCH = 'SEARCH';
export const SEARCH_RESULT = 'SEARCH_RESULT';
export const SEARCH_ERROR = 'SEARCH_ERROR';

const defaultState = {
  loading: false,
  text: '',
  results: undefined,
  error: undefined
};

export function search(state=defaultState, action)
{
  switch(action.type)
  {
    case SEARCH:
      return Object.assign({}, state, {
        loading: true,
        text: action.text,
        error: undefined
      });
  }
}
```



```
    },
    search: {
      text: {
        field: "",
        loading: false,
        results: "name",
        error: undefined,
      },
    },
    newUserRole: {
      loading: false,
      error: undefined,
      roles: [],
    },
    account: {
      loading: {
        error: undefined,
        details: false,
        notes: undefined,
        hoverNote: undefined,
      },
      roles: {
        loading: {
          error: undefined,
          loadingAllUsers: false,
          loadingAllUsersAndRoles: false,
          loadingAllUsersAndRolesError: false,
          roles: undefined,
        },
      },
    },
  },
}
```


// combineReducers example

```
import { login } from './login/login.reducer';  
import { search } from './search/search.reducer';  
import { newUserRole } from './roles/newUserRole/newUserRole.reducer';  
import { account } from './account/account.reducer';  
import { role } from './roles/roles.reducer';
```

```
import { combineReducers } from 'redux';
```

```
const rootReducer = combineReducers({  
  login,  
  search,  
  newUserRole,  
  account,  
  role  
});
```

```
export default rootReducer;
```



```
login: {
  loading: false,
  error: undefined,
  user: readUserFromLocalStorage(window.localStorage)
},
search: {
  text: "",
  field: "name",
  loading: false,
  results: undefined,
  error: undefined
},
newUserRole: {
  loading: false,
  error: undefined,
  roles: []
},
account: {
  loading: false,
  error: undefined,
  details: undefined,
  notes: undefined,
  hoverNote: undefined
},
roles: {
  loading: false,
  error: undefined,
  loadingAllUsersAndRoles: false,
  loadingAllUsersAndRolesError: undefined,
  users: undefined,
  roles: undefined,
  history: []
}
```

Store

- holds your state. There is only 1.
- you access it through getState (kinda pure)
- update state via dispatch(action)
- for views/GUI, listen via subscribe(listener)
- can set default state via 2nd param of createStore

// now create Store

```
import { createStore } from 'redux'
```

```
import searchReducer from './reducers'
```

```
const store = createStore(searchReducer);
```



```
// can I see what's inside?  
const state = store.getState();  
  
// can I hear about what changes?  
store.subscribe(()=>  
{  
  const state = store.getState();  
  log(state);  
  // {  
  //   loading: false,  
  //   results: undefined,  
  //   error: undefined  
  // }  
});
```

```
// give it some default data
const store = createStore(searchReducer, {
  loading: true,
  results: [1],
  text: "",
  error: undefined
});

// change it later
store.dispatch(
  {
    type: SEARCH,
    text: 'some search text'
  }
);

// did it change?
expect(store.getState().text).toBe('some search text');
```

```
// how do I stop listening in Views?  
const unsubscribe = store.subscribe(()=>  
{  
  
});  
  
unsubscribe();
```


Data Flow

- `store.dispatch(action)`
- reducer handles change request
- new state tree, saves it
- new state of your app via `store.subscribe(listener)`

```
// #1
store.dispatch(
  {
    type: SEARCH,
    text: 'some search text'
  }
);
```

```
// #2
// store.getState()
const currentState = {
  loading: false,
  results: undefined,
  text: "",
  error: undefined
};

// action
const action = {
  type: SEARCH,
  text: 'some search text'
};

const nextState = searchReducer(state, action);
```

```
// #3
// multiple reducers are called if a bigger tree

// #4
// new state saved
const newState = store.getState();
expect(newState).toBe(currentState);
```

Async

- THANKS MEH
- SAGAS GOOD
- WELCOME TO AMURRRRRICA, MAX

ngRedux

- \$ngReduxProvider (setup)
- \$ngRedux (connect)
- \$onDestroy (unsubscribe)
- mapStateToThis


```
import ngRedux from 'ng-redux';
import rootReducer from './reducers';

export default angular.module('project', [
  ngRedux
])
.config(($ngReduxProvider) => {
  $ngReduxProvider.createStoreWith(rootReducer, []);
})
.name;
```

```
import {LOGIN} from '../actions';

export default function LoginController(
  $ngRedux,
  $q,
  $http,
  $state)
{
  var vm = this;
  vm.hasError = false;
  vm.loginError = undefined;
  vm.loading = false;
  vm.onSubmit = onSubmit;
  vm.$onDestroy = $onDestroy;
  vm.mapStateToThis = mapStateToThis;
```

Angular 2
ngOnDestroy


```
var unsubscribe = $ngRedux.connect(vm.mapStateToProps)(vm);  
  
function $onDestroy()  
{  
  unsubscribe();  
}
```

```
function onSubmit()  
{  
  return $ngRedux.dispatch({  
    type: LOGIN,  
    EID: vm.EID,  
    $q,  
    $http,  
    $state  
  });  
}
```

```
function mapStateToThis(state)
{
  return {
    loading: state.login.loading,
    hasError: !_.isNil(state.login.error),
    loginError: state.login.error
  };
}
```



```
<p ng-if="$ctrl.hasError"  
  style="color: #660000;">{{$ctrl.loginError}}</p>
```

```
<md-progress-circular  
  ng-show="$ctrl.loading"  
  md-mode="indeterminate"></md-progress-circular>
```


```
import angular from 'angular';
import LoginController from './login.controller';


export default angular.module('project.login', [])
.component('jxlLogin', {
  templateUrl: 'main/login/login.component.html',
  controller: LoginController,
  bindings: {
    username: '<',
    password: '<'
  }
})
```


File Organization


- Node module to ES6 Module
- sock drawer vs. features
- <http://cliffmeyers.com/blog/2013/4/21/code-organization-angularjs-javascript>
- reducer(s) & saga(s) per feature


roles


 editUserRoles

 newUserRole

 rolesSelect

 services

 ic_people_white_48px.svg

 roles.component.html

JS roles.component.js

JS roles.controller.js

JS roles.controller.test.js

JS roles.page.js

JS roles.reducer.js

JS roles.routes.js

JS roles.saga.js

JS roles.saga.test.js

Conclusions

- Redux gives you a 2kb functional programming framework that ensures your data is kept as pure **as possible**.
- clear flow of data (action > reducer > store > subscribe)
- single data store, scale to multiple functions & class files
- solves the functional bubble problem via pure functions

Resources

1. Eric Elliot on What a Pure Function Is <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976>
2. Learn Array Comprehensions <http://reactivex.io/learnrx/>
3. Jesse Warden's Beginner's Guide to Functional Programming <http://jessewarden.com/2016/08/beginners-guide-to-functional-programming-part-1.html>
4. Lodash <https://lodash.com/docs>
5. Dan Abramov teaches Redux on egghead.io <https://egghead.io/lessons/javascript-redux-the-single-immutable-state-tree>
6. Redux Documentation <http://redux.js.org/docs/api/>
7. Redux Saga Documentation <http://yelouafi.github.io/redux-saga/>
8. John Papa Angular Style Guide <https://github.com/johnpapa/angular-styleguide>

Questions?

- Jesse Warden
- jesse@jessewarden.com
- [@jesterxl](#) on Twitter