Transmission (TX): A New Flux Architecture

Liyanage, H. N.
hasithaliyan@gmail.com

Abstract. Redux [1], while being a widely used architecture for applications built upon reactive component libraries such as React, has demonstrated only limited success in battling the traditional complexities involved in the development of large front-end applications. Here we briefly look at some of the reasons for this, and propose an alternative architecture derived from Flux [2], with a reactive binding driven view-model layer and an event-driven business logic layer.

1. Advantages and disadvantages of Redux

Shortly after the introduction of the Flux architecture, a wide variety of libraries and variants arose, out of which Redux eventually became dominant. All these architectures were aimed primarily at solving the problem depicted in the following diagram, taken directly from the original introductory Flux talk by Facebook developers [3]:

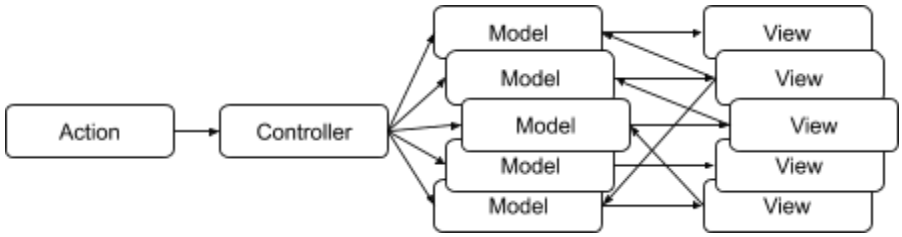

Figure 1. The problem of nested invocations between models and views in MVC, as depicted in the original Flux architecture presentation by Facebook.

When the traditional MVC (model-view-controller) architecture is applied to front-ends, it can result in difficult-to-predict execution pathways arising from nested calls from views to models and vice versa, as shown in the diagram above. The original Flux architecture solved this problem by converting this call graph into a one-way loop: views trigger actions, which are funneled through a dispatcher into a store, which in turn is connected to the view via one-way data bindings.

The Redux library took this a step further by introducing the concept of a reducer, a pure function that takes as input, an action plus a given subset of the application state, and then outputs the end state. A collection of reducer functions that listen to various actions and operate on the store, forms the basis of Redux.

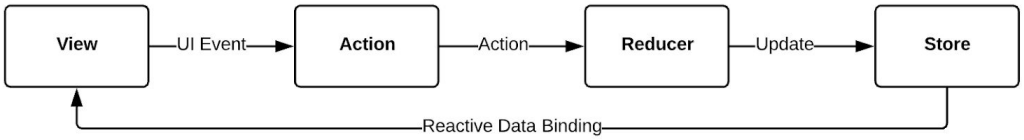

Figure 2. A depiction of the Redux architecture

Redux affords the following advantages over MVC:

1. A developer familiar with the Redux architecture can readily understand an unfamiliar code base with minimal assistance. This is due to clarity about where in the code base one needs to look for a specific type of concern (e.g. state changes will always be in the reducer, grouped by action).
2. Debugging state related bugs become significantly easier (the relevant code can always be traced to a case statement in a reducer).
3. Views are easier to develop since they are stateless (the entire state will be in the Redux store).
4. Redux, being a variant of Flux, solves the same problems as Flux: it prevents the formation of difficult-to-trace execution pathways through code.

However, as the application grows larger, Redux code bases become increasingly difficult to manage. The authors of Redux acknowledge that it may not be the right architecture for every use case [4,5]. The architecture proposed below is meant for developers of large front-end applications who experience the following difficulties:

1. Business logic cannot be neatly decoupled from the presentation layer because the business logic code is not localized. It is split between actions and reducer code. Business logic code from a Redux application is often not readily reusable in other contexts.
2. In Redux, clean modularization can be difficult when there are many interactions between different parts of the UI. For example, in an e-commerce application, if both the profile screen and the orders screen need to listen and respond to changes from the shopping cart screen, the states of those three screens cannot be neatly modularized.
3. When there are multiple effects to an action, it can be very difficult to trace execution through a Redux code base.
4. Handling asynchronous actions (which are common in web based front-ends), require the introduction of thunks [6].
5. When Redux is used with React, additional patterns/libraries are required to optimize render performance (by reducing deep inspection of props during rendering) such as immutable objects, *shouldComponentUpdate* lifecycle hook usage and reselectors.

2. Principles and assumptions

The proposed architecture is based on the following principles and assumptions. It is our experience that these principles and assumptions apply primarily to large, complex front-end applications that need to be maintained over a long period of time by a relatively large number of developers. The reader's experiences may vary.

1. Code comprehensibility is the primary problem present-day front-end architectures attempt to solve.
2. While back-end systems have successfully employed layered architecture (which address vertical complexity) and modularization (which address horizontal complexity) to reduce code complexity, the same architectural solutions have had limited success in addressing the code complexity of front-end applications.

3. Front-ends applications are event-driven systems with real-time constraints, running on non-scalable infrastructure (a user's machine), within a single runtime environment (the browser). These factors contribute to the difficulties of successfully applying traditional back-end architectural solutions to front-ends [7].
4. It is better to address code complexity by localizing code (within functions and directories) rather than by introducing design patterns or abstractions.
5. It is better to organize code based on business scope (e.g. users, orders) rather than separation of concerns (e.g. actions, stores, reducers).
6. It is better to minimize interactions between modules. While complexity grows linearly with module size, it grows geometrically with the number of interactions between modules.
7. It is better to communicate between modules by transmitting events than by calling functions (thereby treating modules as black boxes).
8. It is better to limit reactive data binding to the view and view-model layers only.
9. Every state variable within a view-model must have a one-to-one correspondence to something displayed on-screen.
10. In a blocking environment (such as a single-threaded front-end), return values should only be employed by read-only API functions. API functions that modify state or reach out to data sources outside the module, should not return values, but instead employ *events* to transmit results back to the caller (this is a slight departure for the frequently used callback and promise patterns).
11. Business logic must be strictly decoupled from the front-end code.
12. Rely on standardized technologies (e.g. WHATWG approved JS/HTML/CSS APIs) whenever possible, resorting to third party libraries and trademarked products only when necessary.

3. Event-emitting models

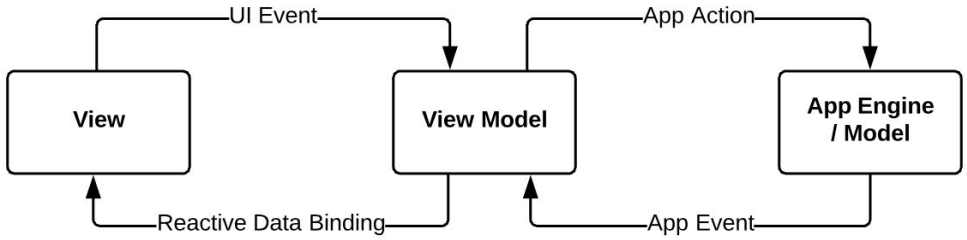

Figure 3. A depiction of the proposed architecture.

The primary difference between the Redux architecture and the proposed architecture is the replacement of the "store" with a model that encapsulates both the business state and business logic. This may alternatively be termed an "application engine".

This "engine" encapsulates all business data and logic, and is completely devoid of presentational concerns, in that it can be integrated into any type of application that requires the business logic in question: GUIs, CLIs, batch processes etc.

Based on the principles outlined in the previous section, the engine will expose a well-defined API to the outside world: a set of getter functions that immediately return local state within the engine, a set of "action" functions that set off operations within the model but do not immediately return anything. At the completion of such operations, the engine emits events, to which the callers of the API must listen.

Here is a partial source code listing of a hypothetical to-do application engine:

```
class TodoEngine {
  addEventListener(listener) {
    this.listeners.push(listener);
  }

  createTodo(summary) {
    let todo = new Todo(summary);
    this.todos.push(todo);
    this.emit(CREATE_TODO, todo); // operations emit events, rather than return values
  }


  findTodoById(id) {
    return this.todos.find({id: id}); // getters can immediately return values
  }

  completeTodo(id) {
    let todo = this.findTodoById(id);
    if (!todo) {
      this.emit(COMPLETE_TODO, {error: 'No entry by that id'});
      return;
    }

    todo.status = COMPLETED;
    this.emit(COMPLETE_TODO, todo);
  }
}
```

4. Event-listening view models

The second difference in the proposed architecture is that the view does not react to a central store, but only to its own view model. The view model in turn listens to events emitted by the model/engine and updates its own state. In the following example, we have used the state of a root level container component written using React as the view-model:

```
const engine = require("todo-engine"); // a singleton module

class TodoView extends React.Component {
  constructor(props) {
    super(props)
  }

  componentDidMount() {
    engine.addEventListener(this.onTodoEngineEvent);
  }

  componentWillUnmount() {
    engine.removeEventListener(this.onTodoEngineEvent);
  }

  onClickCreateTodoButton() {
    engine.createTodo(this.state.todoInputText);
  }

  onTodoEngineEvent(e) {
    switch (e.type) {
```

```
      case CREATE_TODO:
        if (e.error) {
          this.showMessage('Failed to create todo: ' + e.message);
          return;
        }

        this.setState({todos: this.engine.getTodos()});
      break;
    }
  }
}
```

Note how the view-model responds to view events (e.g. clicking the create todo button) by simply calling the corresponding engine function *without expecting a return value, a callback or a promise resolution*. The view model will instead consider that point the end of that particular interaction, and will expect the corresponding CREATE_TODO event to arrive at a later point. That event too, will be considered a separate interaction.

5. Using discrete events

The above examples use virtual events, in that an event emitting pattern is employed in code, but the caller and event are on the same callstack, within a single JavaScript event. One could argue that this can result in nested actions (the original problem the Flux architecture attempted to solve with a "dispatcher"). An example:

```
onTodoEngineEvent(e) {
    switch (e.type) {
        case CREATE_TODO:
            if (e.error) {
                this.showMessage('Failed to create todo: ' + e.message);
                return;
            }

            this.setState({ todos: this.engine.getTodos() });
            engine.notifyUser(e.data.user); // re-entering the engine
            break;
    }
}
```

Notice that the view's call to the engine's *createTodo()* will result in a callback to its *onTodoEngineEvent()*, which it turn will call the engine's *notifyUser()*, essentially re-entering the engine from within an engine call.

We can avoid this by converting engine calls into true events within the JavaScript engine. This can be done either by using *setTimeout* or *postMessage*.

```
createTodo(summary) {
    setTimeout(() => { // or any mechanism to defer execution into a separate event
        let todo = newTodo(summary);
        this.todos.push(todo);
        this.emit(CREATE_TODO, todo);
    });
}
```

This achieves almost the same level of decoupling between the view-model and the engine, as exists between the front-end and the BFF (back-end for front-end). However, it is not advisable to make getters (or read-only APIs) of the engine event driven. In the interest of performance, they can immediately return values to the caller.

6. Production usage

This architecture has been used in four production applications, including two applications that have been in production for close to two years. Three of the four applications perform business critical functions. In each application, the view layer was built with React, while the view model layer is the state of a root level “container component” [8]. The child components within the container components are mostly stateless. The engine is passed to child components that needed it via a singleton module import.

While the view and view-model layers of this architecture have scaled well as the applications grew in size, the engine often became unmanageably heavy. The solution to this problem was relatively trivial, and could be implemented with no effect to the view layer: the internal implementation of the engine was split into smaller event-driven modules, each handling specific functional areas (such as users, orders), while keeping the public API interface of the engine unchanged.

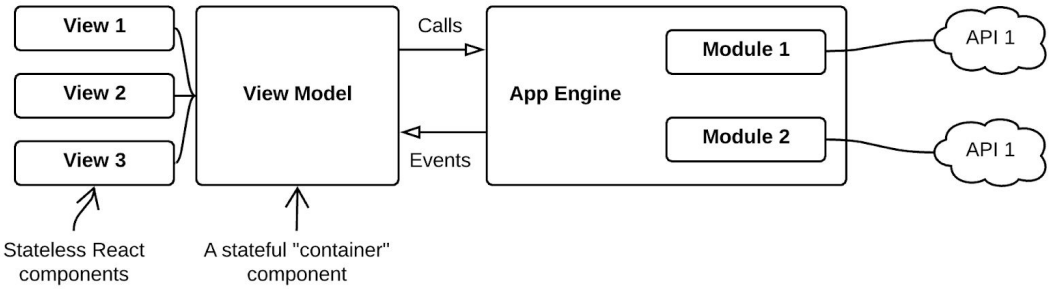

Figure 4. A depiction of the proposed architecture, with a modularized application engine.

7. Conclusion

The name "Transmission" comes from the fact that this architecture consists of two distinct loops, with communication between those two loops happening entirely via the transmission of events, rather than direct function calls.

The two one-way loops (as opposed to the original Flux architecture's single loop) are as follows:

1. View → UI event → View model → Reactive data binding → View
2. View model → Action → Application engine → Event → View model

The architecture affords the following advantages:

1. An application engine completely decoupled from the view layer. It can be independently developed, tested and reused in any context, including mobile and back-end environments.
2. A very thin, business-logic-free front-end layer that is completely decoupled from the business logic layer.
3. Lightweight, due to minimal use of third party libraries: immutability, react reselectors, action thunks and side-effects are largely optional.
4. Significantly reduced boilerplate code compared to Redux.
5. Business logic code is localized within the application engine (as opposed to being spread across actions, reducers and effects).
6. Ease of reasoning about execution flow that is either comparable or better than Redux, but without the added complications of third partly libraries, boilerplate code, runtime overhead and non-localized business logic code.
7. Minimal semantics: view (stateless react components), view-model (stateful react components, or "containers"), application engine, actions/operations and events. Does not feature semantics such as action creators, dispatchers, map-state-to-props/connect, reducers, side-effects.

Potential disadvantages:

1. As a relatively novel architecture, developer buy-in (relative to Redux) may be more time consuming.

References

[1] Redux architecture https://github.com/reduxjs/redux

[2] Flux architecture https://facebook.github.io/flux/docs/in-depth-overview

[3] Rethinking Web App Development at Facebook https://www.youtube.com/watch?v=nYkdrAPrdcw

[4] D. Abramov, You might not need redux
https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

[5] When should I use Redux? https://redux.js.org/faq/general#when-should-i-use-redux

[6] Thunk https://en.wikipedia.org/wiki/Thunk

[7] C. Jackson, Micro Frontends, https://martinfowler.com/articles/micro-frontends.html

[8] D. Abramov, Presentational and container components
https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0