An event-driven MVVM alternative to the Flux front-end architecture

Liyanage, H.N.
hasithaliyan@gmail.com

Abstract. Flux [1], while being a widely used architecture for applications built on the React UI component library, has demonstrated only limited success in battling the traditional complexities involved in front-end application development. Here we briefly look at some of the reasons for this, and propose an alternative event-driven MVVM (model-view-view-model) architecture.

1. Problems with Flux/Redux

A depiction of the Flux architecture

For the purposes of this paper, the reader may consider the popular Redux [2] library as a Flux variant. While Flux (and Redux) makes specific UI workflows easier to reason about, as the application grows larger the code bases become increasingly difficult to manage. The architecture proposed next is meant for developers of large front-end applications who experience the following difficulties:

1. Business logic cannot be neatly decoupled from presentation because the business logic code is not localized. It is split between actions and store/reducer code. Business logic code from a Flux application is often not readily reusable in other contexts.
2. Modularization cannot be performed neatly 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 Flux/Redux code base.

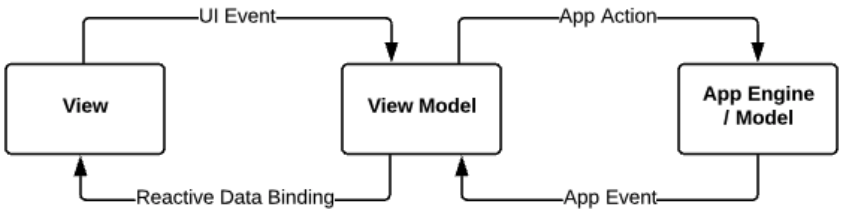
4. Handling asynchronous actions (which are common in web based front-ends), require the introduction of thunks [5].
5. When using React, store immutability is often required to improve render performance (by reducing deep inspection of props). At times, the `shouldComponentUpdate` lifecycle hook is also used.

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 experience 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 (addressing vertical complexity) and modularization (addressing 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 [6].
4. The best way to address complexity is not through the introduction of patterns, through scoping code, i.e. by ensuring that the code implementing a given feature is localized (to a function, a file or module) and do not interact with other parts of the code except through well-defined API calls.
5. Organizing code based on separation of concerns (e.g. actions, stores, reducers) rather than data and events (e.g. users, orders) reduces comprehensibility as the application grows more complex.
6. Complexity of an application grows linearly with the number of events, and geometrically with the interactions between those events.
7. Data flow between different layers of the application should always take place via well-defined APIs.
8. Reactive data binding (which is an implicit data flow mechanism between a view and its data) should be limited to the view-model (VM) layer only.
9. In a non-blocking environment (such as a single-threaded front-end), only those API functions that read the local state of a module should return values. API functions that modify the module state or reach out to data sources outside the module, should not return values, but instead employ events to return values to the caller.
10. Business logic must be strictly decoupled from the front-end code.

3. Event-emitting models



The proposed event-driven MVVM architecture. The View Model listens to UI events and calls the App Engine, and also listens to App Engine events and updates itself (resulting in View updates).

The primary difference between the popular Flux/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", and thought of in this manner:

Engine = Model + Controller – View

This model (or "engine") encapsulates all business data and logic and is completely devoid of presentational concerns (especially view models), in that it can be integrated into any type of application that requires the business logic concerned: GUIs, CLIs, batch processes etc.

Based on the principles outlined in the previous section, the model/engine will expose a well-defined API to the outside world: a set of getter functions that immediately return local state within the model, a set of "action" functions that set off operations within the model but do not return anything, and a set of events that are emitted at the end of operations, which the callers of the API must subscribe to.

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);
  }


  findTodoById(id) {
    return this.todos.find({id: id});
  }
```

```
  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. Here we have used the state of a root level container component written using React as an example:

```
class TodoView extends React.Component {
  constructor(props) {
    super(props)
    this.engine = props.engine;
  }

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

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

  onClickCreateTodoButton() {
    this.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, callback or promise resolution. The view model will instead consider that point the end of that particular event, and will expect the corresponding CREATE_TODO event to arrive at a later point. This too, will be considered a separate event.

5. 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 was the state of a root level "container component" [4]. The child components within the container components were mostly stateless. The engine was passed to child components that needed it via the React Context API.

While the view and view-model layers of this architecture have scaled well as the applications grew in size, the model/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 mini-engines, each handling specific functional areas (such as users, orders), while keeping the public API interface of the engine unchanged.

References

1. Flux architecture https://facebook.github.io/flux/docs/in-depth-overview
2. Redux architecture https://github.com/reduxjs/redux
3. You might not need redux https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367
4. Presentational and container components https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
5. Thunk https://en.wikipedia.org/wiki/Thunk
6. Micro Frontends, Jackson, C. https://martinfowler.com/articles/micro-frontends.html