Wrapping Up

This lesson brings all the little components we have learned in the previous lessons together, and presents the final product

WE'LL COVER THE FOLLOWING

Let's write a simple counter app that doesn't involve React so we see the pattern in action.

The markup

We'll need some UI to interact with it so:

```
<!-- Simple HTML document with `increase` and `decrease` buttons -->
html
<div id="counter">
  <span></span>
  <button>increase</button>
  <button>decrease</button>
</div>
```

The `span` will be used for displaying the current value of our counter. The buttons will change that value.

The view

```
const View = function (subscribeToStore, increase, decrease) {
  var value = null;
  var el = document.querySelector('#counter'); // selecting div by ID
  var display = el.querySelector('span'); // value displayed here
  var [ increaseBtn, decreaseBtn ] =

    Array.from(el.querySelectorAll('button'));

  var render = () => display.innerHTML = value; // render value
  var updateState = (store) => value = store.getValue(); // update state

  // subscribe state to store
  subscribeToStore([updateState, render]);

  // listen to button clicks
  increaseBtn.addEventListener('click', increase);
  decreaseBtn.addEventListener('click', decrease);
};
```

It accepts a store subscriber function and two action function for increasing and decreasing the value. The first few lines of the view are just fetching the DOM elements.

After that, we define a `render` function which puts the value inside the `span` tag. `updateState` will be called every time when the store changes. So, we pass these two functions to `subscribeToStore` because we want to get the view updated and we want to get an initial rendering. Remember how our consumers are called at least once by default.

The last bit is calling the action functions when we press the buttons.

The store

Every action has a type. It’s a good practice to create constants for these types so we don’t deal with raw strings.

```
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';
```

Very often we have only one instance of every store. For the sake of simplicity, we’ll create ours as a singleton.

```
// returns updated value on state change
const CounterStore = {
  _data: { value: 0 },
  getValue: function () {
    return this._data.value;
  },
  update: function (action, change) {
    if (action.type === INCREASE) {
      this._data.value += 1;
```

```
        this._data.value -= 1;
      }
      change();
    }
};
```

`_data` is the internal state of the store. `update` is the well known method that our dispatcher calls. We process the action inside and say `change()` when we are done. `getValue` is a public method used by the view so it reaches the needed info. In our case this is just the value of the counter.

Wiring all the pieces

So, we have the store waiting for actions from the dispatcher. We have the view defined. Let's create the store subscriber, the actions and run everything.

```
const { createAction, createSubscriber } = Fluxiny.create();
const counterStoreSubscriber = createSubscriber(CounterStore);
// create `increase`, `decrease` actions to cause a state change
const actions = {
  increase: createAction(INCREASE),
  decrease: createAction(DECREASE)
};

View(counterStoreSubscriber, actions.increase, actions.decrease);
```

And that's it. Our view is subscribed to the store and it renders by default because one of our consumers is actually the `render` method.

A live demo

A live demo could be seen in the following JSBin. If that's not enough and it seems too simple for you please checkout the example in Fluxiny repository . It uses React as a view layer.

Quick Quiz on Flux:

1 What is the hub for all the events in the Flux architecture?

COMPLETED 0%

1 of 3

And that is about it for Flux Architecture. Let's now move onto Redux.