Bounding Views and Stores

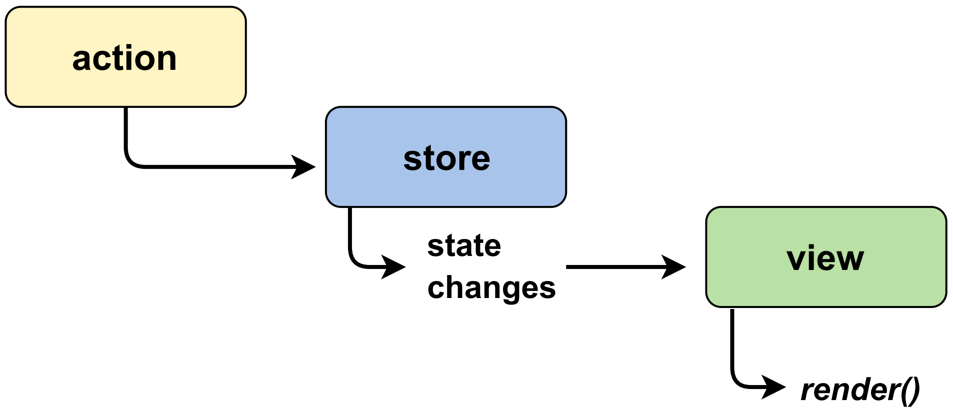
This lesson introduces multiple ways to connect views to stores

WE'LL COVER THE FOLLOWING

- Bounding the views and the stores
  - Using a helper
  - With a mixin
  - Using a context
  - Higher-Order components concept
  - Using Higher Order Components

Bounding the views and the stores

The next logical step is to connect our views to the stores so we re-render when the state in the stores is changed.



There are multiple ways to achieve this. Let's look at a few:

Using a helper

Some of the flux implementations available out there provide a helper function that does the job. For example:

```
Framework.attachToStore(view, store);
```

However, to make `attachToStore` work we expect to see a specific API in the view and in the store. We kind of strictly define new public methods. Or in other words, we say “Your views and store should have such APIs so we are able to wire them together”. If we go down this road then we will probably define our own base classes which could be extended so we don’t bother the developer with Flux details. Then we say “All your classes should extend our classes”. This doesn’t sound good either because the developer may decide to switch to another Flux provider and has to amend everything.

With a mixin

What if we use React’s mixins? Like this:

```
var View = React.createClass({
  mixins: [Framework.attachToStore(store)]
  ...
});
```

That’s a nice way to define the behavior of an existing React component. So, in theory, we may create a mixin that does the bounding for us. Mixins modify the components in a non-predictable way. The developer has no idea what is going on behind the scenes.

Using a context

Another technique that may answer the question is React’s context. It is a way to pass props to child components without the need to specify them in every level of the tree. Facebook suggests a context in the cases where we have data that has to reach deeply nested components.

> Occasionally, you want to pass data through the component tree without having to pass the props down manually at every level. React’s “context” feature lets you do this.

You might see a similarity with the mixins here. The context is defined somewhere at the top and magically serves props for all the children below. It's not immediately clear where the data comes from.

Higher-Order components concept

Higher-Order components pattern is introduced by Sebastian Markbåge and it is about wrapping a component and attaching some new functionalities or props to it. For example:

```
// creating a wrapper component that may send props
// and apply additional logic
function attachToStore(Component, store, consumer) {
  const Wrapper = React.createClass({
    getInitialState() {
      return consumer(this.props, store);
    },
    componentDidMount() {
      store.onChangeEvent(this._handleStoreChange);
    },
    componentWillUnmount() {
      store.offChangeEvent(this._handleStoreChange);
    },
    _handleStoreChange() {
      if (this.isMounted()) {
        this.setState(consumer(this.props, store));
      }
    },
    render() {
      return <Component {...this.props} {...this.state} />;
    }
  });
  return Wrapper;
};
```

`Component` is the view that we want to be attached to the `store`. The `consumer` function says what part of the store's state should be fetched and sent to the view. A simple usage of the above function could be:

```
class MyView extends React.Component {
  ...
}

ProfilePage = connectToStores(MyView, store, (props, store) => ({
  data: store.get('key')
}));
```

That is an interesting pattern because it shifts the responsibilities. It is the view fetching data from the store and not the store pushing something to the view. This of course has it's own pros and cons. It is nice because it makes the store dummy. A store that only mutates the data and says "Hey, my state is changed". It is not responsible for sending anything to anyone.

The downside of this approach is maybe the fact that we have one more component (the wrapper) involved. We also need the three things - view, store, and consumer to be in one place so we can establish the connection.

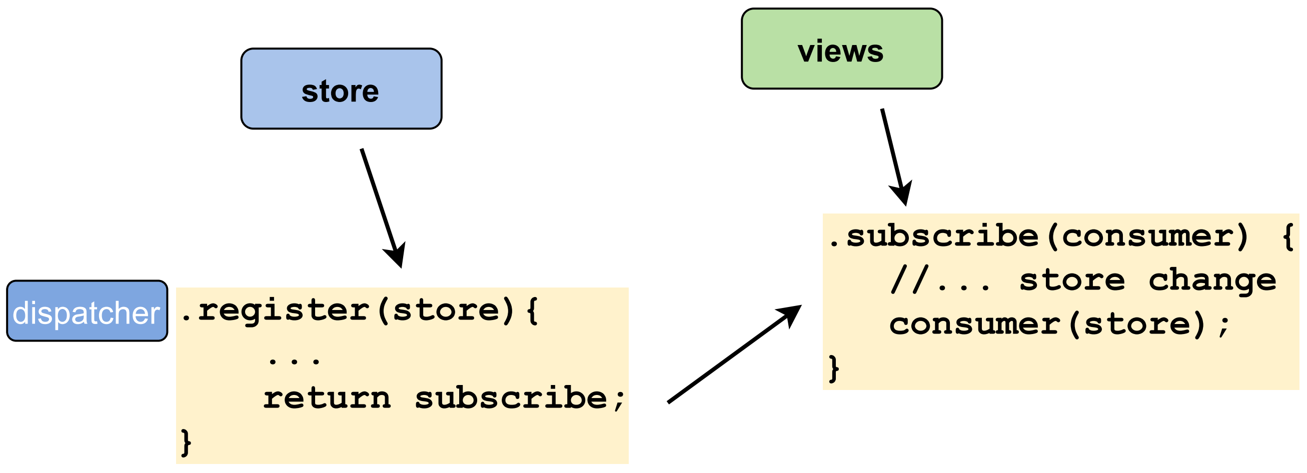
Using Higher Order Components

In higher-order components, the view decides what it needs. That *knowledge* anyway exists in the component so it makes sense to keep it there. That's also why the functions that generate higher-order components are usually kept in the same file as the view. What if we can use a similar approach but not passing the store at all. Or in other words, a function that accepts only the consumer. And that function is called every time when there is a change in the store.

So far our implementation interacts with the store only in the `register` method.

```
register: function (store) {
  if (!store || !store.update) {
    throw new Error('You should provide a store that has an `update` metho
d.');
  } else {
    this._stores.push({ store: store });
  }
}
```

By using `register` we keep a reference to the store inside the dispatcher. However, `register` returns nothing. And instead of nothing, it may return a **subscriber** that will accept our consumer functions.



Connect store and view

Let's say we decided to send the whole store to the consumer function and not the data that the store keeps. Like in the higher-order components pattern the view should say what it needs by using the store's getters. This makes the store really simple and there is no trace of presentational logic.

Here is what the register method looks like after the changes:

```
register: function (store) {
  // throw error if no update method exists
  if (!store || !store.update) {
    throw new Error(
      'You should provide a store that has an `update` method.'
    );
  } else {
    var consumers = [];
    var subscribe = function (consumer) {
      consumers.push(consumer);
    };

    this._stores.push({ store: store });
    return subscribe;
  }
  return false;
}
```

The last bit in the story is how the store says that its internal state is changed. It's nice that we collect the consumer functions but right now there is no code that executes them.

According to the basic principles of the flux architecture the stores change their state in response of actions. In the `update` method we send the `action` but we could also send a function `change`. Calling that function should trigger the consumers:

```
register: function (store) {
  // throw error if no update method exists
  if (!store || !store.update) {
    throw new Error(
      'You should provide a store that has an `update` method.'
    );
  } else {
    var consumers = [];
    var change = function () {
      consumers.forEach(function (consumer) {
        consumer(store);
      });
    };
    var subscribe = function (consumer) {
      consumers.push(consumer);
    };

    this._stores.push({ store: store, change: change });
    return subscribe;
  }
  return false;
},
dispatch: function (action) {
  if (this._stores.length > 0) {
    this._stores.forEach(function (entry) {
      entry.store.update(action, entry.change);
    });
  }
}
```

Notice how we push `change` together with `store` inside the `_stores` array. Later in the `dispatch` method, we call `update` by passing the `action` and the `change` function.

A common use case is to render the view with the initial state of the store. In the context of our implementation, this means firing all the consumers at least once when they land in the library. This could be easily done in the `subscribe` method:

```
var subscribe = function (consumer, noInit) {
  consumers.push(consumer);
  !noInit ? consumer(store) : null;
};
```

Of course, sometimes this is not needed so we added a flag which is by default false. Here is the final version of our dispatcher:

```
var Dispatcher = function () {
  return {

    _stores: [],
    register: function (store) {
      // throw error if no update method exists
      if (!store || !store.update) {
        throw new Error(
          'You should provide a store that has an `update` method'
        );
      } else {
        var consumers = [];
        var change = function () {
          consumers.forEach(function (consumer) {
            consumer(store);
          });
        };
        var subscribe = function (consumer, noInit) {
          consumers.push(consumer);
          !noInit ? consumer(store) : null;
        };

        this._stores.push({ store: store, change: change });
        return subscribe;
      }
      return false;
    },
    // check all stores for update
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action, entry.change);
        });
      }
    }
  }
};
```

We will continue to build upon this in the following section to produce a complete working Flex architecture.