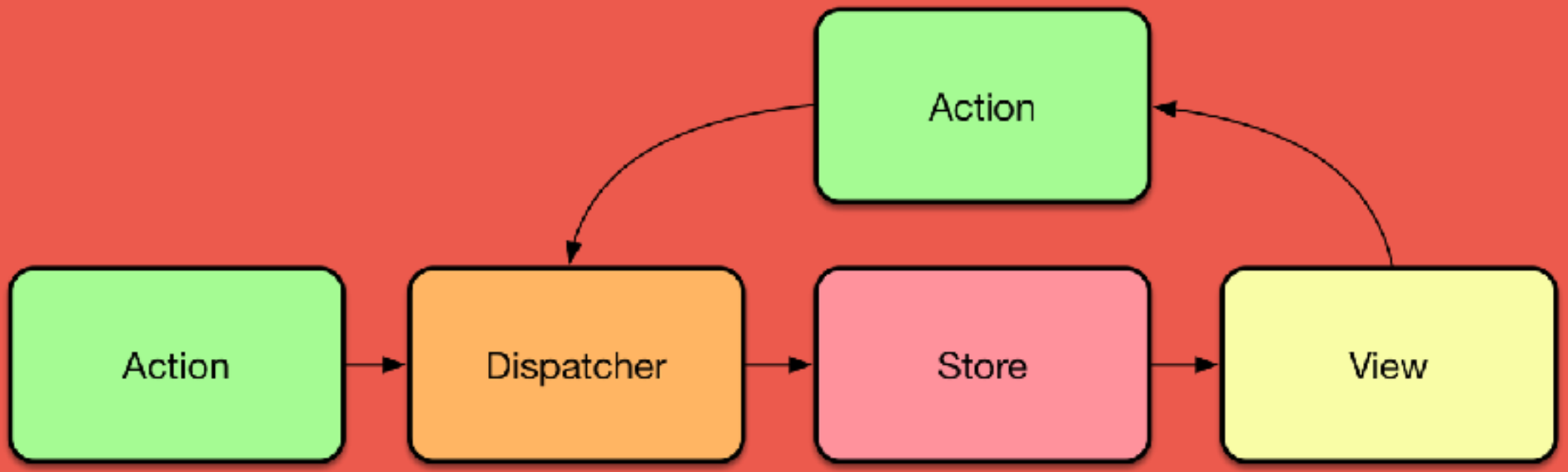
Flux

something you need
to learn react native

FLUX

what flux is

- application architecture associated with react
- based on unidirectional flow of data
- developed by Facebook
- a github repo with useful javascript utilities



Actions

ACTIONS

what actions are

- objects with data relevant to the application state
- has a type and data

```
{
  actionType: 'SELECTED_COUNTRY',
  selectedCountry: 'Mexico'
};
```

ACTIONS

action creators create actions

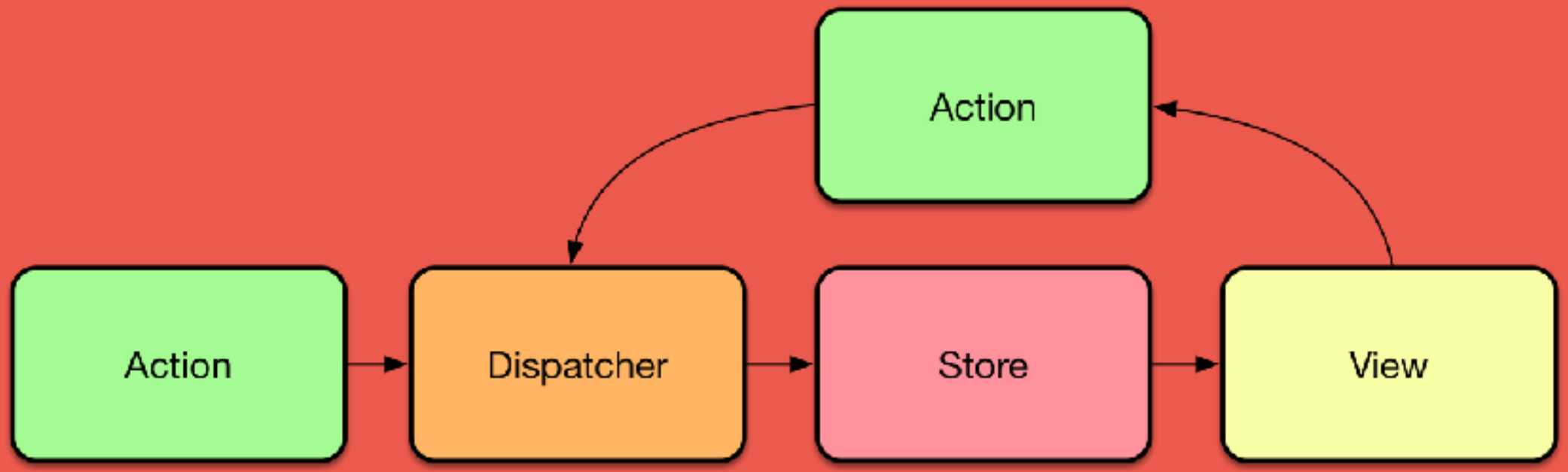
- they describe actions taken
- they are not setters

```
function selectedCountry(country) {
  return {
    actionType: 'SELECTED_COUNTRY',
    selectedCountry: country
  };
}
```

ACTIONS

when actions happen

- triggered by user interaction
- during data initialization
- when the server has updates to provide the application (web socket)



Dispatcher

DISPATCHER

what a dispatcher is / does

- there is only one dispatcher in application
  - it’s a singleton
- dispatches actions to registered callbacks

```
const action = CountryActions.selectedCountry('Mexico');
AppDispatcher.dispatch(action);
```

DISPATCHER

you don’t have to roll your own, but you may want to extend one

```
import { Dispatcher } from 'flux';

class AppDispatcher extends Dispatcher {
  dispatch(action) {
    // log each action that is dispatched
    console.log(action);
    super.dispatch(action);
  }
}

export default new AppDispatcher();
```

DISPATCHER

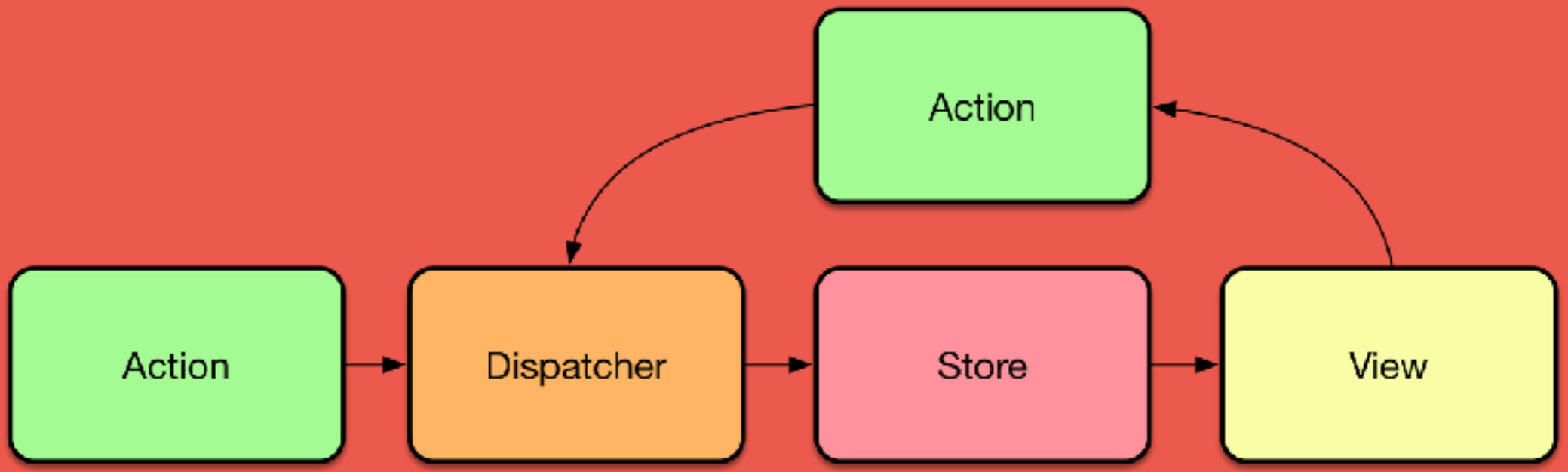
different than pub-sub

- callbacks are not subscribed to particular actions
- callbacks can defer execution (more on that later)

DISPATCHER

sample API (taken from flux github repo)

- register(function callback) - returns a token
- unregister(string token) - void
- dispatch(action) - void
- isDispatching() - returns boolean



Store

STORE

what a store does

- manages application state and logic
- each store is a singleton
- each store manages a particular domain
  - you control structure of store’s state
  - (example on following slide)

STORE

here the store’s state is an object, it could just as easily be a string, an array, or anything

```
getInitialState() {
  return Object.freeze({
    selectedCountry: 'Canada'
  });
}
```

STORE

stores digest actions from the dispatcher

- registers a function with dispatcher
- function changes state in response to actions
  - uses switch statement
- store state **only** changes within that function
- (example on following slide)

STORE

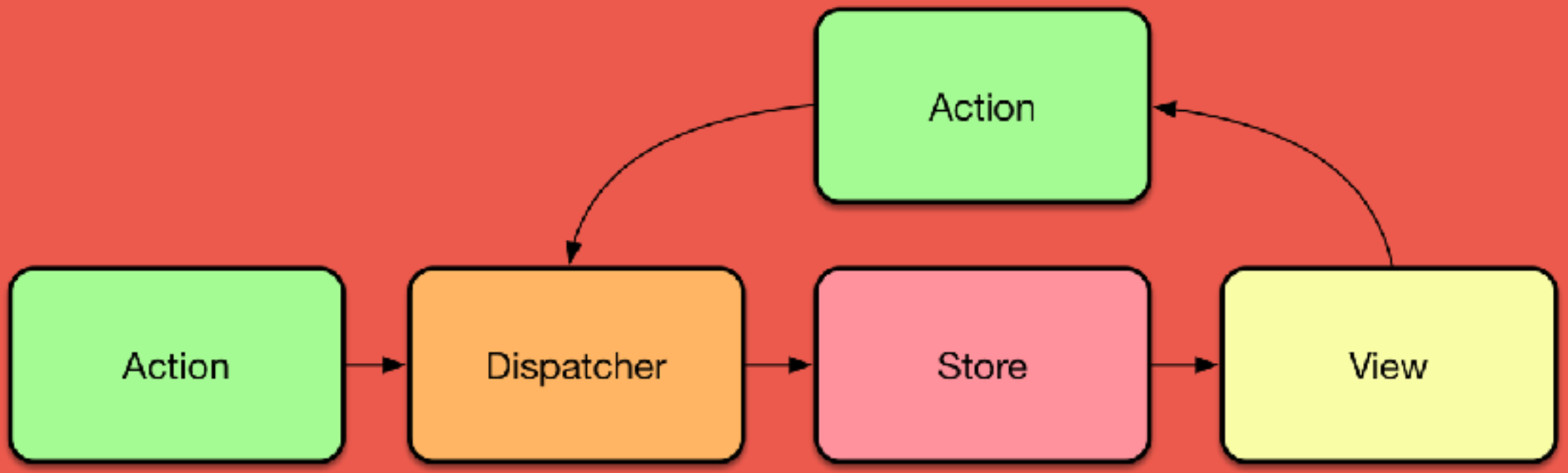
this function gets an action from the dispatcher and returns the new state

```
reduce(state, action) {
  switch (action.actionType) {
    case ActionTypes.SELECTED_COUNTRY:
      return Object.freeze({
        selectedCountry: action.selectedCountry
      });
    default:
      return state;
  }
}
```

STORE

stores are information resources for views

- a store emits a *change* event when its state changes (more on that later)
- offers public getters for data (never setters)



Views

VIEWS

what views do

- present user with info and interface
- ex: react components (out of scope for this presentation)

VIEWS

views display info from stores

- views subscribe to updates from a store
- views change state in response to the *change* events that a store emits
- (example on following slide)

VIEWS

```
componentDidMount() {
  this.countryStoreToken = CountryStore.addListener(
    () => {
      this.setState({
        country: CountryStore.getState().selectedCountry
      });
    }
  );
}

componentWillUnmount() {
  this.countryStoreToken.remove();
}
```

VIEWS

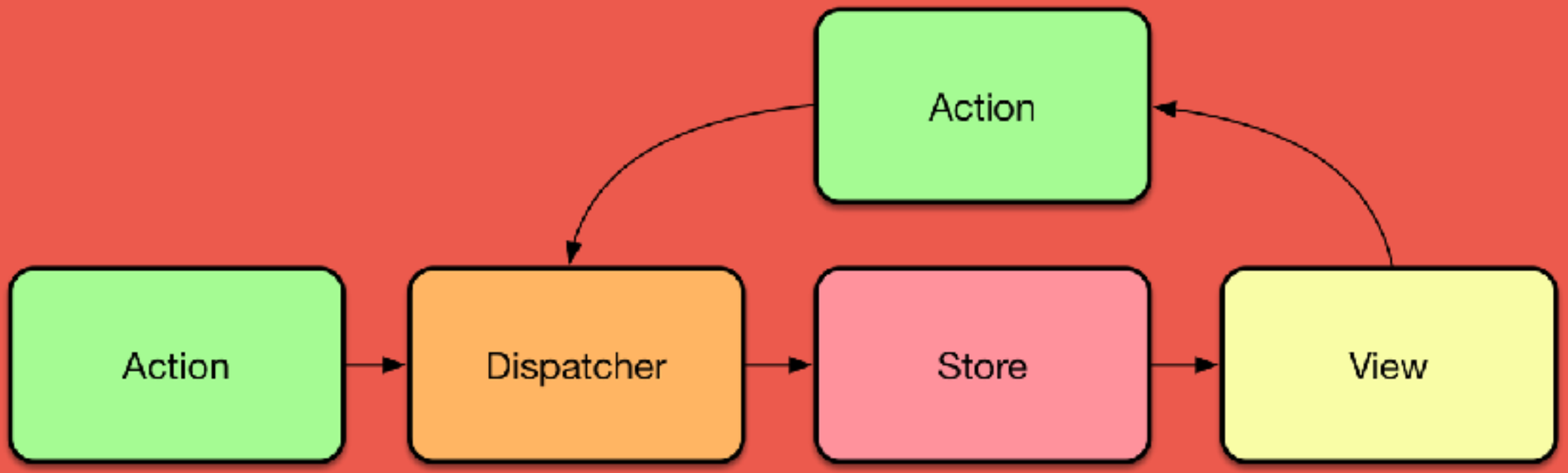
views can trigger actions

- bind functions to user actions
- invoke dispatcher with action
- (example on following slide)

VIEWS

```
changeCountry(country) {
  AppDispatcher.dispatch(CountryActions.selectedCountry(country));
}

render() {
  return (
    <Picker
      selectedValue={this.state.country}
      onValueChange={this.changeCountry}
    >
      <Picker.Item label="Mexico" value="Mexico" />
      <Picker.Item label="Canada" value="Canada" />
    </Picker>
  );
}
```



Back to Stores and Dispatchers

DEFERRING

stores can defer responding to an action

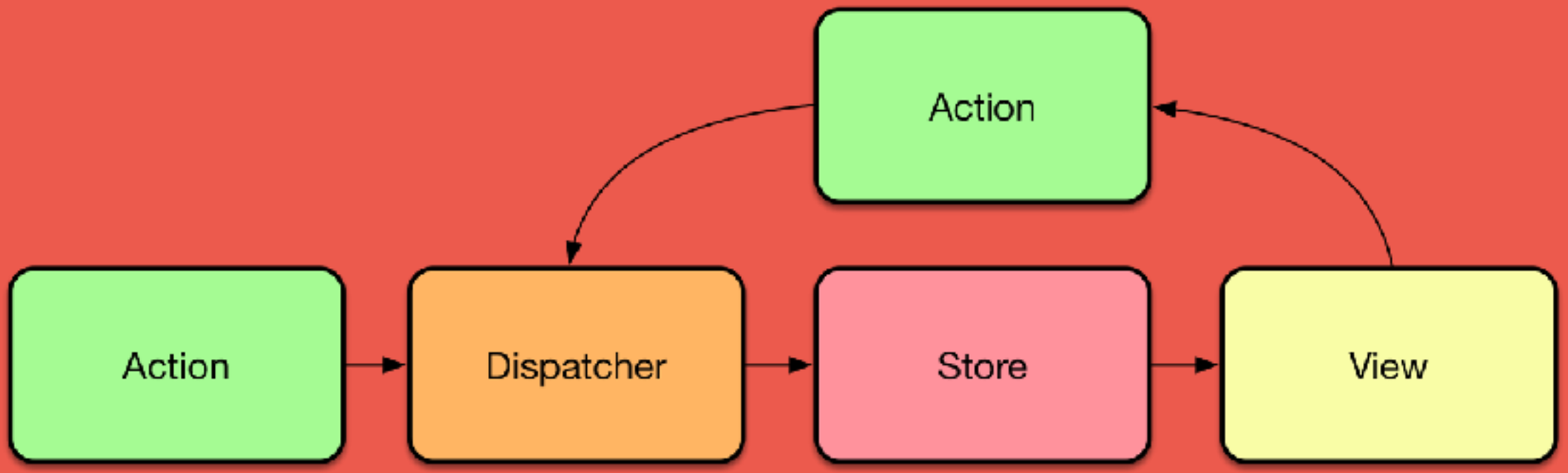
- one store may have a state that depends on another store
- this is why there must only be one dispatcher
  - dispatcher `waitFor` method allows it to manage execution order

DEFERRING

example: CityStore stores a list of cities based on the current country selected

```
reduce(state, action) {
  switch (action.actionType) {
    case ActionTypes.SELECTED_COUNTRY:
      const countryStoreToken = CountryStore.getDispatchToken();
      AppDispatcher.waitFor([countryStoreToken]);

      const selectedCountry = CountryStore.getState().selectedCountry;
      return Object.freeze({
        cities: getCitiesForCountry(selectedCountry),
      });
    default:
      return state;
  }
}
```



Takeaways

TAKEAWAYS

- unidirectional data flow (the chart)
- actions are dispatched
  - in views from user interaction
  - during data initialization
  - … and in other situations
- stores respond to actions from dispatcher
- views listen to stores emitting changes

thank you

rzwdevclub 4ever