IMMUTABLE.JS

АРХИТЕКТУРА FLUX

LUXOFT TRAINING

LXFT LISTED NYSE

ИММУТАБЕЛЬНОСТЬ

Последнее, что бы мы хотели, чтобы делал программист – это путался в меняющемся внутреннем состоянии, говоря фигурально. Очень жаль, что то, что мы называем "объектно-ориентированным программированием" – это обычное процедурное программирование с более современными конструкциями.

Алан Кей. "Ранняя история Смоллток"

IMMUTABLE.JS

Иммутабельность можно поддерживать стандартными средствами JavaScript. Однако, это требует внимательности и дисциплины. Facebook разработала библиотеку, которая позволяет использовать иммутабельность с гарантией, что все изменения будут иммутабельными.

Immutable.js предоставляет множество постоянных неизменяемых структур данных, включая: List, Stack, Map, OrderedMap, Set, OrderedSet и Record.

IMMUTABLE.JS

Существенную трудность при разработке приложений представляют **отслеживание изменений** и **поддержание состояния**. Разработка с использованием неизменяемых данных переворачивает наше представление о потоках данных в приложении.

```
<script src="immutable.min.js"></script>
<script>
  var map1 = Immutable.Map({a:1, b:2, c:3});
  var map2 = map1.set('b', 50);
  map1.get('b'); // 2
  map2.get('b'); // 50
 const map3 = map1.set('b', 50);
  assert(map1.equals(map3) === false);
</script>
```

Неизменяемые коллекции должны рассматриваться не как **объекты**, а как **значения**. Если **объект** представляет некий предмет, который может **меняться с течением времени**, то **значение** представляет **состояние** этого предмета в **определенный момент**.

СОЗДАНИЕ КОПИИ

Если объект неизменяемый, его можно «скопировать», просто сделав еще оду ссылку на него, вместо того, чтобы копировать весь объект. Поскольку ссылка намного меньше, чем сам объект, использование ссылок позволяет экономить память и увеличивает скорость выполнения программ, в которых широко используются копии (например, стек отмены).

```
let map1 = Map({ a: 1, b: 2, c: 3 })
let clone = map1;
```

НЕИЗМЕНЯЕМЫЕ КОЛЛЕКЦИИ

Неизменяемые коллекции **не изменяют** коллекцию с помощью таких методов, как **push**, **set**, **unshift** или **splice**. Вместо этого, она **возвращает новую неизменяемую** коллекцию.

```
const { List } = require('immutable')
const list1 = List([ 1, 2 ]);
const list2 = list1.push(3, 4, 5);
const list3 = list2.unshift(0);
const list4 = list1.concat(list2, list3);
assert(list1.size === 2);
assert(list2.size === 5);
assert(list3.size === 6);
assert(list4.size === 13);
assert(list4.get(0) === 1);
```

Практически все методы Array находятся на Immutable.List, методы Map — на Immutable.Map, а методы Set — на Immutable.Set:

```
const { Map } = require('immutable')
const alpha = Map({ a: 1, b: 2, c: 3, d: 4 });
alpha.map((v, k) => k.toUpperCase()).join();
```

РАБОТА С ОБЪЕКТАМИ JAVASCRIPT

Предназначенная для работы с JavaScript библиотека Immutable.js принимает простые массивы и объекты JavaScript там, где метод предполагает наличие коллекции:

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3, d: 4 })
const map2 = Map({ c: 10, a: 20, t: 30 })
const obj = { d: 100, o: 200, g: 300 }
const map3 = map1.merge(map2, obj);
// Map { a: 20, b: 2, c: 10, d: 100, t: 30, o: 200, g: 300 }
```

При использовании объектов JS для конструирования Maps свойства JS Object всегда являются строками:

```
const { fromJS } = require('immutable’)
const obj = { 1: "one" }
Object.keys(obj) // [ "1" ]
obj["1"] // "one"
obj[1]   // "one”

const map = fromJS(obj)
map.get("1") // "one"
map.get(1)   // undefined
```

ОБРАТНАЯ КОНВЕРТАЦИЯ В JAVASCRIPT

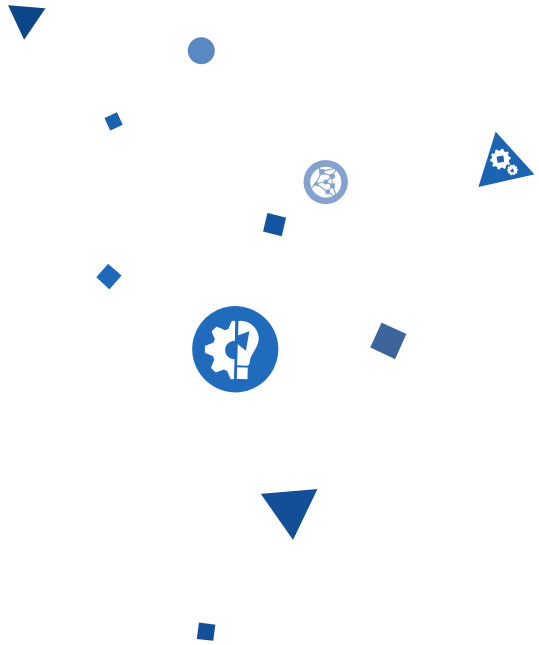
Все коллекции Immutable.js можно конвертировать обратно в обычные массивы и объекты JavaScript неглубоко с помощью **toArray()** и **toObject()** или глубоко с помощью **toJS()**. Кроме того, все неизменяемые коллекции реализуют метод **toJSON()**.

```
const { Map, List } = require('immutable')

const deep = Map({ a: 1, b: 2, c: List([ 3, 4, 5 ]) })

deep.toObject() // { a: 1, b: 2, c: List [ 3, 4, 5 ] }
deep.toArray() // [ 1, 2, List [ 3, 4, 5 ] ]
deep.toJS() // { a: 1, b: 2, c: [ 3, 4, 5 ] }

JSON.stringify(deep) // '{"a":1,"b":2,"c":[3,4,5]}'
```

ВЛОЖЕННЫЕ СТРУКТУРЫ

Коллекции в Immutable.js изначально являются вложенными, что дает возможность использовать глубокие древовидные структуры данных, аналогичные используемым в JSON.

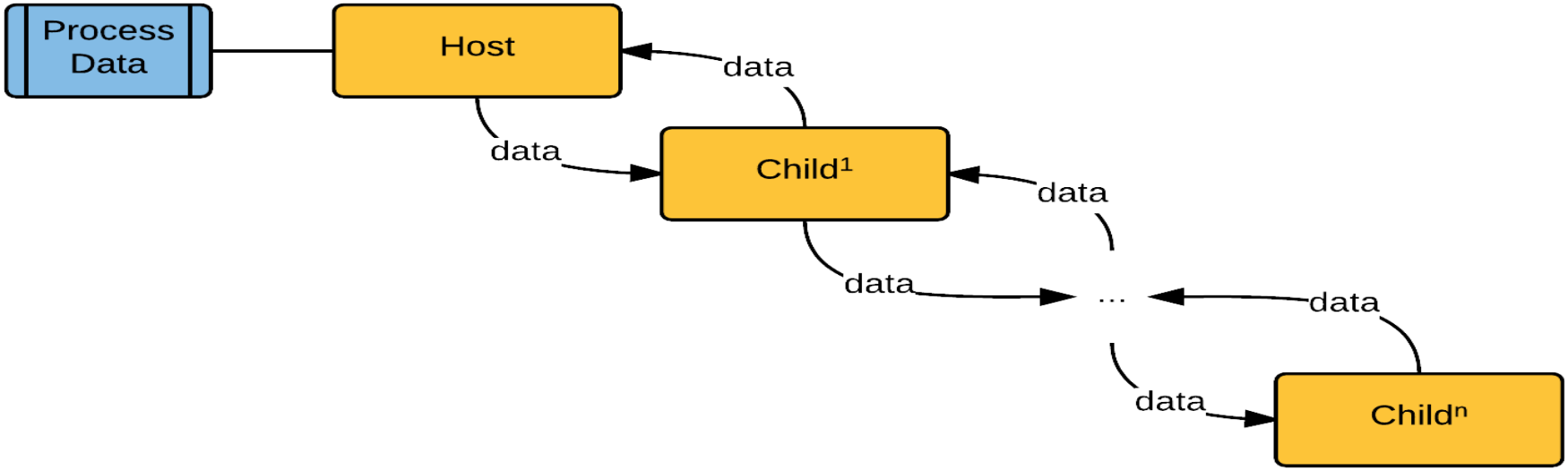
```
const nested = fromJS({ a: { b: { c: [ 3, 4, 5 ] } } })
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ] } } }

const nested2 = nested.mergeDeep({ a: { b: { d: 6 } } })
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 6 } } }

nested2.getIn([ 'a', 'b', 'd' ]) // 6

const original = { x: { y: { z: 123 }}}
const res = setIn(original, ['x', 'y', 'z'], 456) // res == { x: { y: { z: 456 }}}

const nested3 = nested2.updateIn([ 'a', 'b', 'd' ], value => value + 1)
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 7 } } }

const nested4 = nested3.updateIn([ 'a', 'b', 'c' ], list => list.push(6))
// Map { a: Map { b: Map { c: List [ 3, 4, 5, 6 ], d: 7 } } }
```



АРХИТЕКТУРА FLUX

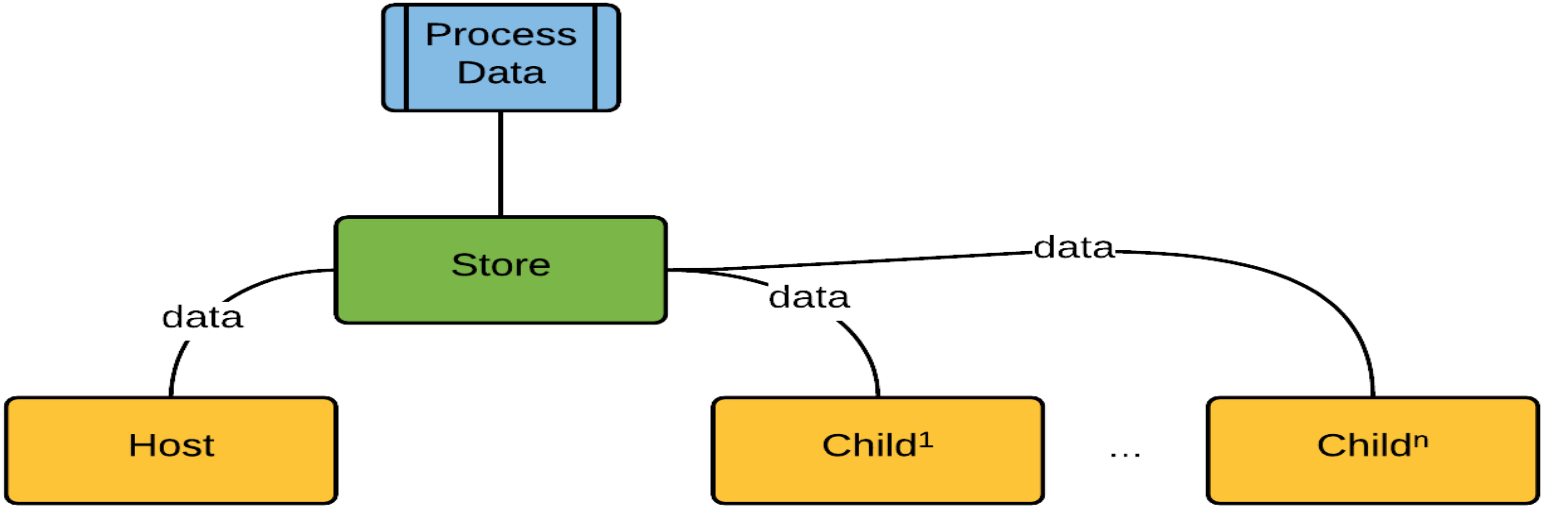
ТРАДИЦИОННАЯ ПРИВЯЗКА ДАННЫХ

Традиционная привязка данных (однонаправленная или двунаправленная) приводит к созданию цепи повторной передачи данных вниз или вверх по дереву компонентов:



STORE (МАГАЗИН)

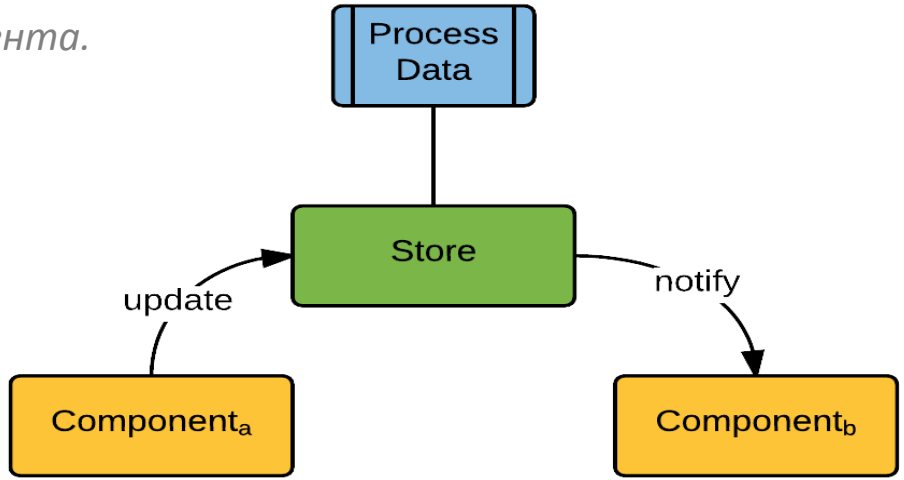
Store поддерживает одну или несколько моделей, поэтому и называется Store (магазин), а не модель.



Данные, хранящиеся в store, представляют **состояние приложения**. Состояние приложения можно представить как **все, что должно быть прямо или косвенно визуализировано**.

ИСПОЛЬЗОВАНИЕ STORE КАК ХРАНИЛИЩЕ ОБЩИХ ДАННЫХ

```
// Компоненты, которые подключены к данному store.
let components = fromJS([]);

// Собственно store состояний, в котором хранятся данные приложения.
let store = fromJS({});

// Определяет состояние store, затем —
// состояние каждого подключенного компонента.
export function setState(state) {
  store = state;
  for (const component of components) {
    component.setState({
      data: store,
    });
  }
}
// Возвращает состояние store.
export function getState() {
  return store;
}
```

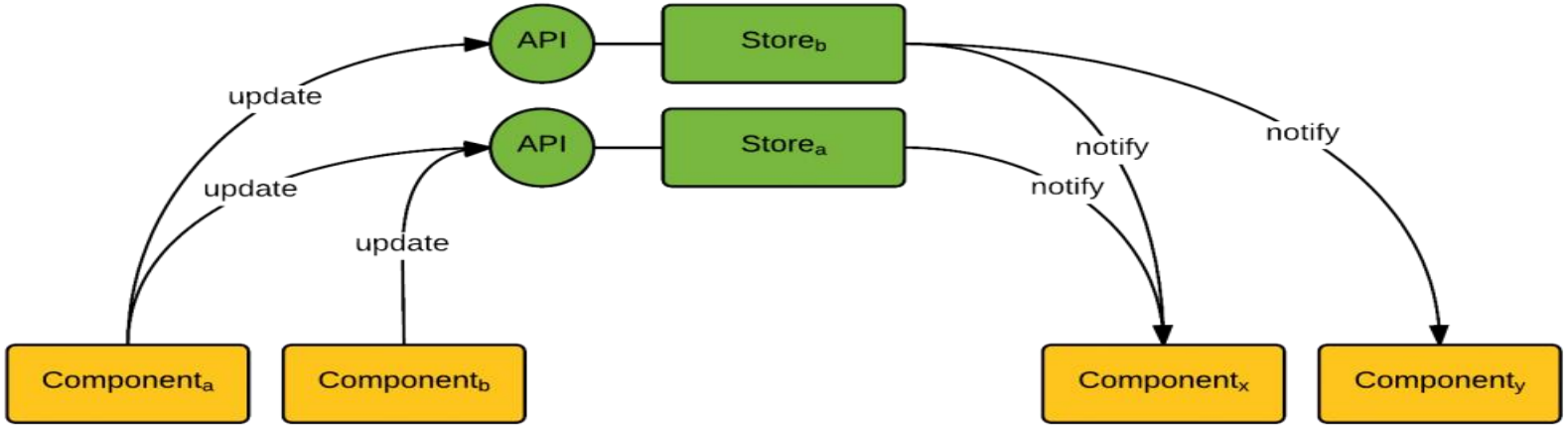


ФУНКЦИЯ CONNECT ДЛЯ ПОДКЛЮЧЕНИЯ КОМПОНЕНТА К STORE

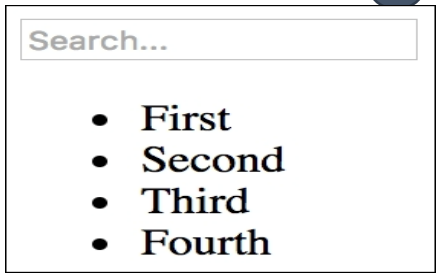
```
// Возвращает компонент более высокого порядка, который подключен к "store".
export function connect(ComposedComponent) {
  return class ConnectedComponent extends Component {
    state = { data: store }
    // Когда компонент подключен, добавьте его в "components", чтобы он
    // получал обновления при изменении состояния store.
    componentWillMount() {
      components = components.push(this);
    }
    // Удаляет этот компонент из "components", когда он исключается из DOM
    componentWillUnmount() {
      const index = components.findIndex(this);
      components = components.delete(index);
    }
    // Отображает "ComposedComponent", используя в качество свойств состояние "store".
    render() {
      return (<ComposedComponent {...this.state.data.toJS()} />);
    }
  }; }
```

STORES

Обычно Store предоставляет набор методов, например add(), update(), delete(), для обновления своих моделей. Поэтому каждый Store может иметь собственный интерфейс.



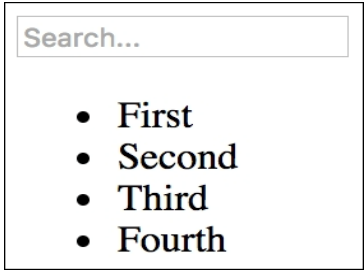
ПРИМЕР: СПИСОК С ФИЛЬТРОМ




```
// Отображает простой элемент ввода для
фильтрации списка.
const MyInput = ({ value, placeholder }) => (
  <input
    autoFocus
     value={value}
    placeholder={placeholder}
    onChange={onChange}
  />
);

MyInput.propTypes = {
  value: PropTypes.string,
  placeholder: PropTypes.string,
};
```

```
// Отображает список элементов...
const MyList = ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    ))}
  </ul>
);

MyList.propTypes = {
  items:
    PropTypes.array.isRequired,
};
```

РЕАЛИЗАЦИЯ ONCHANGE




```
// Когда изменяется входящее значение фильтра.
function onChange(e) {
  const state = getState(); // Состояние, с которым мы работаем...
  const items = state.get('items');
  const tempItems = state.get('tempItems');
  // Новое состояние, которое мы устанавливаем в store.
  let newItems;
  let newTempItems;
  if (e.target.value.length === 0) { // Если входящее значение пустое, его можно восстановить из tempItems
    newItems = tempItems;
    newTempItems = fromJS([]);
  } else {
    if (tempItems.isEmpty()) newTempItems = items;
    else newTempItems = tempItems;
    // Фильт и установка "newItems".
    const filter = new RegExp(e.target.value, 'i');
    newItems = items.filter(i => filter.test(i));
  }
```

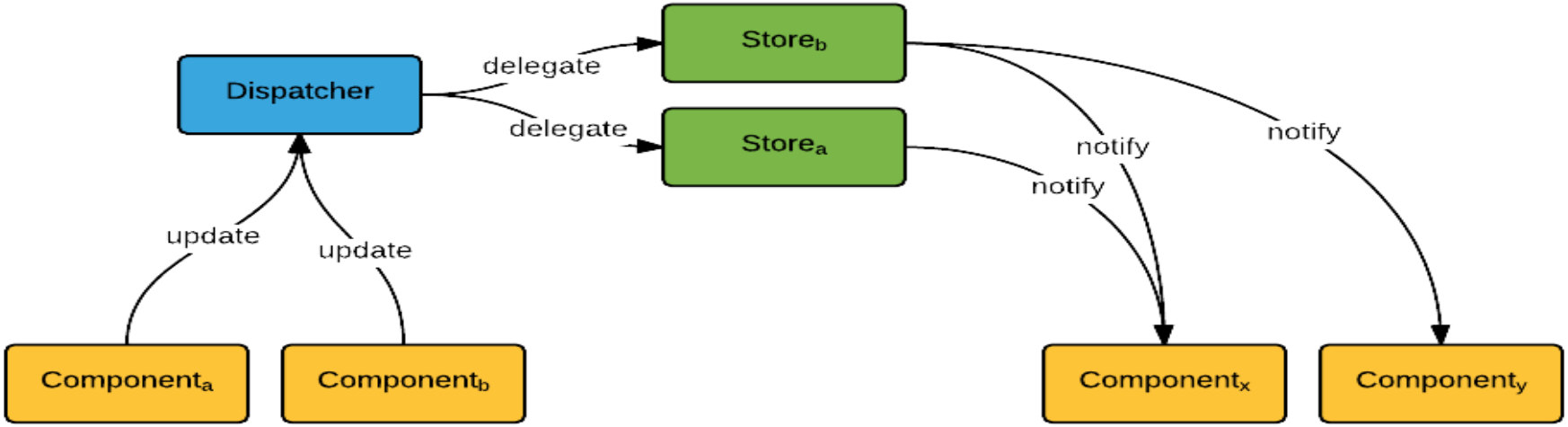
```
// Обновление состояния store.
setState(state.merge({
  items: newItems,
  tempItems: newTempItems,
}));
```

ОТОБРАЖЕНИЕ

```
// Создаем "подключенные" версии "MyInput" и
// "MyList", чтобы они автоматически получали обновления,
// когда store меняет состояние.
const ConnectedInput = connect(MyInput);
const ConnectedList = connect(MyList);
// Устанавливаем состояние store по умолчанию...
setState(getState().merge({
  placeholder: 'Search...',
  items: ['First', 'Second', 'Third', 'Fourth'],
  tempItems: [],
}));
render((
    <section>
      <ConnectedInput />
      <ConnectedList />
    </section>
  ), document.getElementById('app')
);
```

ИСПОЛЬЗОВАНИЕ DISPATCHER

Dispatcher делегирует/передает действия по обновлению в соответствующий Store.



DISPATCHER

```
let appDispatcher = new Dispatcher();

export function filterList( value ) {
  let payload = {
    type: 'FILTER_LIST',
    filter: value
  }
  appDispatcher.dispatch(payload);
}
```

in MyInput.js:

```
// Когда изменяется входящее значение
фильтра.
function onChange(e) {
  filterList(e.target.value);
}
```

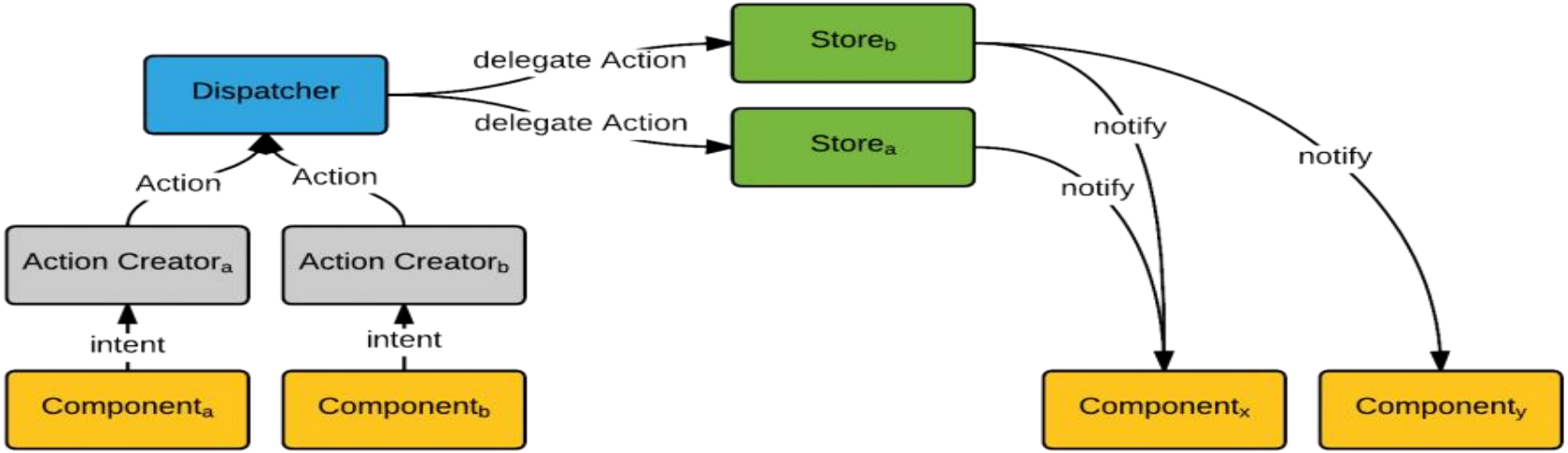
```
export class Dispatcher {
  callbacks = {};
  lastId = 0;
  register(callback) {
    let id = this.lastId;
    this.lastId++;
    this.callbacks[id] = callback;
  }
  unregister(id) {
    delete this.callbacks[id];
  }
  dispatch(payload) {
    for (let id in this.callbacks) {
      let callback =
                          this.callbacks[id];
      callback(payload);
    }
  }
}
```

ОБРАТНЫЙ ВЫЗОВ DISPATCHER ДЛЯ ОБРАБОТКИ FILTER_LIST

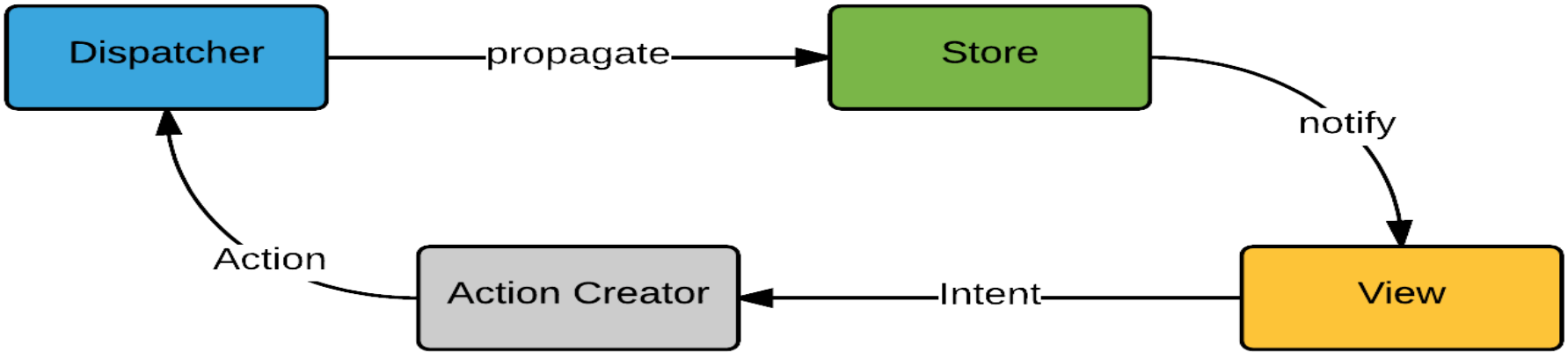
```
appDispatcher.register(function(payload) {
  switch(payload.type) {
    case "FILTER_LIST":
      const state = getState();
      const items = state.get('items');
      const tempItems = state.get('tempItems');
            let newItems, newTempItems;

      if (payload.filter.length === 0) { // восстановление из tempItems
        newItems = tempItems;
        newTempItems = fromJS([]);
       } else {
        if (tempItems.isEmpty()) newTempItems = items;
        else newTempItems = tempItems;
        // Применяем фильтр и устанавливает "newItems".
        const filter = new RegExp(payload.filter, 'i');
        newItems = items.filter(i => filter.test(i));
       }
            setState(getState().merge({ // Обновляет состояние store
        items: newItems,
        tempItems: newTempItems,
       }));
      break;
}});
```

ДЕЙСТВИЯ И ГЕНЕРАТОР ДЕЙСТВИЙ

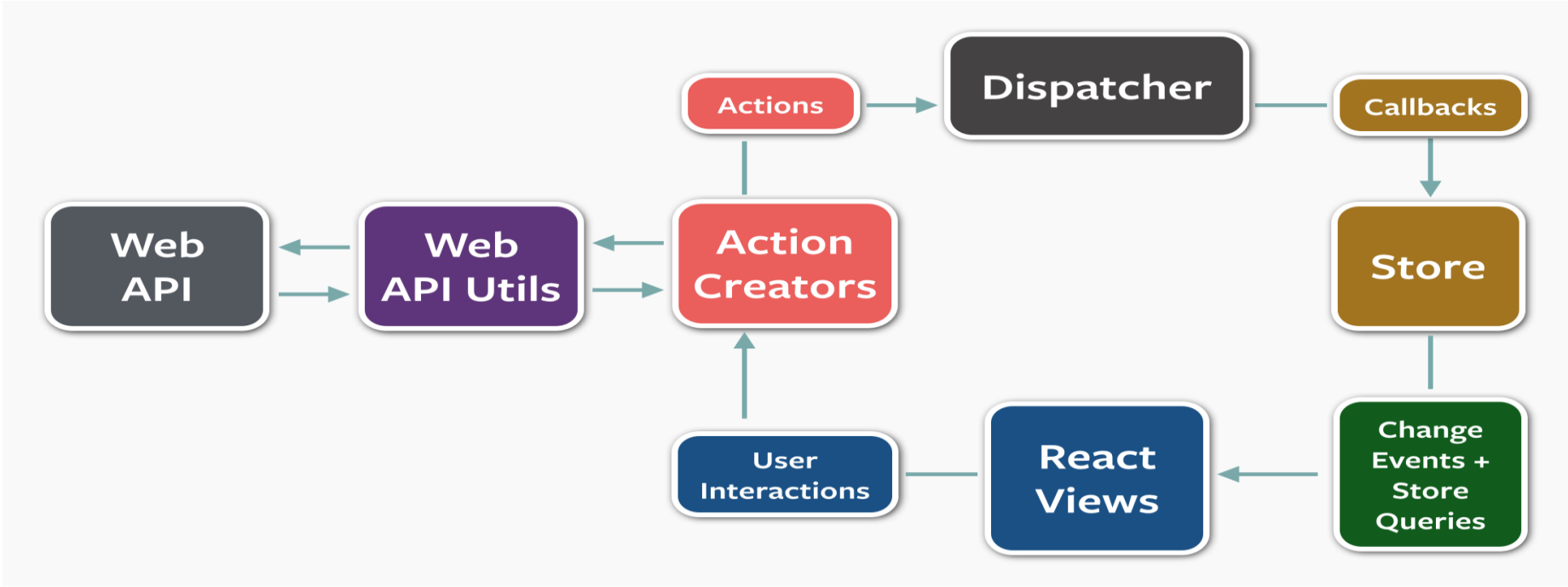
Действие (Action) рассматривается как объект, обычно имеющий идентификатор и данные о полезной нагрузке, который направляется через Dispatcher к целевому Store.



АРХИТЕКТУРА FLUX



АРХИТЕКТУРА FLUX



Блок 2.
Задание 3.
Изучаем Flux