Тестирование производительности клиентской части React/Redux приложения с использованием Enzyme

Ольга Кукса,
Software Engineer In Test,
Modelus LLC

olga.kuksa@gmail.com

Зачем тестировать производительность клиента?

- значительная часть бизнес-логики находится на клиенте;
- пользователь работает с довольно большим объёмом данных (1000 записей и более);
- на этих данных нужно проводить расчёты;
- приложение не должно подвисать вне этой “объёмной” части.

Enter text to filter by

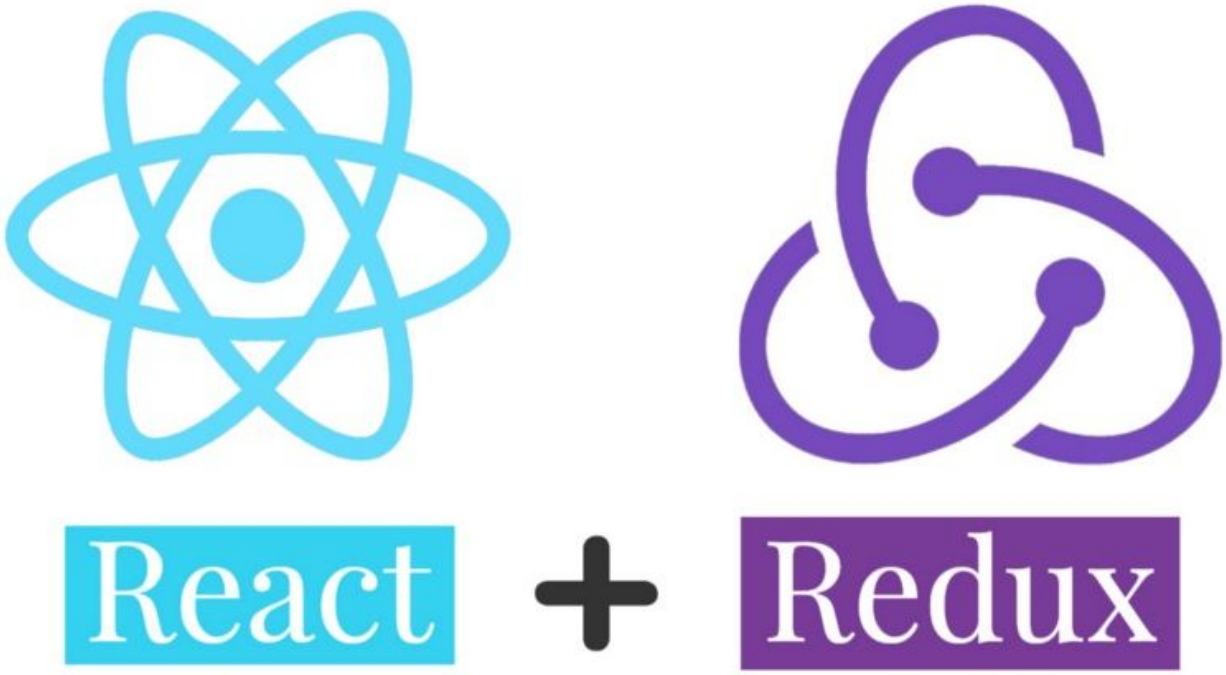
Green Red

Slow Fast Broken

1968 bug(s) found

Line	Bug Number	Refs	Title	User		Body	Created at
1	18082	0	Profiler tooltip tweaks	bvaughn	closed	M follow ups to #18048	2/19/2020
2	18080	0	Add guard around FocusWithin re nder root even tru m 4)		closed	We un into an interesting problem internally that is like ue to a strange set of events from the recent oreBlur` change, causing `addRootEventTypes` to try and add the same event before removing it. This change should guard the change to prevent the invariant from occuring in these cases. I'll try and make a repro test tomorrow and see if there's an alternative way to fix this.	2/19/2020
3	18079	0	Bug: react-hooks/exhaustive-deps misses a case with useCallback	jdolearydl (2)	open	The following causes an eslint warning (as it should) ``` const callback = useCallback(()=> cb(), []) ``` `React Hook useCallback has a missing dependency: 'cb'. Either include it or remove the dependency array. If 'cb' changes too often, find the parent component that defines it and wrap that definition in useCallback. (react-hooks/exhaustive-deps)eslint` However, the following does **not** cause the warning ``` const callback = useCallback(cb, []) ``` but it should because this line is exposed to the same stale closure bug as the first. (Maybe this is an antipattern, but in any case, the linter should be able to notice the problem, because this pattern can be used to "get around"	2/19/2020

Часть 1. React и Redux



Что такое React?

JavaScript-библиотека для создания UI, основанных на компонентах

Основные части React-компоненты

- **render()** - возвращает то, что отрисует браузер
- **props** - объект со входными данными для React-компоненты, передаются от родительского компонента дочернему (“внешние” данные)
- **state** - объект, в котором хранится состояние компоненты

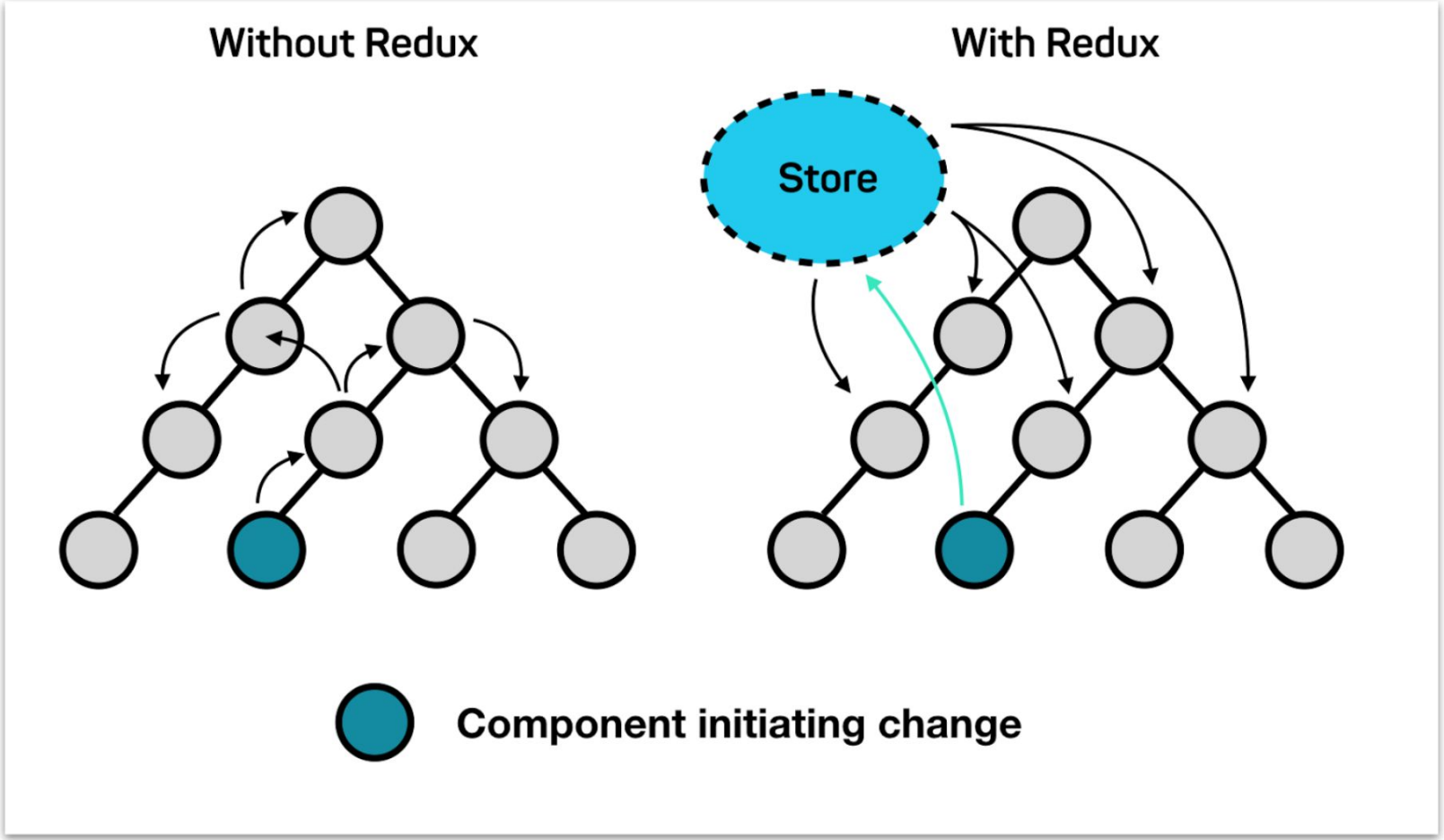
На самом деле методов, хуков и свойств React-компоненты гораздо больше - https://ru.reactjs.org/docs/react-component.html


```
class Filter extends React.Component {
  constructor(props) {
     super();
     this.state = { value: props.initialValue || "" }; //передача из props в state
   }
  onValueChange = e => {
      this.setState({ value: e.target.value });
    };
  render() {              // рендерим, используя данные из props и state компоненты
    return (
      <input
        type="text"
        value={this.state.value}
        onChange={this.onValueChange}
        placeholder={this.props.placeholder}
      ></input>
    )}};
```

```
render() {
    return (
      <div>
          <Filter placeholder="Enter text to filter by..." initialValue=""/>
          {...}
          <BugTable results={this.props.results}/>
     </div>
    );
  }
```

А что, если BugTable результаты не нужны? Зато они нужны её дочерней компоненте

```
render() {
    return (
      <div>
          <ResultCountMessage count={this.props.results.length}>
          <Table results={this.props.results}/>
     </div>
    )}
```

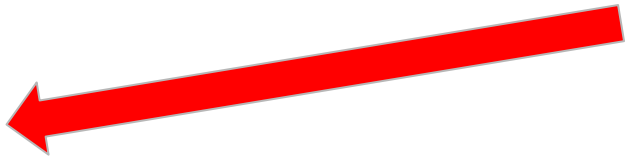

Without Redux
With Redux
Store
Component initiating change

```
class Filter extends React.Component {
  constructor(props) {
      super();
      this.state = { value: props.filterValue || "" };
    }
  {...}

const mapStateToProps = state => ({
  filterValue: state.filterValue
});

const mapDispatchToProps = { updateFilter };

export default connect(mapStateToProps, mapDispatchToProps)(Filter);
```



mapStateToProps бывают сложные...

```
const mapStateToProps = state => {
  const filterValues = state.filterValue
      .split(" ")
      .map(it =>  it.trim().toLowerCase())
      .filter(Boolean);
```

```
Application State:
{
  filterValue: "",
  results:[...],
  activeColorScheme:"green"
}
```

```
   const calculatedResults = state.results.map(it => {...});
   const filteredResults = filterValues.length
     ? calculatedResults.filter(it =>  {...} )
     : calculatedResults;
   return { filteredResults };
};
render() { return <BugTable results={this.props.filteredResults} />; }
export default connect(mapStateToProps)(Slow);
```

Reselect: использование мемоизации при расчёте mapStateToProps

```
class Fast extends React.Component {
  render() { return <BugTable results={this.props.filteredResults} />; }
}
const getFilterValue = state => state.filterValue; //считываем данные из state
const getResults = state => state.results;

const getCalculatedResults = createSelector( [getResults],  results => {...}); //обсчёт
const getFilterValues = createSelector([getFilterValue], filterValue => {...});

const getFilteredResults = createSelector(         //применяем обсчитанный ввод фильтра на
  [getCalculatedResults, getFilterValues],         //обсчитанные результаты в таблице
  (calculatedResults, filterValues) => {...};

 const mapStateToProps = createStructuredSelector({
   filteredResults: getFilteredResults
 });

export default connect(mapStateToProps)(Fast);
```

Но не все селекторы одинаково полезны...

```
class Broken extends React.Component {
  render() { return <BugTable results={this.props.filteredResults} />; }
}
const getFilterValue = state => state.filterValue;
const getResults = state => state.results;
const getAllState = state => state;

const getCalculatedResults = createSelector( [getResults, getAllState],  (results, state) =>
{...});

const getFilterValues = createSelector([getFilterValue], filterValue => {...});
const getFilteredResults = createSelector(
  [getCalculatedResults, getFilterValues],
  (calculatedResults, filterValues) => {...);

const mapStateToProps = createStructuredSelector({
  filteredResults: getFilteredResults
});
export default connect(mapStateToProps)(Broken);
```

Не все случаи “протечек” в селекторах очевидны

- функция берёт слишком много данных из state, в том числе и ненужных для компоненты. Меняются “ненужные” данные - запустился виток перерасчёта.

- “цепочка селекторов”: если есть сбой в одном из звеньев, пострадают следующие

Что же делать?

Часть 2. Unit? UI? Integration?

- Unit-тесты
- UI-тесты
- Integration-тесты

Unit-тесты

многократный вызов mapStateToProps,
точечная проверка селекторов

плюсы :

- скорость написания и простота единичного теста;
- скорость прогона единичного теста;

минусы :

- долгий поиск слабых мест/покрытие всего;
- покроется mapStateToProps, а не весь цикл с action-reducer- render;

UI-тесты

навешивание проверки производительности на UI тесты

плюсы :

- покрытие критичных путей;
- тестируется полный путь данных в react-redux (mapStateToProps-actions-reducer-render);

минусы :

- самые медленные (написание, поддержка, прогон);
- самые хрупкие;
- более сложное развёртывание окружения (по сравнению с Unit, Integration);

Integration-тесты

рендеринг компоненты как части приложения в JSDom с симуляцией действий пользователя

плюсы :

- покрытие критичных путей;
- тестируется полный путь данных в react-redux (action-render);
- приемлемая скорость;
- стабильность;
- есть готовые инструменты в unit-тестировании (Enzyme, React Testing Library и др.)

минусы :

- много мокать

Часть 3. Enzyme

Что такое Enzyme?

Enzyme - это тестовая утилита для React-компонентов. Она позволяет:

- “отрендерить” компоненту в JSDom - headless JS-браузере (дефолтная среда в jest);
- провести с ней определённые манипуляции (например, “просимулировать” действия пользователя);
- получить её “компонентную” структуру и свойства или html.

Enzyme API: shallow-рендеринг компоненты

shallow - “мелкий” рендеринг. Как правило, используется для юнит-тестов. Компонента не рендерится вглубь, при вызове метода debug() вложенные компоненты отображаются в стиле html-тэгов:

```
const shallowWrapper = shallow(
      <PageViewHeader initialSelectedComponentId={id} />
    );
console.log(shallowWrapper.debug())
```

Результат:

```
<div>
        <Filter />
<div>
```

Enzyme API: mount-рендеринг компоненты в JSDom

mount - “глубокий” рендеринг в DOM. Рендерится вглубь не только верхняя, но и вложенные компоненты. При вызове метода debug() отображается то, что возвращает метод render() вложенных компонент:

```
const mountWrapper = mount(
      <PageViewHeader initialSelectedComponentId={id} />
    );

console.log(mountWrapper.debug());
```

```
<div>
        <Filter updateFilter={[Function]}>
           <input className="filter-input" type="text"
value="" onChange={[Function]} placeholder="Enter text to
filter by" />
         </Filter>
<div>
```

shallow - быстрый, более “лёгкий”, но не тестирует влияние изменений одной компоненты на другую. Первый выбор для юнит-тестов.

mount - более медленный, более тяжёлый, позволяет отследить влияние изменений в одной компоненте на другую. Первый выбор для integration-тестов.

Enzyme API: find

find() - метод, который по переданному селектору находит в DOM-дереве соответствующие ноды

find() принимает:

- CSS-селектор

```
expect(wrapper.find(".bar").length).toBe(3)
```

- имя компоненты

```
expect(wrapper.find("Foo").length).toBe(1);
```

- объект со свойствами

```
expect(wrapper.find({ prop: 'value' }).length).toBe(1)
```

- конструктор компоненты

```
import Foo from '../components/Foo';

const wrapper = mount(<MyComponent />);

expect(wrapper.find(Foo).length).toBe(1);
```

Enzyme API: filterWhere, findWhere, prop()

filterWhere(fn) - фильтрует текущие ноды по fn
findWhere(fn) - возвращает ноды из дерева с фильтрацией по fn
prop(key) - возвращает значение свойства key

```
pageViewWrapper
    .find(SchemeInput)
    .filterWhere(it => it.prop("color") === "red")
```

есть дополнительные методы: first(), last(), at(position) и т.д., подробней - https://airbnb.io/enzyme/docs/api/mount.html

Enzyme API: simulate

simulate() - симулирует события на выбранной ноде

```
pageViewWrapper
      .find(ComponentMenuItem)
      .filterWhere(it => it.prop("id")==="fast")
      .simulate("click"); //"mousedown" etc.

pageViewWrapper
      .find(Filter)
      .simulate("change", {target:{value:"test"}});
```

Часть 4. Пишем тест

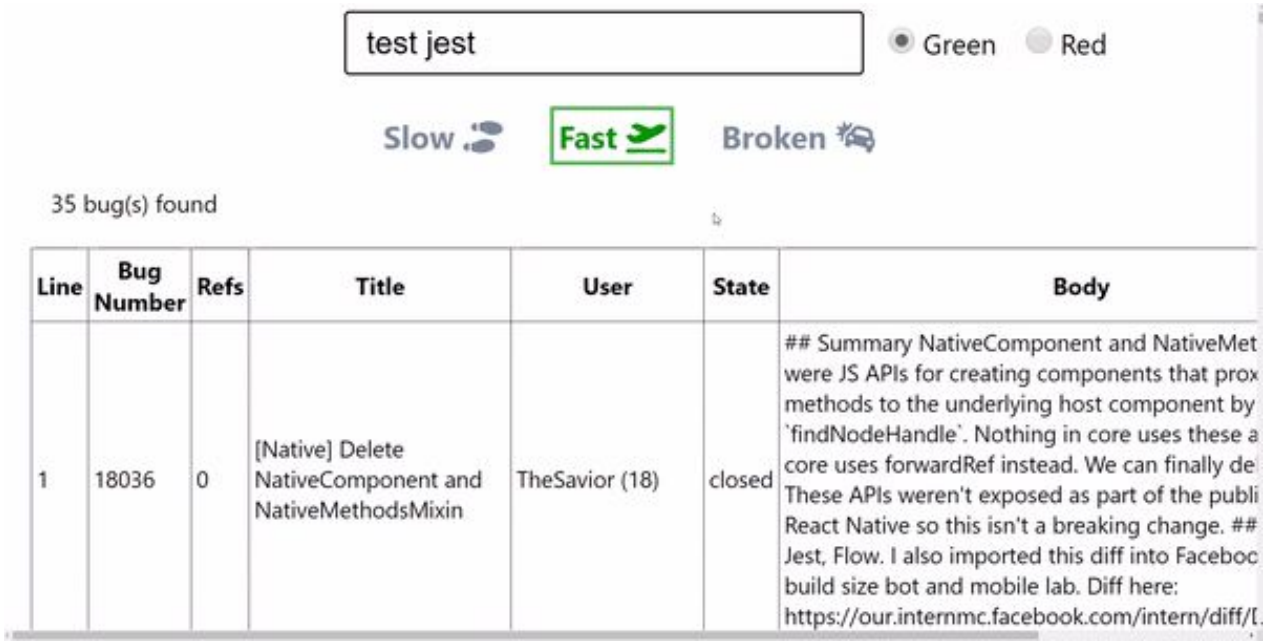
Сценарий для автоматизации:

Исходное состояние: >1900 багов, выбрана компонента (fast, slow или broken), в фильтр введено значение, цветовая схема - зелёная

Сценарий:

1. Кликнуть на радио красной цветовой схемы

Ожидаемый результат: цветовая схема изменилась на красную достаточно быстро.



Line	Bug Number	Refs	Title	User	State	Body
1	18036	0	[Native] Delete NativeComponent and NativeMethodsMixin	TheSavior (18)	closed	## Summary NativeComponent and NativeMet were JS APIs for creating components that prox methods to the underlying host component by `findNodeHandle`. Nothing in core uses these a core uses forwardRef instead. We can finally de These APIs weren't exposed as part of the publi React Native so this isn't a breaking change. ## Jest, Flow. I also imported this diff into Faceboc build size bot and mobile lab. Diff here: https://our.internmc.facebook.com/intern/diff/[

Напишем простой Integration-тест

A - arrange

- сгенерируем данные
- создадим объект state на основе данных
- сделаем Redux-store из state
- отмаунтим высокоуровневую компоненту, присоединяя store

A - act

- будем засекать время
- найдём нужную ноду с помощью find ()
- просимулируем действия пользователя

A - assert

- оценим время, которое затратилось на этапе act
- проверим, что изменились части приложения, на которые повлияли действиями пользователя

Arrange

1. **Генерация тестовых данных:**

```
const generateResults = (count = 2000) => {
  const results = [];
  for (let i=0; i < count; i++) {
    const result = {number: i,...    };
    results.push(result);
  }
  return results;
};
const testResults = generateResults();
```

2. **Создание объекта state:**

```
const state = {
      filterValue: "race",
      results: testResults,
      activeColorScheme: "green"
    };
```

Arrange

3. Создание store:

```
const testStore = createStore(reducer, state);
```

или

```
const testStore = createStore(reducer, state, [thunk]);
```

* вариант helper'a:

```
import { createStore } from "redux";
import reducer from "../reducer";

export const createStoreFromState=(state)=>createStore(reducer, state,
middlewares);
-----------------
//в тестовом файле после импорта функции createStoreFromState:
const store=createStoreFromState(state);
```

Arrange

4.Создание enzyme-компоненты PageView с подключением store

```
const pageViewWrapper = mount(
  <Provider store={testStore}>
    <PageView initialSelectedComponentId={id} />
  </Provider>
);
```

Arrange закончен!

Act

```
const startTime = performance.now();
pageViewWrapper
    .find(SchemeInput)//радио-кнопка
    .filterWhere(it => it.prop("value") === "red")//красная
    .find("input")
    .simulate("change");//onChange={this.props.onChange}
const endTime = performance.now();
```

Это очень простой сценарий!

Assert

```
//ниже будет ваша проверка
console.log(`Switching color took ${endTime - startTime)} ms for {id}`);
```

Раз уж мы всё равно здесь…

```
expect(
      pageViewWrapper
        .find(ComponentMenuItem)
        .filterWhere(it => it.prop("id") === id)
        .hasClass("red-color")
    ).toBe(true);                          //появился класс красного цвета
expect(pageViewWrapper.find(BugTable).prop('results').length).toBe(13)
                                  //результатов в таблицу багов пришло 13
```

Протестируем все 3 компоненты…

```
const testColorSwitch = id => {
    const state = {...};
    const testStore = createStore(...);
    const pageViewWrapper = mount(
          <Provider store={testStore}>
            <PageView initialSelectedComponentId={id} />
          </Provider>);

    const startTime = performance.now();
    pageViewWrapper.find(...).simulate(...);
    const endTime = performance.now();
    console.log(endTime-startTime);
    expect(...);//'red' class; 13 result in BugTable
    pageViewWrapper.unmount()  };
```

```
test("Fast", () => {
testColorSwitch("fast")})
```

Результаты тестов:

```
PASS  src/__tests__/PageView.js (5.168s)
  change color scheme tests
    √ Fast (587ms)
    √ Slow (996ms)
    √ Broken (991ms)

  console.log src/__tests__/PageView.js:67
    Switching color took 23.849 ms for FAST

  console.log src/__tests__/PageView.js:67
    Switching color took 491.68 ms for SLOW

  console.log src/__tests__/PageView.js:67
    Switching color took 490.54 ms for BROKEN

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        5.691s, estimated 6s
```

Но разработчики порой ошибаются…

```
class SchemeChanger extends React.Component {
 ...
 onChangeColorScheme = e => {
    this.setState({ activeScheme: e.target.value },
      () =>this.props.updateColorScheme(this.state.activeScheme 'blue')
    );
  };
  render() {
    return (
        <span>
          <SchemeInput checked={this.state.activeScheme === "red"} value="red"
displayName="Red" onChange={this.onChangeColorScheme}/>
        <SchemeInput value="green"... />
      </span> )}};
```

```
$ node scripts/test.js
 FAIL  src/__tests__/PageView.js (5.219s)
  change color scheme tests
    × Fast (591ms)
    × Slow (999ms)
    × Broken (987ms)

  ● change color scheme tests › Fast

    expect(received).toBe(expected) // Object.is equality

    Expected: true
    Received: false

      62 |           .filterWhere(it => it.prop("id") === id)
      63 |           .hasClass("red-color")
    > 64 |       ).toBe(true);
         |         ^
      65 |       expect(pageV                              lts').length).toBe
      66 |
      67 |       console.log(

      at testColorSwitch (src/__tests__/PageView.js:64:7)
      at Object.<anonymous> (src/__tests__/PageView.js:75:5)
```

всё 3 теста упали

на этапе проверки наличия класса red-color

Полезные методы для проверок из Enzyme

- **text()** - возвращает текст, который рендерится внутри html дерева
- **html()** - возвращает HTML-разметку enzyme-компоненты в виде строки
- **exists(selector)** - существует ли элементы, соответствующие селектору
- **is(selector)** - подходит ли обёрнутая нода под селектор
- **everyWhere(fn)** - подходят ли ноды под fn
- **state(key)** - вернёт значение key из стэйта компоненты
- **prop(key)** - вернёт key из props компоненты

... и многое другое - https://airbnb.io/enzyme/docs/api/mount.html

\+ toJson() из enzyme-to-json для снэпшотов

Полезные методы для проверок из Redux и jest

- **store.getState()** - вернёт state приложения

```
const state=store.getState();
expect(state.filterValue).toBe('new text')
```

- **jest mock functions** - удобно мокать запросы на сервер:

```
const serverMockFn=jest.fn();
…
expect(serverMockFn.mock.calls).toEqual([["req1", "arg1"]])
```

Для Integration-тестов можно использовать подходы UI-тестирования

PageObject паттерн:

```
const pageView=new PageView(state, id);

const duration = getDuration(()=>pageView.clickSchemeChanger("red"));

expect(pageView.hasClassMenuItem("red-color", id)).toBe(true);
expect(pageView.getBugCount()).toBe(13);
expect(duration).toBeLessThan(100)};

pageView.end();
```

У нас есть Integration-тесты клиента, которые:

- могут реализовать UI сценарии (в т.ч. с состоянием приложения, которого не очень просто достичь в UI-тестах)
- имеют простую подготовку среды (по данным - state и мокер ответа сервера)
- имеют хорошие стабильные селекторы (конструкторы компоненты)
- дают возможность быстрой разработки и поддержки
- работают быстрее UI тестов
- поддерживают паттерны, которые помогут создать удобную структуру тестов

Можно ли полностью заменить ими UI-тесты?

Ограничения Integration-тестов клиента vs UI:

- не тестируют взаимодействие клиента и сервера системно (только клиент)
- нет кросс-браузерного тестирования
- нет поддержки проверки видимости элемента из коробки (*но есть @testing-library/jest-dom и его DOM-мэтчеры:* `expect(wrapper.getDOMNode()).toBeVisible());`)
- React разрабатывается быстрее Enzyme: фичи могут появиться в коде, но возможность их покрыть будет не сразу (Suspense, функциональные компоненты)

Так где могут помочь Integration-тесты клиента?

В первую очередь - в тестировании логики, зашитой в JavaScript клиента, на уровне межкомпонентного взаимодействия.

- ❖ **Unit-тесты** - на уровне 1 компоненты, функции и т.п.

- ❑ **API тесты** - запросы на сервер.
- ❑ **Integration-тесты** клиента - на уровне действие в одной компоненте -> последствия в других местах (другой компоненте, state, запросах на сервер).

- ★ **UI** - покрытие самых важных сценариев.

частично

Часть 5. Проблемы и решения

Проблема: асинхронные вызовы и обработчики.

Описание: этап Act вызывает асинхронные действия, которые не успевают завершиться на момент Assert - проверки (вызовы на сервер, Promise)

Решение: добавить асинхронность в тесты

- **setTimeout()+done**

```
it (“my test”, (done)=>
...Act;//этап Act
setTimeout(()=>{
...Assert; //этап Assert
done();
})
```

- **асинхронный хелпер**

```
export function delay(ms) {
 return new Promise(resolve => {
  setTimeout(() => {resolve();
   return;}, ms)})}
```

в тесте: it(“my test”, **async** ()=>{
...Act //этап Act
await delay(0);
...Assert //этап Assert})

Проблема: enzyme-компонента не подхватывает изменения

Описание: после simulate компонента должна измениться, но изменения в ней не произошли. Видимо, Enzyme считает, что не нужно вызывать цикл обновления компоненты.

Решение: вызвать wrapper.update() вручную. Возможно, вместе с хелпером асинхронности, если обновление происходит после асинхронных действий

в тесте:

```
it(“my test”, async ()=>{
...Act //этап Act
await delay(0);
wrapper.update();
...Act //продолжение Act
...Assert //этап Assert})
```

Проблема: нужно получить ответ с сервера

Описание: после определённых действий в UI отправляется запрос на сервер. Результаты запроса клиент использует дальше.

Решение: моки. Если есть API, мокаем API (возможно, даже через обёртку над createStore), иначе мокаем модули fetch, axios и т.п.

в тесте:

```
const mockResponseFn = jest.fn(() => dataFromServer);
jest.mock("axios", () => ({
  get: mockResponseFn
}));
beforeEach(jest.clearAllMocks);//если проверяем
                mockResponseFn.mock.calls, зачищаем их между тестами
```

Проблема: в проекте используются styled-components

Описание: при генерации html библиотека styled-components каждый раз рассчитывает новое имя класса. Если снэпшотить html, снэпшоты валятся.

Решение: использовать библиотеку jest-styled-components. С ней имена классов будут постоянными

в тесте:
import “jest-styled-components”

Проблема: использование разных тем

Описание: styled-компоненты ожидают, что у них в props обязательно будет поле theme. В приложении это решается через ThemeProvider. Если в тестах его нет, тесты не запускаются.

Решение: можно обернуть mount в ThemeProvider с дефолтной темой и прописать запуск этого мокера в setupFiles в jest.config.

мокер:

```
jest.mock("enzyme", () => {
 const mount = require.requireActual("enzyme").mount;
 const ThemeProviderWrapper = ({children}) => (
  <ThemeProvider theme={defaultTheme}>{children}</ThemeProvider>
 );
 return {
  ...require.requireActual("enzyme"),
  mount: tree => mount(tree, {wrappingComponent: ThemeProviderWrapper}
 }});
```

Проблема: использование Router из react-router

Описание: компонента экспортируется через withRouter() и требует рутер для рендеринга

Решение: использовать MemoryRouter

```
const pageViewWrapper = mount(
    <Provider store={this.store}>
     <MemoryRouter initialEntries={[initialPath]}>
            <PageView / >
     </MemoryRouter>
    </Provider>);
```

* если нужно проверять переходы и location:

```
<Route name="pageView" path="/PageView" component={PageView} />
<Route path="*" render={({ history, location }) => {this.history = history;
        this.location = location; return null; }} />
```

Проблема: стык Enzyme, Jest и сторонних модулей, которые используются в проекте

Описание: если на свой код мы можем как-то повлиять, то решить проблему багов в сторонних модулях не так просто (баги на стыке Jest/Enzyme и lodash/DraftJS и т.д.)
Решение: искать и использовать workarounds и надеяться на фиксы.
Пример:

jest.runAllTimers() подвисал если использовался lodash.debounce

issue: https://github.com/facebook/jest/issues/3465

спустя почти 3 года пофиксили, в jest 27 бага не будет :-)

вариант решения:

```
jest.mock("lodash/debounce", () => fn => {
  const debounce = jest.requireActual("lodash/debounce");
  return debounce(fn, 0);
});
```

Проблема: изменения составных частей начального state приложения (мутация данных)

Описание: в одиночку тест проходит, при запуске в сьюте падает. Обнаруживается, что один из тестов изменяет исходные данные, из которых конструируется начальный state.

Решение: заводить багу. Или идти к разработчикам и сообщать, что они мутируют Redux-state напрямую.

```
import deepFreeze from 'deep-freeze';

deepFreeze(results);
```

…

TypeError: Cannot assign to read only property 'results' of object '#<Object>'

at src/components/Fast/index.js:107:7

Проблема: недостаточная квалификация тестировщиков

Описание: для написания Integration-тестов нужно разбираться во внутренней структуре приложения, владеть несколькими инструментами (как минимум Enzyme и jest или похожими)

Решение: учиться и не бояться спрашивать девелоперов. Развиваться и действовать!

+ вы растёте в профессиональном плане (тех. скиллы, востребованность на рынке труда)
+ ваш проект получает автотесты, которые более эффективно решают определённый пул задач.

Итого: Integration-тесты клиента

- отлично справляется с поиском протечек в производительности клиента;
- способны словить критичные баги в JS клиента, если поставить соответствующие проверки, - и таким образом могут встроиться в пирамиду тестирования React/Redux приложения;
- при разработке можно столкнуться с проблемами, но они решаемы.

И в заключение…

Спасибо за внимание!

email: olga.kuksa@gmail.com

demo-репозиторий с тестами из презентации:
https://github.com/OlgaKuksa/client-perf-demo