

ReactJS. Базовый курс

Тестирование и оптимизация приложений на React

[React 17.0.1]



На этом уроке

1. Узнаем о Jest и тестировании приложений на React
2. Научимся использовать Jest и react-testing-library для создания простых тестов.
3. Познакомимся с оптимизацией веб-приложений, узнаем о shouldComponentUpdate и PureComponent.
4. Познакомимся с Lighthouse и узнаем о его метриках.

Оглавление

[На этом уроке](#)

[Теория урока](#)

[Тестирование приложений на Реакт](#)

[Почему важно тестировать свой код](#)

[Unit-тестирование. Jest](#)

[Конфигурация Jest](#)

[Работа с jest](#)

[Тестирование снейшотами. React-testing-library.](#)

[Работа jest с react-testing-library](#)

[Оптимизация производительности приложений на Реакт](#)

[Чистые компоненты. React.memo.](#)

[Профайлинг](#)

[Lighthouse](#)

[Глоссарий](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Теория урока

Тестирование приложений на Реакт

Почему важно тестировать свой код

Тестирование — один из важнейших аспектов разработки. Оно позволяет убедиться в качестве разрабатываемого продукта и значительно уменьшить количество багов, с которыми столкнется пользователь. Ручное тестирование — ответственность команды QA, его мы здесь не рассматриваем. Но существуют тесты, за написание которых ответственен сам разработчик — автор кода.

Каждый раз, когда мы добавляем новый инструментарий или вносим изменения в существующий, нам необходимо проверить — а не сломалось ли то, что мы сделали раньше? Даже в небольших проектах зачастую бывает сложно удержать в голове все взаимосвязи отдельных частей программы.

К примеру, у нас есть функция `FormatString`. Она должна вернуть некоторую строку.

```
export function formatTimeStrings(strings) {
  if (strings.length > 1) {
    return `${strings[0]} - ${strings[strings.length - 1]}`
  }

  if (strings.length) {
    return `${strings[0]} - Till tomorrow`
  }

  return `None`;
}
```

Эта функция используется нами в нескольких компонентах. В какой-то момент заказчик решил, что в одном из компонентов строка должна быть форматирована по-другому, например, так:

```
export function formatTimeStrings(strings) {
  if (strings.length > 1) {
    return `${strings[0]} - ${strings[strings.length - 1]}`
  }

  if (strings.length) {
    return strings[0];
  }

  return '';
}
```

Разработчик изменил функцию согласно новым требованиям, но забыл, что она используется и в других компонентах. В итоге у нас получилось неверное отображение строки. Чтобы такого не случалось, полезно иметь тест. Он будет проверять работу нашей функции и не даст нам совершить ошибку. Мы запустим тесты, увидим, какие упали, и поймём, что надо исправлять — тесты или реализацию.

Конечно, скорее всего, эта ошибка была бы очевидна на стадии QA. Но нужно помнить, что ручное тестирование — длительный, дорогостоящий процесс. Большинство мелких ошибок намного проще отловить ещё до этой стадии.

Unit-тестирование. Jest

Различают разные виды автоматического тестирования - E2E, сервисные, Unit-тесты и др. В данном уроке мы будем рассматривать unit-тестирование, как наиболее распространенное и наиболее часто требующееся на практике. (Подробнее о разных видах тестирования см, например [Пирамида тестирования](#)).

Формальное определение unit (или модульного) тестирования следующее: тестирование, в ходе которого проверяется работа каждой нетривиальной функции; То есть, с его помощью мы проверяем работу каждой части нашей программы по отдельности. Каждый тест для каждого компонента или функции (в идеале) должен быть таким, чтобы его можно было запустить отдельно и независимо от других тестов и компонентов.

К примеру, тест, проверяющий работу функции `formatString` из предыдущего раздела будет являться модульным - мы тестируем работу одной функции вне зависимости от того, где и как она вызывается.

Как правило, для запуска unit-тестов в js используется библиотека `jest` - она стала настолько стандартной, что уже предустановлена в вашем проекте (через CRA), и на ней написан как минимум один тест:

src/App.test.js

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

У него наиболее простая интеграция с другими инструментами для тестирования приложений на Реакт.

Тест для функции `formatTimeStrings`, написанный с использованием `jest`, будет выглядеть следующим образом:

```
describe('formatTimeStrings tests', () => {
  it('returns None if no opening hours passed', () => {
    const expected = 'None';
    const received = formatTimeStrings([]);

    expect(received).toEqual(expected);
  });

  it("returns 'start - Till tomorrow' if only one opening hours passed", () => {
    const openingHours = ["12-00"];
    const expected = `${openingHours[0]} - Till tomorrow`;
    const received = formatTimeStrings(openingHours);

    expect(received).toEqual(expected);
  });

  it("returns 'start - end' if more than one opening hours passed", () => {
    const openingHours = ["12-00", "16-00", "18-00"];
    const expected = `${openingHours[0]} - ${openingHours[2]}`;
    const received = formatTimeStrings(openingHours);

    expect(received).toEqual(expected);
  });
});
```

Заметьте, что, даже не вдаваясь в подробности того, что происходит во время тестов, из них нам становится понятно, что функция предназначена для форматирования часов открытия. Модульные тесты не только защищают нас от ошибок, но и могут являться описанием для нашего кода.

Тесты с использованием `Jest` в принципе написаны так, как строится предложение на простом английском. Описываем: тесты `formatTimeStrings`. Она: возвращает `None` если ей не передали часы открытия и т.д.

Помните - писать описания следует так, чтобы и вам, и вашим коллегам было очевидно, какая именно функциональность проверяется в данном тесте и, если тест падает - можно было легко отыскать источник ошибки.

Конфигурация Jest

Для более удобной работы с jest мы можем кастомизировать его настройки. Сделать это можно либо добавив секцию jest в package.json, либо создав в корне проекта файл jest.config.js.

package.json

```
{ } package.json ×
{ } package.json > ...
43   },
44   "devDependencies": {
45     "@testing-library/react-hooks": "^3.4.2",
46     "jest-fetch-mock": "^3.0.3",
47     "react-test-renderer": "^17.0.1"
48   },
49   "jest": {
50     "collectCoverage": true,
51     "errorOnDeprecated": true,
52     "setupFiles": [".jestsetup.js"],
53     "coverageThreshold": {
54       "global": {
55         "branches": 80,
56         "functions": 80,
57         "lines": 80,
58         "statements": -10
59       }
60     }
61   }
62 }
```

jest.config.js

```
JS jest.config.js ×
JS jest.config.js > ...
1  module.exports = {
2    verbose: true,
3    errorOnDeprecated: true,
4    setupFiles: [".jestsetup.js"],
5    coverageThreshold: {
6      global: {
7        branches: 80,
8        functions: 80,
9        lines: 80,
10       statements: -10,
11     },
12   },
13 };
14
```

Здесь мы можем указать различные параметры для запуска наших тестов - переопределить глобальные переменные, указать, как и откуда собирать метрики покрытия, указать паттерны, по которым jest будет искать файлы с нашими тестами, и многое другое.

Обратим внимание на два параметра - `setupFiles` и `setupFilesAfterEnv` - здесь указывается путь к скриптам, которые будут запущены перед запуском наших тестов. Это полезно, если вы хотите произвести дополнительную настройку - например, подключить некоторые дополнительные модули. Полный список параметров находится здесь - <https://jestjs.io/docs/en/configuration>.

Помимо конфигурационных файлов, можно использовать параметры командной строки. Например, запустив `jest path/to/test.tsx`, мы выполним тестовый файл по переданному пути. Параметр `-o` позволяет запустить только тесты, файлы которых изменились с момента последнего коммита.

Полный список доступных аргументов присутствует в официальной документации (<https://jestjs.io/docs/en/cli>)

Внимание!

При запуске тестов через "yarn test" параметры командной строки используются так же, как и при запуске jest. При использовании npm run test между командой и параметрами (кроме пути к файлу с тестами) необходимо добавить "--" (двойное тире).

При создании приложения через create-react-app не все параметры можно изменять.

Работа с jest

По умолчанию jest будет запускать все, что находится в файлах `.test.js` и `spec.js` (или `.ts`). Как правило, такие файлы помещают в папки `__tests__` непосредственно рядом с тестируемым модулем.

Тесты, как правило, выглядят следующим образом - мы вызываем функцию `it` (или `test`), первым параметром передается строка, которая описывает некоторую деталь поведения нашей функции. Вторым параметром передается коллбэк, в котором и находится сам наш тест. Здесь мы, как правило, вызываем тестируемую функцию, и делаем предположение (expect) - мы ожидаем, что полученный результат будет равен правильному, определенному нами результату. Функция `expect` возвращает объект со множеством методов, называемых `matchers`, для оценки результата - `toEqual`, `toBeDefined`, `toBeNull` и др. Полный список представлен в документации - <https://jestjs.io/docs/en/expect>.

`src/store/articles/__tests__/reducer.js`

```
it("returns state with status loading after requestArticles action",
  () => {
    const expected = {
      articles: [],
      request: {
        status: REQUEST_STATUS.LOADING,
```

```

        error: null,
      },
    ];

    const received = reducer(undefined, getArticlesRequest());
    expect(received).toEqual(expected);
  });

```

Вызовы `it` можно группировать с помощью функции `describe` - в первую очередь для удобства чтения кода и результатов. Жестких правил, определяющих группировки тестов, нет - они, как правило, определяются командой разработки. Один из вариантов приведен ниже

```

describe('article tests', () => {
  describe('snapshot tests', () => {
    it('matches snapshot with article', () => {
      const component = render(<Article article={article} />);
      expect(component).toMatchSnapshot();
    });

    // more tests...
  });

  describe('functionality tests', () => {
    it('renders placeholder when no article present', () => {
      const component = render(<Article />);
      const placeholder = component.getByText('SUMMARY').parentElement;
      expect(placeholder).toHaveClass('post-empty');
    });

    // more tests...
  });
});

```

Здесь тесты, проверяющие правильность отображения UI и тесты, проверяющие функционал компонента, выделены в отдельные блоки `describe`.

Тесты, находящиеся в одном файле, называются `test suite`. При запуске тестов (по умолчанию `npm run test`) в командной строке мы увидим сперва информацию о том, какие тесты выполняются в текущий момент, а по завершении работы - отчет о прошедших и упавших тестах.


```

> 2 snapshots written.
> 2 snapshots obsolete.
  • article tests matches snapshot with article 1
  • article tests matches snapshot with no article 1

Snapshot Summary
> 2 snapshots written from 1 test suite.
> 2 snapshots obsolete from 1 test suite. To remove them all, press `u`.
  ↳ src/components/__tests__/article.test.js
    • article tests matches snapshot with article 1
    • article tests matches snapshot with no article 1

Test Suites: 1 failed, 4 passed, 5 total
Tests:       1 failed, 9 passed, 10 total
Snapshots:   2 obsolete, 2 written, 2 total
Time:        7.136 s
Ran all test suites related to changed files.

```

```

FAIL src/utils/__tests__/formatString.test.js
  ● formatTimeStrings tests > returns 'start - end' if more than one opening hours passed

    expect(received).toEqual(expected) // deep equality

    Expected: "12-00 - 15-00"
    Received: "12-00 - 18-00"

      22 |         const received = formatTimeStrings(openingHours);
      23 |
    > 24 |         expect(received).toEqual(expected);
         |                             ^
      25 |       });
      26 |     });
      27 |

      at Object.<anonymous> (src/utils/__tests__/formatString.test.js:24:22)

```

Из этого отчета мы сразу можем понять, какой тест в каком сьюте упал и почему (какой результат ожидали и какой получили).

Тестирование снэпшотами. React-testing-library.

Снэпшот - снимок состояния объекта. Как правило, с их помощью тестируют компоненты Реакт, однако они могут применяться для тестирования любых объектов. Jest предоставляет матчер `toMatchSnapshot`, при первом его вызове снимок объекта будет сохранен в папке `snapshots` в файле с расширением `snap`. При всех последующих вызовах новый снимок будет сравнен с существующим, и, если будут найдены различия - тест упадет.

С помощью снэпшотов особенно удобно тестировать функции, возвращающие большие объекты - например, редьюсеры `redux`.

```

it('returns correct state after SET_NAME action', () => {
  const userStore = userReducer(initialState, setName('new Name'));
  expect(userStore).toMatchSnapshot();
});

```

Обратите внимание - мы не создаем вручную объект с ожидаемым результатом. Если стор большой, то этот процесс может быть достаточно трудоемким. Но мы сохранили состояние нашего объекта, полученное после вызова редьюсера, и тест упадет, если в работе редьюсера произойдут изменения.

Конечно, иногда изменения были внесены намеренно - в таком случае для прохождения теста нужно обновить снэпшот. Это делается с помощью флага `-u` при запуске тестов в терминале.

Работа jest с react-testing-library

Функции, не являющиеся компонентами Реакт, как правило, составляют не такую уж большую часть наших проектов. Jest не работает ни с браузерами, ни с DOM - он запускает тесты в своем собственном окружении. Чтобы иметь возможность проверить, как работают наши компоненты, нам понадобятся дополнительные библиотеки, которые смогут “рендерить” наши компоненты, вызывать их методы жизненного цикла, работать с хуками, и т.д. Одной из наиболее часто используемых для этих целей библиотек является `react-testing-library`. В семейство `testing-library` входит несколько модулей, в их числе модули для эмуляции браузерного DOM, для работы с `react-native`, для работы с хуками.

Внимание!

В саму библиотеку Реакт уже входит тестовый рендерер - о нем можно почитать здесь:

<https://ru.reactjs.org/docs/test-renderer.html#testrenderer> - однако по сравнению с другими библиотеками он обладает достаточно ограниченным функционалом и редко используется на крупных проектах. Другой распространенной библиотекой является `enzyme` - ее API и принципы очень схожи с `react-testing-library`, здесь мы ее не рассматриваем. Ее изучение предлагается на самостоятельную работу.

Основной метод для рендера компонентов - `render`. При передаче ему компонента он эмулирует создание компонента (т.е. будут вызваны все методы жизненного цикла) и создаст объектное представление результата. Такие представления можно сравнивать друг с другом с помощью уже упомянутого выше метода `toMatchSnapshot`.

Внимание!

По умолчанию `matchers` сравнивают снэпшоты и объекты, преобразовав их к строкам - как простое строковое равенство. Ключи объектов перед этим упорядочиваются по алфавиту.

`src/components/article/__tests__/article.test.js`

```
it('matches snapshot with no article', () => {
  const component = render(
    <Gists />
  );
```

```
expect(component).toMatchSnapshot();
});
```

src/components/article/__tests__/snapshots/article.test.js.snap

```
exports[`article tests matches snapshot with article 1`] = `
Object {
  "asFragment": [Function],
  "baseElement": <body>
    <div>
      <a
        class="post"
        href="/"
      >
        <div
          class="header"
        >
          Title
        </div>
        <div
          class="imageWrapper"
        >
          
        </div>
        <div
          class="summary"
        >
          long text
        </div>
        <div
          class="bottom"
        >
          <div
            class="source"
          >
            some news
          </div>
          <div
            class="date"
          >
            Mon Nov 23 2020
          </div>
        </div>
      </a>
    </div>
  </body>,
  // ... additional components
}
```

```
;
```

Тестирование снимками может быть весьма полезным инструментом, однако не стоит употреблять его избыточно. Как правило, достаточно одного (максимум двух) снимков для компонента. Если внешний вид компонента в значительной степени контролируется пропсами (например, кнопка, которой передается проп `disabled`), намного полезнее проверить состояние или атрибуты полученного в DOM элемента (в упомянутом примере - проверить, что у кнопки есть атрибут `disabled`, или что в этом случае при нажатии не будет вызван `onPress`). В противном случае вы рискуете получить правильный снимок при неправильном поведении компонента. Подробнее о том, почему следует с осторожностью относиться к тестированию снимками, можно прочитать в [этой статье](#).

Заметьте, что мы можем передать компоненту только те пропсы, которые необходимы для текущего теста. Остальные заменяются либо пустыми объектами и функциями, либо опускаются совсем.

Кроме тестирования “внешнего вида” компонентов с помощью снимков мы можем взаимодействовать с ними - нажимать на кнопки, вводить информацию в текстовые поля, и т.п. Основной инструмент для тестирования таких взаимодействий - функция `fireEvent` из `react-testing-library`. Он предоставляет несколько методов, соответствующих событиям DOM (`click`, `scroll` и т.д.). Также можно вызвать его как функцию - в этом случае ему нужно передать элемент, на котором должно сработать событие, и сам объект события.

```
fireEvent(component, 'click');
```

Чтобы найти элемент, на котором мы хотим запустить событие - используем `queries`. Все они начинаются с `getBy`, `queryBy` или `findBy` и ищут в DOM элемент по различным свойствам (`getByText`, `getByRole`, `getByTestId` - полный список есть в [документации](#)). Их основные отличия представлены в таблице ниже:

	getBy*	queryBy*	findBy*
Асинхронный	нет	нет	да
Поведение, если элемент не найден	Выбрасывает исключение	Вернет null	Выбрасывает исключение

```
const component = render(<Gists />);
const placeholder = component.getByText("Loading...");
const placeholderNullable = component.queryByText("Loading...");
```

```
const placeholderAsync = await findByText("Loading...");
```

Если тестируемый компонент имеет доступ к стору redux - через connect или хуки - или к любому другому контексту - в тесте при рендере нам необходимо обернуть его в соответствующий контекст. Для redux используем библиотеку redux-mock-store. С ее помощью создаются фейковый редьюсер и провайдер. В тесте необходимо обернуть компонент в провайдер и передать ему фейковый стор (если компонентов, привязанных к стору, достаточно много, то удобно создать фабрику, которая будет возвращать обертку для ваших тестов).

Если компонент диспатчит actions в стор, мы можем это проверить - для этого используем функцию mockStore.getActions, которая возвращает массив всех экшенов, попавших в стор.

```
it('dispatches getArticles on mount', () => {
  render(
    <Provider store={mockStore}>
      <Gists />
    </Provider>
  );

  const actions = mockStore.getActions();
  const lastAction = actions[actions.length - 1];
  expect(lastAction).toEqual(getGistsRequest());
});
```

Внимание!

Не используйте настоящий стор из redux в тестах. Создавайте mockStore, это позволит вам контролировать состояние вашего в стор в тесте, а также даст возможность проверять попавшие туда actions.

Оптимизация производительности приложений на Реакт

Чистые компоненты. React.memo.

Основной проблемой производительности в приложениях на Реакт, как правило, является избыточное количество рендеров и обновлений DOM. Любое обновление пропсов или стейта будет вызывать перерисовку компонента, но это может быть не всегда необходимо. Рассмотрим следующий пример:

```
const Child1 = ({ count }) => {
  console.log("render child1");
```

```

    return <div>{count}</div>;
  };

const Child2 = () => {
  console.log("render child2");

  return <div>Child2</div>;
};

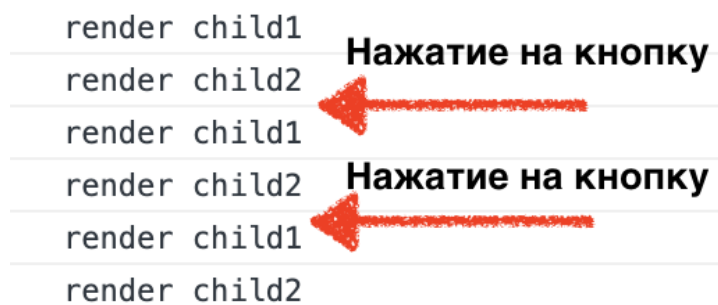
const Counter = () => {
  const [count, setCount] = useState(0);

  const updateCount = () => setCount((prevCount) => prevCount + 1);

  return (
    <>
      <button onClick={updateCount}>+1</button>
      <Child1 count={count} />
      <Child2 />
    </>
  );
};

```

При нажатии на кнопку обновляется состояние компонента Counter, что вызывает перерисовку как Child1 (получающего новое значение пропса), так и Child2:



При этом Child2, очевидно, не зависит от count и его перерисовка после обновления этой переменной не требуется. В данном (несколько утрированном) примере это не приводит к проблемам с производительностью, но если таких избыточных обновлений будет много, причем каждое из них будет связано с некоторыми вычислениями (напр., коллбэки useEffect), то это может вызвать замедление работы приложения.

Для того, чтобы функциональный компонент обновлялся только когда изменяются его пропсы, мы можем использовать НОС мемо (не путать с хуком useMemo):

```

const Child2Pure = React.memo(Child2);

const Counter = () => {

```

```

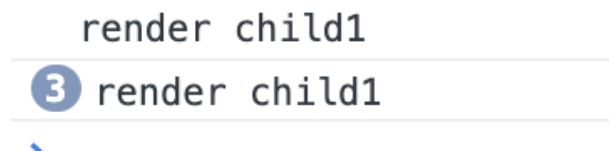
const [count, setCount] = useState(0);

const updateCount = () => setCount((prevCount) => prevCount + 1);

return (
  <>
    <button onClick={updateCount}>+1</button>
    <Child1 count={count} />
    <Child2Pure />
  </>
);
};

```

После использования этой обертки Реакт перед рендером компонента будет проводить поверхностное сравнение старых и новых значений пропсов, и обновлять компонент, только если пропсы изменились:



Внимание!

React.метод принимает и второй аргумент - функцию, которая будет производить сравнение старых и новых значений пропсов. Эта функция должна вернуть true, если компонент нужно обновить, и false - если не нужно.

Для классовых компонентов аналогичный функционал достигается за счет наследования класса от `React.PureComponent` вместо `React.Component`:

```

class Child2PureClass extends React.PureComponent {
  // ...
}

```

Данный класс реализует метод жизненного цикла `shouldComponentUpdate` - он вызывается перед рендером и работает аналогично второму аргументу `React.метод` - проводит поверхностное сравнение предыдущих и текущих значений пропсов и стейта компонента. В любом компоненте `shouldComponentUpdate` можно реализовать самостоятельно - он должен вернуть `true`, если компонент нужно обновить, и `false` - если не нужно.

Компоненты, использующие такие сравнения пропсов и стейта, называют чистыми.

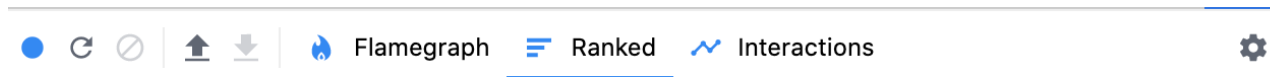
Внимание!

Во многих случаях избежать лишних рендеров можно избежать без мемоизации компонентов - несколько примеров приведены в [данной статье Дэна Абрамова](#).


Профайлинг

Профайлинг - измерение производительности приложения.

Прежде чем начать использовать чистые компоненты, необходимо верно определить те, для которых это актуально. Для этого нужно выяснить, какие из компонентов рендерятся слишком часто или слишком долго. Это удобно делать с помощью расширения для devtools React Profiler:



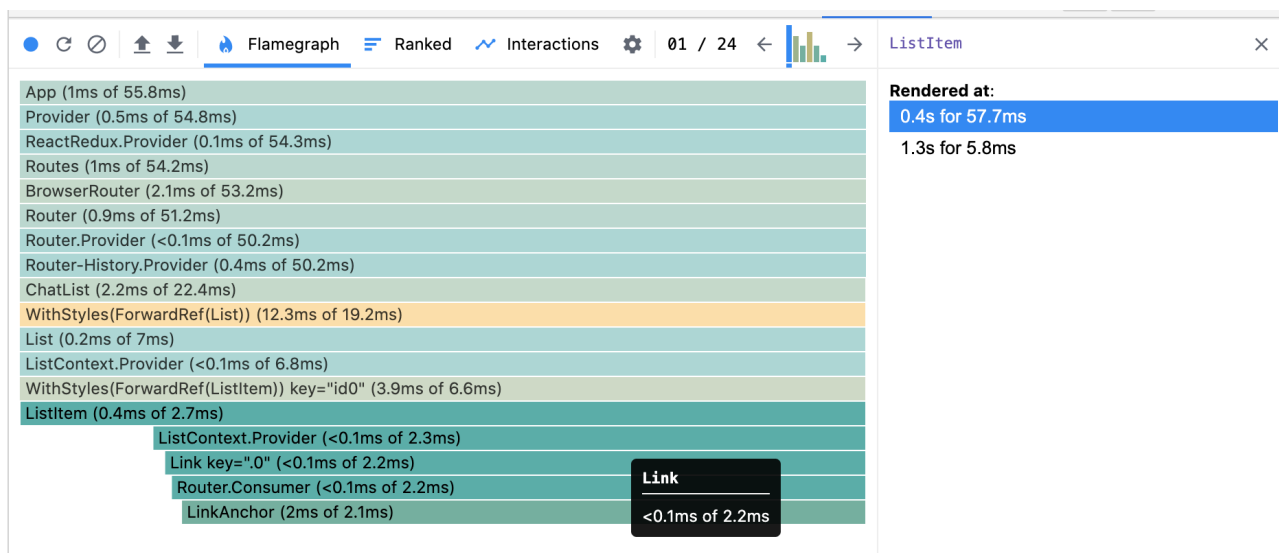
No profiling data has been recorded.

Click the record button  to start recording.

После его запуска необходимо “записать” работу приложения - React Profiler проанализирует его работу и выдаст результат:

1. На вкладке FlameChart - в виде графика, на котором можно увидеть длительность и количество рендеров каждого компонента.
2. На вкладке Ranked - аналогичный график, но упорядоченный по длительности рендера
3. Вкладка Interactions служит для отслеживания взаимодействия пользователей со страницей. Interactions находится в экспериментальном режиме, для работы с ним можно воспользоваться [данной статьей](#).

Кроме этого, в правой части доступна информация о выбранном компоненте - его состояние, пропсы, длительность каждого рендера и т.д.



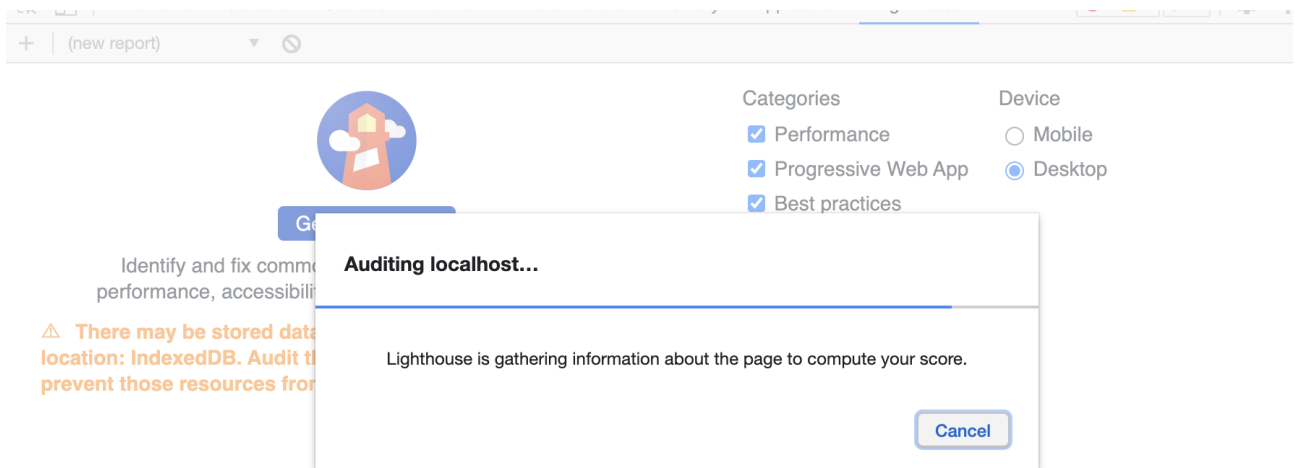
Кроме того, для оценки производительности и скорости работы приложения следует пользоваться вкладкой Performance в Chrome Devtools.

В целом, при оптимизации приложения следует руководствоваться следующими принципами:

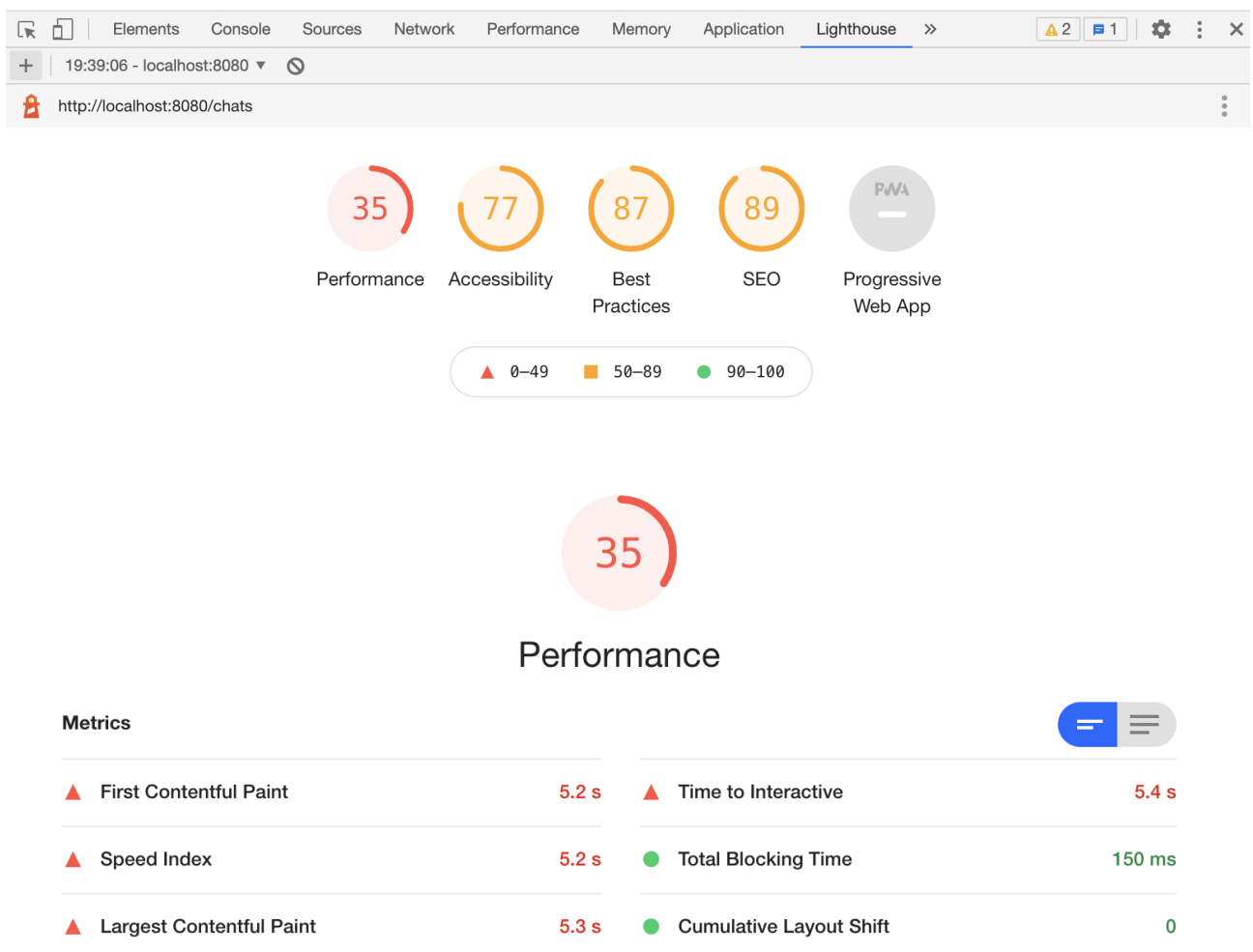
1. По возможности устранить узкие места (bottleneck).
2. Используйте мемоизацию в компонентах и чистые компоненты.
3. Не занимайтесь излишней оптимизацией. Оборачивание в React.memo всех компонентов подряд может привести к обратному результату - ресурсы, затраченные на мемоизацию и выполнение проверок, окажутся больше, чем сэкономленные за счет этих оптимизаций. Пользуйтесь React Profiler, чтобы определить, какой компонент рендерится слишком часто или слишком долго. Сравнивайте результаты работы профайлера до и после внесения оптимизаций.

Lighthouse

Lighthouse - еще одно полезное расширение для Chrome. Оно генерирует отчет о работе вашего сайта с точки зрения пользователя - с его помощью можно оценить насколько быстро работает для пользователей наше приложение. Взглянем на его работу и используемые метрики:



В результате работы дополнения получаем следующий отчет:



Рассмотрим основные пункты в разделе Performance:

1. First contentful paint - время до рендера первого текста или изображения.
2. Speed index - индекс скорости, рассчитывается, исходя из скорости наполнения страницы элементами и времени до окончания первого рендера.
3. Largest contentful paint - время, через которое отрендерен самый большой (визуально) элемент на странице).
4. Time to interactive (TTI) - время от начала загрузки, через которое пользователь может взаимодействовать со страницей.
5. Total blocking time (TBT) - сумма длительностей временных промежутков между First contentful paint и TTI (учитываются интервалы более 50 миллисекунд).
6. Cumulative layout shift - показатель, учитывающий перемещение элементов внутри окна.

Подробнее о показателях Lighthouse можно узнать из [их документации](#).

Для улучшения пользовательского опыта (ускорение первой загрузки и работы приложения в целом) и, как следствие, увеличения показателей раздела Performance в Lighthouse, применяют, в первую очередь, разделение кода. Вместо подгрузки одного большого бандла сперва пользователь получает небольшой кусок (chunk), необходимый для начала работы с приложением, а затем, по мере необходимости, подгружается остальной код. Это обеспечивается, в первую очередь, корректной архитектурой проекта и настройкой webpack. Со стороны Реакта используется “ленивая загрузка” модулей - напр., [React.lazy](#).

Внимание!

При генерации отчета как через Lighthouse, так и через Performance используйте production сборку. В этой версии сборки приложения, по сравнению с development-версией, webpack производит дополнительные оптимизации.

Глоссарий

1. **Unit-тесты** — модульные или блочные — тесты, направленные на проверку работы минимальной составляющей программы (функции, компонента) в отрыве от остальных составляющих.
2. Чистый компонент - компонент, обернутый в React.memo (для функциональных компонентов) или наследующий от PureComponent (для классовых). Реализует сравнение пропсов (а для классовых компонентов - и стейта) перед рендером и может приостановить обновление компонентов.
3. Разделение кода - подход, при котором бандл загружается пользователем по частям, в несколькими запросами.
4. Lighthouse - расширение для Chrome, позволяющее измерять производительность приложения.

5. Профайлинг - измерение производительности приложения.
6. Снепшот - здесь - снимок состояния объекта, в т.ч. элемента Реакт, используемый в тестах.

Домашнее задание

1. Написать тесты для (как минимум) одного редьюсера, и одного презентационного компонента.
2. Написать тест для (как минимум) одного компонента-контейнера.
3. Установить Lighthouse. Сгенерировать отчет. Проанализировать и исправить ошибки (воспользуйтесь советами Lighthouse, указанными в отчете).
4. * Провести профайлинг приложения. Найти (если есть) излишние рендеры. Исправить их с помощью чистых компонентов.

Дополнительные материалы

1. [Статья «Почему я перестал использовать снэпшоты»](#).
2. [Статья "Профилирование React приложений"](#)
3. [Статья Дэна Абрамова "Before you memo"](#)
4. [Моки в Jest](#)

Используемые источники

1. [Официальная документация jest](#).
2. [Официальная документация testing-library](#).
3. [Документация Lighthouse](#)
4. [Документация React Profiler](#)
- 1.