Юнит-тестирование скриншотами: преодолеваем звуковой барьер

Роман Дворнов
Avito

Москва, 2017

- Работаю в Avito
- Open source: basis.js, CSSO, component-inspector, csstree, rempl и другие

Доклад-история
про поиски инженерных решений

Вкратце

- Делать велосипеды не всегда плохо
- Если не сдаваться – решение найдется
- Как полезно разобраться в графических форматах
- Ускорять можно не только оптимизируя код
- Экспериментируя можно получить неожиданные результаты (интрига)


В нас пропал дух авантюризма

Тестирование скриншотами

Тестирование бывает разным

- Юнит-тесты
- Функциональное
- Интеграционное
- ...

Говорим про решение для юнит-тестов компонент/блоков

Основные требования

- Просто
- Быстро
- Дешево

Возможные решения

- Использовать готовые сервисы
- Использовать готовые инструменты (Gemini)
- Написать свое 🤗

Сервисы нам не подходят,
на то "есть причины"

Готовые инструменты

Обычно ориентированы на посещение урлов и снятие скриншотов

Инструменты: Gemini

Вероятно крутая штука, но похоже на космический корабль...

Мой быстрый тест
(проверка 100 изображений 282x200):
1 мин 57 сек

Стали делать своё 🤗

Свое решение: просто

```
test('button', async () => {
    const snap = snapshot(<Button>Button</Button>);

    expect(snap.snapshot).toMatchSnapshot();
    expect(await snap.screenshot).toMatchSnapshotImage();
});
```

Свое решение: быстро

- Сравнение скришотов (800x600)
  - ~0ms/скриншот, если совпадают
  - ~100ms/скриншот, если есть разница
- Обновление 25ms/скриншот (800x600)

Делаем сами: план

- Сгенерировать статичную разметку с необходимыми стилями и ресурсами
- Загрузить разметку в браузер и сделать скриншот
- Сравнить скриншот с эталонным изображением

Генерация разметки

Генерация разметки: план

- Генерируем HTML компонента
- Определяем зависимости
  - Находим стили, обрабатываем, включаем
  - Находим изображения, обрабатываем, инлайним
- Получаем HTML документа без локальных зависимостей

Генерация HTML

Способ зависит от стека

В нашем случае – React

Генерация HTML компонента: React

- react-dom/server + jsdom

```
const ReactDOMServer = require('react-dom/server');

const html = ReactDOMServer.renderToStaticMarkup(jsx);
```

Пока все просто 😉

Генерация CSS

Генерация CSS: план

- Найти CSS файлы
- Преобразовать
  - Заменить локальные ссылки на инлайн
  - Избавиться от динамических частей
- Склеить

Способ зависит от стека

Jest / Babel / CSS Modules

Поиск CSS

Поиск CSS файлов: CSS Modules

- В CSS Modules CSS подключается как обычный модуль:

```
import 'path/to/style.css';
require('path/to/style.css');
```

- Нужно перехватить все вызовы import/require() для CSS и сохранить пути

Напишем свой плагин 😉

Подключаем плагин: в настройках Jest

```
"jest": {
    ...
    "transform": {
        "\\.js$": "./scripts/jest-transform.js"
    },
    ...
}
```

Подключаем плагин: jest-transform.js

```
const { createTransformer } = require('babel-jest');

module.exports = createTransformer({
    ...
    plugins: [
        ...
        'path/to/collect-styles-path.js',
        ...
    ]
});
```

Поиск CSS: Пишем плагин для Babel

- `import '*.css'` → `require('*.css')`
- `require('*.css')` →

```
(function(){
    global.includedCssModules = global.includedCssModules || [];

    if (includedCssModules.indexOf('path/to/style.css') === -1) {
        includedCssModules.push('path/to/style.css');
    }

    return { ... original exports ... };
})()
```

Код плагина полностью (52 SLOC)

На момент генерации разметки компонента в includedCssModules будут пути всех подключенных файлов стилей

Обработка CSS

Обработка CSS: план

- Инлайн ресурсов
- Выключаем динамику

Обработка CSS: инлайн ресурсов

Напишем свой плагин 😉

Подключаем плагин: jest-transform.js

```
const { createTransformer } = require('babel-jest');

module.exports = createTransformer({
    ...
    plugins: [
        ...
        ['css-modules-transform', {
            processCss: 'path/to/process-css.js',
            ...
        }],
        ...
    ]
});
```

CSSTree

CSS парсер (и не только)

github.com/csstree/csstree

- Быстрый
- Детальный
- Толерантен к ошибкам

Инлайн ресурсов: трансформация CSS

```
const path = require('path');
const csstree = require('css-tree');
module.exports = function(data, filename) {
    const ast = csstree.parse(data);

    csstree.walk(ast, function(node) {
        if (node.type === 'Url') {
            const { type, value } = node.value;
            const url = type === 'String' ? value.substring(1, value.length - 1) : value;

            node.value = {
                type: 'Raw',
                value: inlineResource(url, path.dirname(filename))
            };
        }
    });

    return csstree.translate(ast);
}
```

«наивная» реализация

TODO: @imports & edge cases

Инлайн ресурсов: url → dataURI

```
const fs = require('fs');
const mime = require('mime');

function inlineResource(uri, baseURI) {
    const filepath = path.resolve(baseURI, uri);
    const data = fs.readFileSync(filepath);
    const mimeType = mime.getType(filepath);

    return 'data:' + mimeType + ';base64,' + data.toString('base64');
}
```

Инлайн ресурсов в 26 SLOC*

* Source Lines Of Code

Обработка CSS: избавляемся от динамики

Проблема:

Динамика (например, анимация) приводит к разным скриншотам – ее необходимо «выключить»

Источник динамики

- CSS Transitions
- CSS Animations
- Каретка в полях ввода
- GIF
- ...

Выключаем: CSS Transitions

```
*, *::after, *::before {
    transition-delay: 0s !important;
    transition-duration: 0s !important;
}
```

Переводит анимацию
в финальное состояние

Выключаем: CSS Animations

```
*, *::after, *::before {
    animation-delay: -0.0001s !important;
    animation-duration: 0s !important;
    animation-play-state: paused !important;
}
```

Переводит анимацию
в финальное состояние

Выключаем: CSS Animations

```
*, *::after, *::before {
    animation-delay: -0.0001s !important;
    animation-duration: 0s !important;
    animation-play-state: paused !important;
}
```

Иначе не работает в Safari 🤗

Выключаем: CSS Animations

```
*, *::after, *::before {
    animation-delay: -0.0001s !important;
    animation-duration: 0s !important;
    animation-play-state: paused !important;
}
```

Не дает анимации проигрываться

Выключаем: Каретка

```
*, *::after, *::before {
    caret-color: transparent !important;
}
```

Chrome 57, Firefox 53, Opera 44, Safari TP38

Для других браузеров будем придумывать что-то еще

GIF

Делаем статичным

Задача:

Сделать GIF статичным,

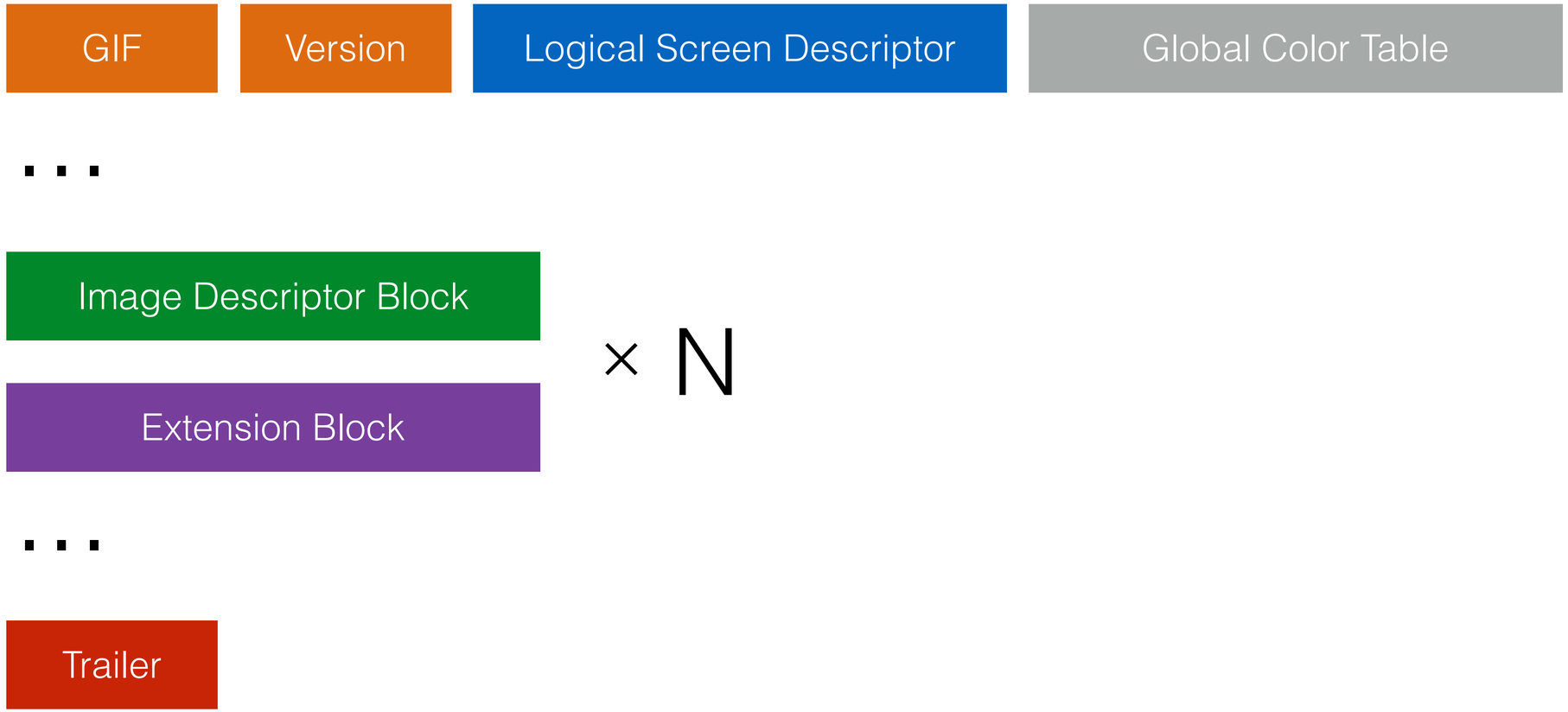
то есть оставить один кадр

Но после пары часов
поиска подходящего пакета,
решил попробовать написать сам 🤓

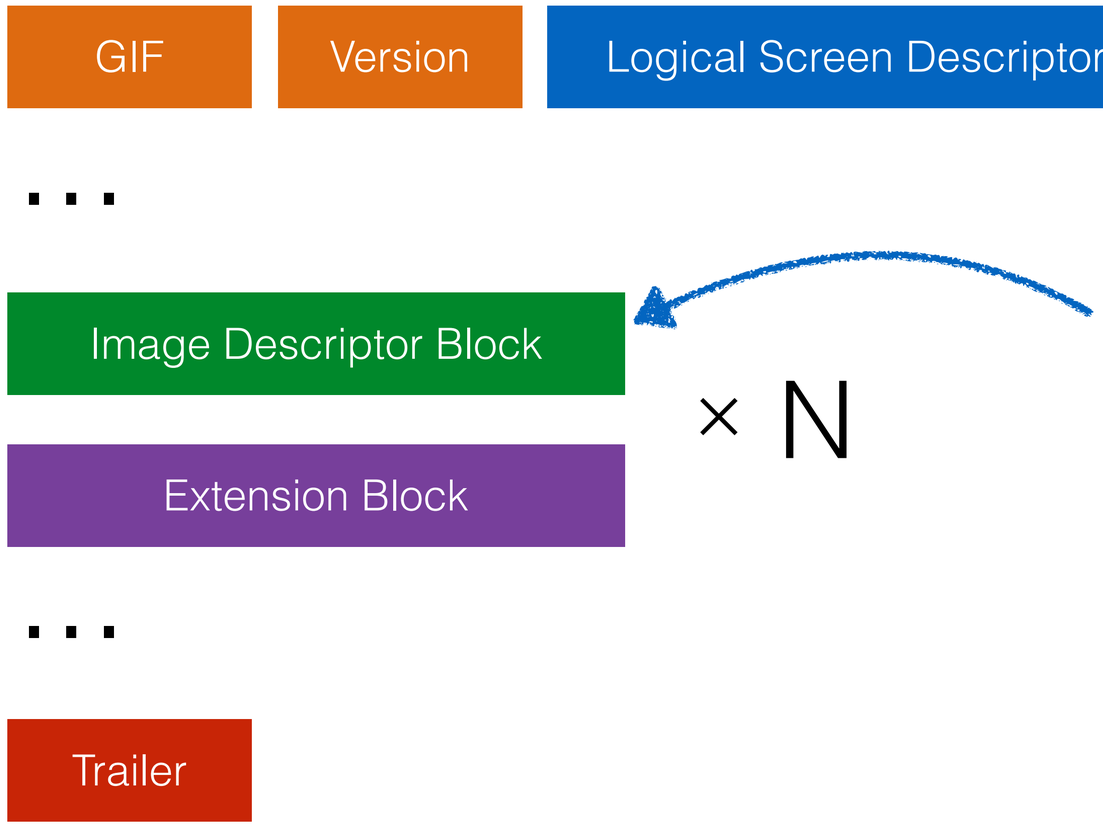
GIF

- Wikipedia
  en.wikipedia.org/wiki/GIF
- GIF89a Specification
  www.w3.org/Graphics/GIF/spec-gif89a.txt

Структура GIF



Структура GIF



Чтобы сделать GIF статическим, необходимо пройтись по блокам и оставить только первый Image Descriptor Block

Решение в 78 SLOC

написано за пару часов

Gist с кодом

Склейка CSS

Как работает Jest?

- Запускает несколько потоков (worker'ов)
- Каждый файл теста запускается в одном из потоков, в своем контексте
- Данные между контекстами не шарятся

Решение

- При преобразовании CSS файла пишем результат во временный файл-карту
- У каждого потока (процесса) свой файл
- Определенно есть решение лучше, но пока не нашли

Сохраняем результирующий CSS

```
function processingCSS(data, filename) {
  // получаем CSS

  // сохраняем результат
  const data = JSON.parse(fs.readFileSync(TMP_STYLES_FILE, 'utf8'));
  data[filename] = css;
  fs.writeFileSync(TMP_STYLES_FILE, JSON.stringify(data));

  return css;
}
```

Используем

```
function generateSnapshot(data, filename) {
    ...

    // сохраняем результат
    const cssMap = JSON.parse(fs.readFileSync(TMP_STYLES_FILE, 'utf8'));
    const styles = includedCssModules.map(path => cssMap[path] || '');

    ...
}
```

Собираем все вместе

Собираем все вместе

```
const htmlForScreenshot = `
    <html>
    <head>
        <style>
            *, *::after, *::before {
                /* убираем динамику */
            }
        </style>
        <style>${styles}</style>
    </head>
    <body>${html}</body>
    </html>
`;
```

Итоги

- Добились своего
- Решения не идеальные, но работают
- Можно сделать лучше (надежнее) но нужно глубже погружаться в инструменты

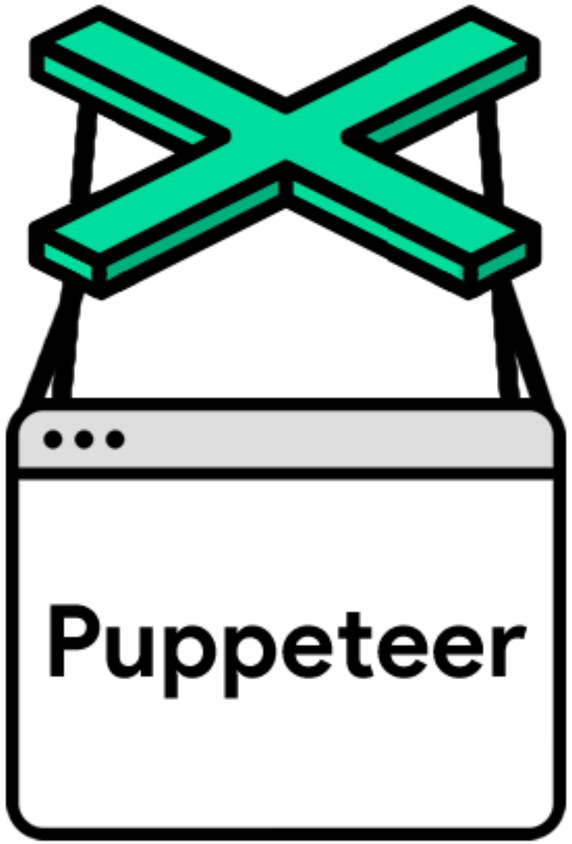
У нас все готово 🤘

Переходим к получению скриншота...

Создание скриншота

Создание скриншота – сегодня

- Headless браузеры – Chrome, Firefox
- Все современные браузеры поддерживают WebDriver
- Нужно немного кода чтобы завелось – достаточно материала по теме



github.com/GoogleChrome/puppeteer

Делаем скриншот

Упрощенный код

```
const puppeteer = require('puppeteer');
const browser = await puppeteer.launch();

async screenshot(html) => {
  const page = await browser.newPage();
  await page.setContent(html);
  const result = await page.screenshot({ fullPage: true });
  await page.close();

  return result;
}
```

Работает 

Не совсем 😔

Разные версии браузера, разные ОС

=

разный результат

Решение:

Делаем микро-сервис,
POST-запрос с кодом → скриншот

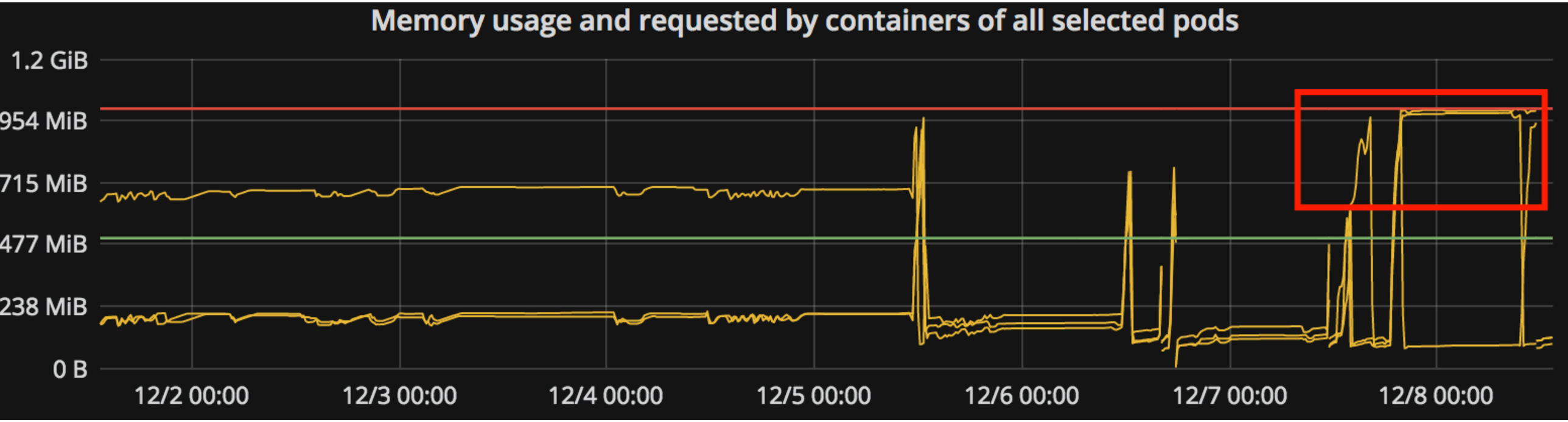
Плюсы

- Нет зависимости от машины, где запускаются тесты
- Не требуется настроек для нового проекта
- В целом работает быстрее (всегда разогретый браузер)

Минусы

- Нужно следить за "здоровьем" сервиса – утечки памяти и баги могут его сломать
- Если ломается сервис, то тестирование скриншотами не возможно
- В пиках нагрузки замедляется прохождение тестов (почти не проблема, у нас облако)

Как выглядит «смерть» сервиса



Side effect:

Мы «научили» сервис отдавать скриншот по url + CSS selector

Оказалось весьма полезно

- Вставляем в документацию/описание URL к сервису
- Получаем скриншот страницы/блока, который будет всегда актуальным
- ...
- PROFIT!

Областей применения гораздо больше,
планируем развивать дальше
(но это другая история)

Итак, у нас есть скриншоты 🤘

Осталось сравнить результаты...

Сравнение изображений

Мы используем Jest для тестирования компонент

Тестирование снепшотами

```
const ReactTestRenderer = require('react-test-renderer');

test('snapshot example test', () => {
    const snapshot = ReactTestRenderer.create(
        <Button>test</Button>
    );

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

toMatchSnapshot()

- Приводит проверяемое значение к строке
- Если тест новый – сохраняет значение как есть
- Если тест не новый – сверяет новое значение с прежним, в случае не совпадения выводит ошибку
- Если тест пропал, то выводит что снепшот устарел

Сравнение изображений

- Jest не имеет встроенных инструментов для сравнения изображений
- Хороший старт jest-image-snapshot

jest-image-snapshot: подключение

```
const { toMatchImageSnapshot } = require('jest-image-snapshot');

expect.extend({
    toMatchImageSnapshot
});
```

jest-image-snapshot: использование

```
const snapshot = require('...'); // самописный хелпер

test('snapshot example test', async () => {
    const snap = snapshot(<Button>test</Button>);

    expect(snap.snapshot).toMatchSnapshot();
    expect(await snap.screenshot).toMatchImageSnapshot();
}
```

Проблемы

- Недостаточно быстрый (недавно стал быстрей)
- Не учитывает режим (mode), в котором запущен Jest, всегда создает diff-изображения
- Плохо работает с счетчиками (кол-во снепшотов)
- Не умеет считать и удалять устаревшие изображения

Ключевая проблема – производительность:
Сравнение двух изображений 800x600
занимает ~1,5-2 сек

Сделали форк и починили проблемы 🤓

Сравнение изображений

С чего начинали (~300 скриншотов):

Инициализация: 10-20 сек

Проверка: 4.5 мин

(сравнение в 3 потока)

«Вскрытие» показало:
каждый раз изображения сравниваются через blink-diff (попиксельно)

Что нужно для сравнения

- Декодировать 2 изображения

  800x600 = 480 000 пикселей x 2

  4 байт x пиксель = ~2Mb x 2

- Пройтись по массивам и сравнить поэлементно

Библиотеки сравнения изображений (png)

- blink-diff ~1.5-2 сек
- pixelmatch ~0.5-0.7сек
- looks-same (от Gemini)

Как сделать быстрее?

Инсайт

При повторных проходах большая часть изображений совпадает

Что получается

- Можно сравнить файлы без декодирования
- PNG хорошо сжимается, нужно сравнивать несколько килобайт вместо мегабайтов
- Если файлы не равны, только тогда приступать к сравнению попиксельно

Как быстро сравнить два файла?

Хеш!

```
const crypto = require('crypto');

function getHash(data) {
    return crypto
        .createHash('md5')
        .update(data, 'utf8')
        .digest('hex');
}

getHash(fs.readFileSync('a.png')) === getHash(fs.readFileSync('b.png'));
//   4Kb - 58,773 ops/sec
// 137Kb -  2,186 ops/sec
```

Хеш!

```
const crypto = require('crypto');

function getHash(data) {
    return crypto
        .createHash('md5')
        .update(data, 'utf8')
        .digest('hex');
}

getHash(fs.readFileSync('a.png')) === getHash(fs.readFileSync('b.png'));
//   4Kb - 58,773 ops/sec
// 137Kb -  2,186 ops/sec
```

Overkill!

Все проще!

```
const bufferA = fs.readFileSync('a.png');
const bufferB = fs.readFileSync('b.png');

bufferA.equals(bufferB);
//   4Kb – 3,273,114 ops/sec
// 137Kb –   148,986 ops/sec
```

Все проще!

```
const bufferA = fs.readFileSync('a.png');
const bufferB = fs.readFileSync('b.png');

bufferA.equals(bufferB);
//   4Kb – 3,273,114 ops/sec
// 137Kb –   148,986 ops/sec
```

В 50-70 раз быстрее!

Сравниваем попиксельно (своими руками)

Считаем кол-во разных пикселей

```
function countImageDiff(actualImage, expectedImage) {
    const actual = png.decode(actualImage);
    const actualPixels = new Uint32Array(actual.data.buffer);
    const expected = png.decode(expectedImage);
    const expectedPixels = new Uint32Array(expected.data.buffer);
    let count = 0;

    for (let i = 0; i < actualPixels.length; i++) {
        if (actualPixels[i] !== expectedPixels[i]) {
            count++;
        }
    }

    return { width: actual.width, height: actual.height, count };
}
```

Считаем кол-во разных пикселей

```
function countImageDiff(actualImage, expectedImage) {
    const actual = png.decode(actualImage);
    const actualPixels = new Uint32Array(actual.data.buffer);
    const expected = png.decode(expectedImage);
    const expectedPixels = new Uint32Array(expected.data.buffer);
    let count = 0;

    for (let i = 0; i < actualPixels.length; i++) {
        if (actualPixels[i] !== expectedPixels[i]) {
            count++;
        }
    }

    return { width: actual.width, height: actual.height, count };
}
```

Считаем кол-во разных пикселей

```
function countImageDiff(actualImage, expectedImage) {
    const actual = png.decode(actualImage);
    const actualPixels = new Uint32Array(actual.data.buffer);
    const expected = png.decode(expectedImage);
    const expectedPixels = new Uint32Array(expected.data.buffer);
    let count = 0;

    for (let i = 0; i < actualPixels.length; i++) {
        if (actualPixels[i] !== expectedPixels[i]) {
            count++;
        }
    }

    return { width: actual.width, height: actual.height, count };
}
```

Считаем кол-во разных пикселей

```
function countImageDiff(actualImage, expectedImage) {
    const actual = png.decode(actualImage);
    const actualPixels = new Uint32Array(actual.data.buffer);
    const expected = png.decode(expectedImage);
    const expectedPixels = new Uint32Array(expected.data.buffer);
    let count = 0;

    for (let i = 0; i < actualPixels.length; i++) {
        if (actualPixels[i] !== expectedPixels[i]) {
            count++;
        }
    }

    return { width: actual.width, height: actual.height, count };
}
```

Для двух изображений 800x600

~100 ms

Генерируем diff-изображение

Как?

- Так же сравниваем попиксельно
- Создаем буфер под результирующее изображение
- Совпадающие пиксели обесцвечиваем и засвечиваем
- Для несовпадающих пишем красный пиксель

Генерируем diff-изображение

```
function findImageDiff(actualImage, expectedImage) {
    ...
    const diff = Buffer.alloc(actual.data.length);
    const diffPixels = new Uint32Array(diff.buffer);

    for (let i = 0; i < actualPixels.length; i++) {
        if (actualPixels[i] !== expectedPixels[i]) {
            count++;
            diffPixels[i] = 0xff0000ff;
        } else {
            // обесцвечиваем и засвечиваем
        }
    }

    return { ..., actual, expected, diff };
}
```

Засвечивание пикселя

```
const pixel = actualPixels[i];

const sum = ((pixel >>  0) & 0xff) +   // red
            ((pixel >>  8) & 0xff) +   // green
            ((pixel >> 16) & 0xff);    // blue

const gray = (255 - 64 + 64 * (sum / (255 + 255 + 255))) | 0;

                // alpha                    blue          green        red
diffPixels[i] = (pixel & 0xff000000) | gray << 16 | gray << 8 | gray;
```

«Загоняем» пиксели
в этот диапазон

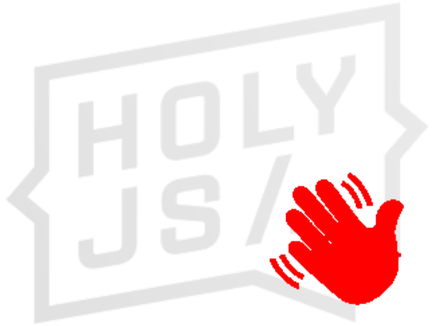
Сохраняем diff-изображение

```
const png = require('fast-png');
const {
    width,
    height,
    actual,
    expected,
    diff
} = findImageDiff(actualImage, expectedImage);

fs.writeFileSync(diffImageFilename, png.encode({
    width,
    height: height * 3,
    data: Buffer.concat([
        expected.data,
        diff,
        actual.data
    ])
}));
```

Результат

Эталонное изображение



Diff (разница)

Новое изображение

Для двух изображений 800x600

~250 ms

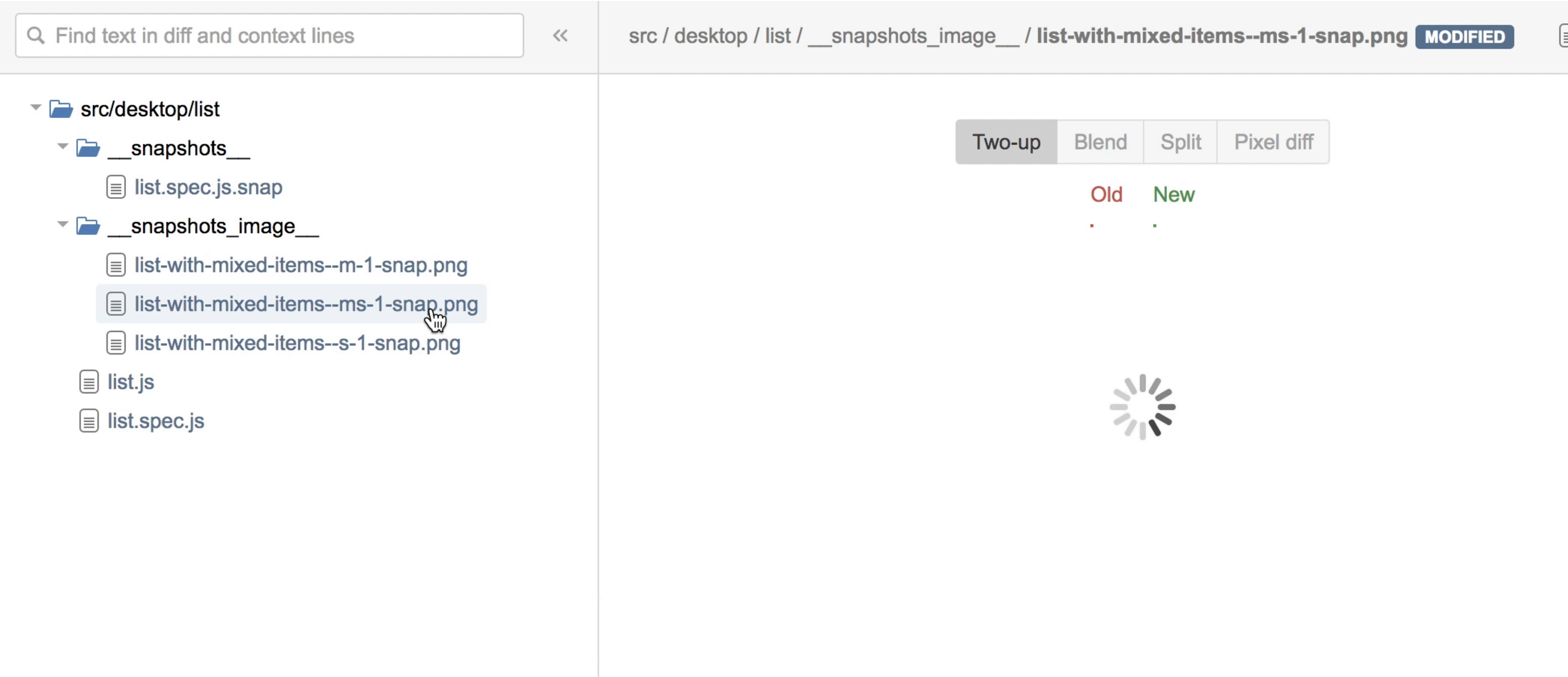
Идея для «стартапа»

Запилить decode, сравнение и decode PNG на WebAssembly – вероятно будет быстрее

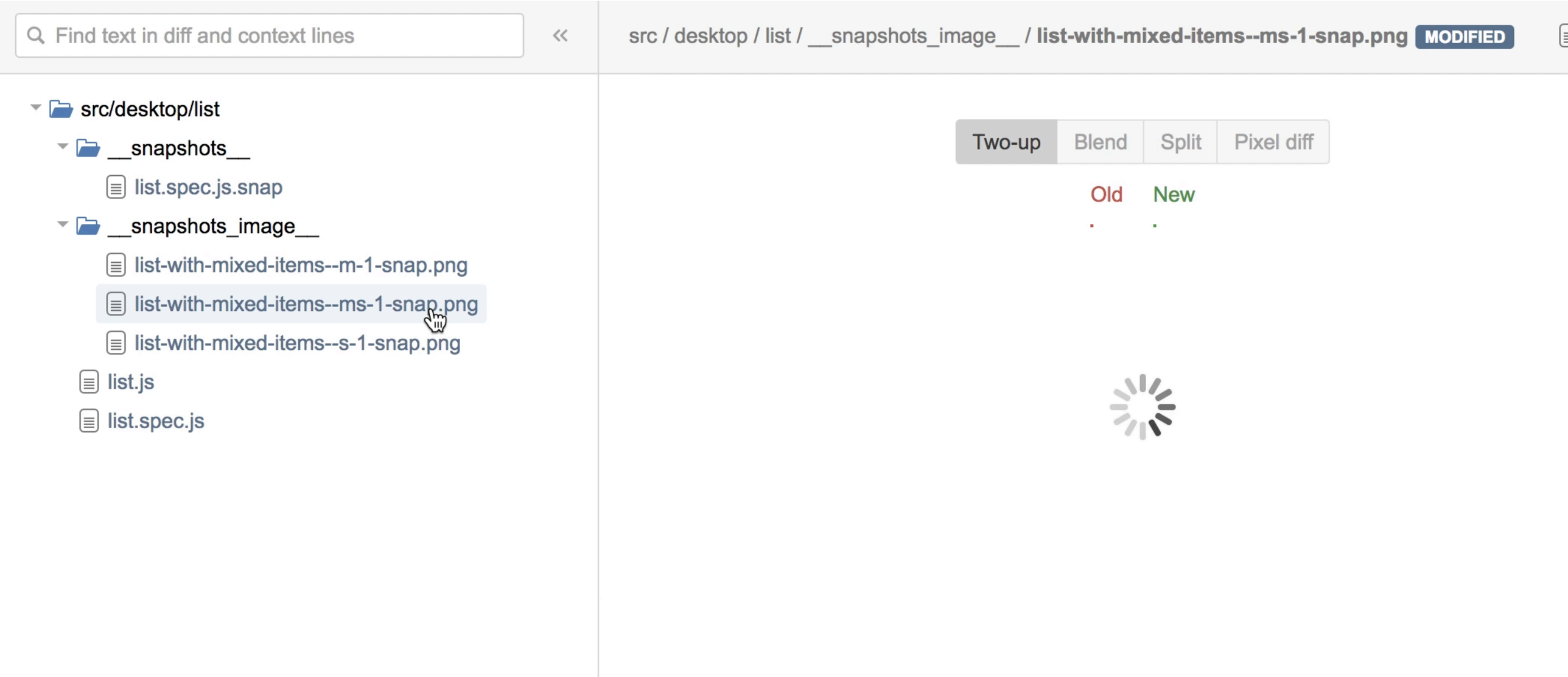
Важно ☝

Делать diff-изображение не нужно, т.к. многие инструменты имеют встроенное сравнение изображений

Например


Find text in diff and context lines
src / desktop / list / __snapshots_image__ / list-with-mixed-items--ms-1-snap.png MODIFIED
src/desktop/list
__snapshots__
list.spec.js.snap
__snapshots_image__
list-with-mixed-items--m-1-snap.png
list-with-mixed-items--ms-1-snap.png
list-with-mixed-items--s-1-snap.png
list.js
list.spec.js
Two-up
Blend
Split
Pixel diff
Old
New

Например


Find text in diff and context lines
src / desktop / list / __snapshots_image__ / list-with-mixed-items--ms-1-snap.png MODIFIED
src/desktop/list
__snapshots__
list.spec.js.snap
__snapshots_image__
list-with-mixed-items--m-1-snap.png
list-with-mixed-items--ms-1-snap.png
list-with-mixed-items--s-1-snap.png
list.js
list.spec.js
Two-up
Blend
Split
Pixel diff
Old
New

Создаем diff-изображение, только если пользователь явно попросил об этом

```
jest --expand
```

Можно ли сделать еще быстрее?

Инсайт

Для одного и того же кода – тот же скриншот

В большинстве случаев

можно не делать запрос к сервису

План

- Генерируем HTML код
- Если есть изображение и его код совпадает с сгенерированным – не делаем запроса, используем имеющееся изображение
- Если код различается или нет изображения – делаем запрос к сервису
- Сохраняем код, которым получаем изображение

Где хранить код?

Можно в самом изображении :)

Похожий сценарий 😉

- Читаем Wikipedia
  en.wikipedia.org/wiki/Portable_Network_Graphics
- Читаем код библиотек
- ...
- Умеем работать с PNG

В PNG может хранится не только графическая информация
Можно хранить произвольные данные

Решение (~70 SLOC)

Не полетело, так как файлы получаются разными но визуальной разницы нет

В результате, храним код скриншота отдельным файлом

Что дало решение?

Проверка ~300 изображений 800x600

45-50 сек → 12-15 сек

(полная проверка, локальный запуск)

Бонус: значительно снизили нагрузку на сервис скриншотов

Хранение изображений в GIT

Проблема

Бинарные данные хранятся в истории целиком – история быстро растет

GIT LFS

GIT LFS

- Включаем в репозитарии GIT LFS
- Определяем файлы, которые нужно хранить в хранилище в .gitattributes

```
*.png filter=lfs diff=lfs merge=lfs -text
```

- Инициализируем: `git lfs install`
- PROFIT: в истории хранятся только хеши файлов, локально реальные файлы, `git push/pull` работают прозрачно

GIT LFS: профит

- В истории git'a хранятся только хеши файлов
- Локально хранятся реальные файлы
- `git push/pull` работают как обычно и делают всю магию

CI или beyond the frontend

Прохождение всех unit-тестов на CI

~4 сек 🤘

Но полный время выполнения на CI

~3,5 мин 😞

Полез ковырять 🤓

(с нулевыми знаниями Teamcity)

Что происходит внутри CI

- git checkout
- npm install
- jest --ci
- eslint
- stylelint

Что происходит внутри CI

- git checkout
- npm install
- jest --ci
- eslint
- stylelint

Тюним настройки, добавляем кеши

3,5 мин → <30 сек

Результаты

Время

- Локально прохождение всех юнит тестов
  несколько минут → 12-15 секунд
- Прохождение всех проверок на CI:
  несколько минут → 20-30 секунд

Свой плагин для jest

- Быстрый, поддерживает digest
- Учитывает режим (mode), в котором запущен Jest
- Создает diff-изображения только если `--expand`
- Правильно работает с счетчиками
- Умеет считать и удалять устаревшие изображения

Будущее плагина:

- План А: попробуем предложить в Jest
- План Б:
  - Либо смержим с jest-image-snapshot
  - Либо опубликуем как самостоятельный пакет

Про остальное

Как обкатаем подумаем о том,

чтобы опубликовать в Open Source

Подводим итоги

Итоги

- Неколько плагинов:
  для Babel, CSS Modules и Jest
- Все под контролем, мы знаем как работают все части решения
- Время unit-тестов не «пострадало»
  от добавления тестирования скриншотами

Время

Без скриншотов

```
Test Suites: 38 passed, 38 total
Tests:       422 passed, 422 total
Snapshots:   413 passed, 413 total
Time:        11.579s
```

Со скриншотами

```
Test Suites: 38 passed, 38 total
Tests:       422 passed, 422 total
Snapshots:   741 passed, 741 total
Time:        11.752s
```

318 изображений

проверено, что они актуальны

Работа над решениями привела

к новым областям применения скриншотов

(использование в документации/инструментах)

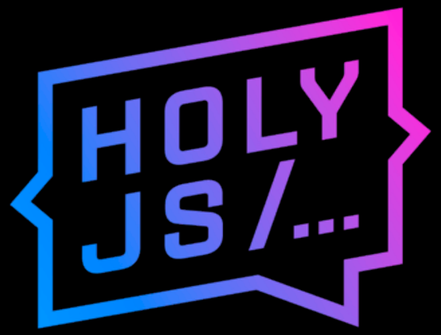
Чему мы научились

- Как работает Jest внутри
- Как устроены форматы GIF и PNG
- Лучше понимание Buffer API
- Как настраивать Teamcity
- Как сделать юнит-тесты скриншотами быстрыми :)

Юнит-тестирование скриншотами может быть со скоростью пули

Не бойтесь делать

свои решения!



Спасибо!

Роман Дворнов

@rdvornov

rdvornov@gmail.com