НЕТОЛОГИЯ

ТЕСТИРОВАНИЕ И СБОРКА

АЛЕКСАНДР ШЛЕЙКО

АЛЕКСАНДР ШЛЕЙКО

Программист в Яндекс

a.shleyko+a@yandex.ru vk.com/shleiko @dustyo_O

ПЛАН ЗАНЯТИЯ

1. Unit-тестирование
2. Jest
3. Покрытие кода
4. Mock'и
5. Webpack

UNIT-ТЕСТИРОВАНИЕ

ЗАЧЕМ НУЖНЫ АВТО-ТЕСТЫ?

Зачем нужны авто-тесты, мне проще проверить руками

Зачем нужны авто-тесты, у меня итак всё работает

ДЕЙСТВИТЕЛЬНО, ЗАЧЕМ?

Авто-тесты - это возможность обезопасить себя от потенциальных ошибок (при создании нового кода или модификации существующего).

Это некая гарантия того, что то, что работало до этого - не сломалось, и то, что мы пишем сейчас - работает так, как мы задумываем.

BEST PRACTICES

В современном мире разработки написание авто-тестов считается одной из лучших практик создания поддерживаемого и качественного кода.

КАК И ЧТО ТЕСТИРОВАТЬ?

Тестирование - отдельная большая область знаний, со своими методами, подходами и теорией.

На самом базовом уровне: запускаем программу или отдельный её кусочек (например, функцию) и сравниваем полученный результат с тем, что должен был получиться.

Результат совпадает - всё ок, нет - ошибка.

Если начнёте с этого - будет уже большой шаг вперёд, после чего нужно ознакомиться с тест-анализом, тест-дизайном и комбинаторикой.

JEST

ФРЕЙМВОРК ТЕСТИРОВАНИЯ

Чтобы нам не делать этого каждый раз вручную, нам нужен инструмент, который будет запускать наши функции, сравнивать результат и собирать статистику.

Инструментов достаточно много, мы с вами будем рассматривать Jest.

УСТАНОВКА

Для установки Jest выполним следующую команду:

```
$ npm install --save-dev jest babel-jest babel-core@^7.0.0-bridge.0
```

Пропишем скрипт `test`:

```
"scripts": {
  "start": "live-server dist",
  "test": "jest",
  "lint": "eslint .",
  "build": "babel src -d dist"
},
```

ОБЩИЙ ВИД ТЕСТА

```
test('<описание того, что проверяем>', () => {
  // Функция проверки

  // 1. Выполняем нужные нам действия
  TODO:

  // 2. Проверяем результат с помощью:
  expect(<что получили>).toBe(<что должно быть>);
})
```

ПРИМИТИВНЫЙ ТЕСТ

```
1  test('should add two numbers', () => {
2    const received = 1 + 1;
3    const expected = 2;
4
5    expect(received).toBe(expected);
6  })
```

ESLINT & JEST

Чтобы ESLint не ругался на Jest, мы можем либо добавить каталог с тестами в `.eslintignore` (плохая идея), либо прописать Jest в секцию `env` файла `.eslintrc.json`:

```
"env": {
    ...
    "jest": true
}
```

ЗАПУСК АВТО-ТЕСТОВ

```
$ npm test
```

```
PASS test/app.test.js
  ✓ shoud add two numbers (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.762s, estimated 1s
Ran all test suites.
```

FAIL

В случае, если тесты завершатся с ошибкой, мы увидим:

```
FAIL test/app.test.js
  ✕ shoud add two numbers (9ms)

  ● shoud add two numbers

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

    Expected: 2
    Received: 4

      1 | test('shoud add two numbers', () => {
    > 2 |   expect(2 + 2).toBe(2);
        |                 ^
      3 | });

      at Object.toBe (test/app.test.js:2:17)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.78s, estimated 1s
Ran all test suites.
npm ERR! Test failed.  See above for more details.
```

ЗАДАЧА

Перед нами стоит следующая задача: написать функцию, которая рассчитывает сумму покупок в магазине.

Покупки приходят в следующем виде:

```
[
  {id: 1, name: '...', price: 100, count: 3},
  {id: 2, name: '...', price: 55, count: 2},
]
```

Давайте попробуем написать эту функцию и авто-тесты для неё.

ФУНКЦИЯ

Начнём с самой простой реализации (файл `script.js`):

```
function calculateTotal(purchases) {
  let result = 0;
  for (const purchase of purchases) {
    result += purchase.price * purchase.count;
  }

  return result;
}
```

ТЕСТ

Подумаем, как должен выглядеть тест для этой функции:

```
test('should calculate total for purchases', () => {
  const input = [
    {
      id: 1, name: '...', price: 33, count: 3,
    },
    {
      id: 2, name: '...', price: 55, count: 2,
    },
  ];
  const expected = 1099;

  const received = calculateTotal(input);

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

СИСТЕМЫ МОДУЛЕЙ

Раньше в JS не было такого понятия, как модуль и браузер фактически «объединял» все подключаемые скрипты.

Это очень неудобно при разработке больших приложений, поэтому были разработаны системы, обеспечивающие модульность.

СИСТЕМЫ МОДУЛЕЙ

На текущий момен наиболее распространёнными являются следующие системы модулей:

- **CommonJS** – система модулей, нативно поддерживается на платформе Node.js;
- **ES Modules** – система модулей, нативно поддерживаются в браузерах (текущий статус поддержки).

Под модулем мы будем понимать js-файл (достаточно упрощённое представление, но достаточное для нас на данном этапе).

Более подробно про модули мы поговорим на лекции, посвящённой модулям, сейчас же нам нужно понять ключевые моменты.

ЗАЧЕМ НАМ ДВЕ?

Большинство инструментов для JS написаны с использованием платформы Node.js, поэтому для них придётся использовать либо CommonJS, либо Babel (который обеспечит поддержку импорта в стиле ES Modules).

Поэтому придётся научиться использовать оба.

EXPORT / MODULE.EXPORTS

Если мы хотим сделать имя (функцию, переменную либо объект) доступным из нашего модуля, то в ES Modules:

```
export <some_name>;
export function <some_function>() { ... };
```

В CommonJS:

```
module.exports = {
  <some_name>: <some_object>,
  <some_function>: function() { ... }
};
```

IMPORT / REQUIRE

Если мы хотим использовать имя, экспортированное из другого модуля, в своём модуле, то в ES Modules:

```
import { <name> } from '<path_to_module>';
```

В CommonJS:

```
const <name> = require('<path_to_module>').<name>;
```

DEFAULT EXPORT

Если вы экспортируете из модуля всего одно имя, то лучше использовать `default export`:

```
export default function <some_function>() { ... }
```

```
import <some_function> from '<path_to_module>';
```

JEST IMPORT/EXPORT

Jest в связке с Babel у нас настроен таким образом, что поддерживает `import` / `export`, но для этого нужно экспортировать нашу функцию:

```
1  export default function calculateTotal(purchases) {
2    let result = 0;
3    for (const purchase of purchases) {
4      result += purchase.price * purchase.count;
5    }
6
7    return result;
8  }
```

ТЕСТ

```
import puchasesTotal from '../src/js/script';

test('should calculate total for purchases', () => {
  const input = [
    {
      id: 1, name: '...', price: 33, count: 3,
    },
    {
      id: 2, name: '...', price: 55, count: 2,
    },
  ];
  const expected = 209;

  const received = calculateTotal(input);

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

TESTS & GIT

Авто-тесты должны храниться вместе с нашим приложением, как и другие исходники. Таким образом, любой участник нашей команды, меняя что-то в нашем приложении сможет удостовериться, что ничего не сломал.

ИСПОЛЬЗУЕМ `reduce`

Попробуем воспользоваться методом массива `reduce` вместо цикла `for..of`:

```
return purchases.reduce(
  (acc, curr) => acc + curr.price * curr.count,
  0,
);
```

У нас есть авто-тесты, мы можем их запустить, чтобы усдостовериться, что всё работает.

ВАЖНО

Важно понимать, что польза от авто-тестов появляется только тогда, когда вы их регулярно пишите и используете.

А кроме того, постоянно совершенствуетесь в навыке написания тестов.

ПОКРЫТИЕ КОДА

CODE COVERAGE

Code Coverage - метрика, показывающая, насколько наш код покрыт авто-тестами.

```
$ npm test -- --coverage
```

```
PASS test/script.test.js
  ✓ should calculate total for purchases (4ms)

-----------|----------|----------|----------|----------|-------------------|
File       |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------|----------|----------|----------|----------|-------------------|
All files  |      100 |      100 |      100 |      100 |                   |
 script.js |      100 |      100 |      100 |      100 |                   |
-----------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.408s
Ran all test suites.
```

ДОБАВИМ ЛОГИКУ

Пришло время модифицировать нашу функцию: в зависимости от переданного флага к итоговой сумме покупок должна применяться скидка 6.1%:

```
export default function calculateTotal(purchases, applyDiscount = false) {
  ...

  if (applyDiscount) {
     return result * 0.939; // bad practice
  }
}
```

ПОСМОТРИМ НА ПОКРЫТИЕ

```
PASS test/script.test.js
  ✓ should calculate total for purchases (7ms)

-----------|----------|----------|----------|----------|-------------------|
File       |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------|----------|----------|----------|----------|-------------------|
All files  |       80 |    66.67 |      100 |       80 |                   |
 script.js |       80 |    66.67 |      100 |       80 |                 8 |
-----------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.963s
Ran all test suites.
```

В каталоге `coverage/lcov-report` расположен отчёт о покрытии:

```
1
2  1x  export default function purchasesTotal(purchases, applyDiscount = false) {
3  1x    const result = purchases.reduce(
4  2x      (accumulator, current) => accumulator + current.price * current.count,
5          0,
6        );
7
8  1x    if (applyDiscount) {
9          return result * 0.03;
10       }
11
12 1x    return result;
13     }
```

НАПИШЕМ ТЕСТ

```
1  test('should calculate total for purchases with discount', () => {
2    ...
3    const expected = 196.25;
4
5    const received = purchasesTotal(input, true);
6
7    expect(received).toBe(expected);
8  });
```

Тест упал:

```
Expected: 196.25
Received: 196.25099999999998

  31 |   const received = purchasesTotal(input, true);
  32 |
> 33 |   expect(received).toBe(expected);
     |                    ^
  34 | });
  35 |
```

MATCHERS

Jest нам предлагает различные виды проверок (не только на точное соответствие).
Полный перечень Matcher'ов можно найти на странице:
https://jestjs.io/docs/en/expect

В частности, в нашем случае хорошо бы подошёл `toBeCloseTo`.

КАК ПОНЯТЬ, ЧТО ТЕСТОВ ДОСТАТОЧНО?

Тесты должны помогать в разработке а не мешать. Именно они должны показывать, какие условия не протестированы, какие участки кода никогда не используются.

Значит:

1. Либо избыточны;
2. Либо мы не можем сказать, что они работают корректно.

Используйте подход TDD, который позволит уменьшить и количество разрабатываемого кода и количество разрабатываемых тестов.

МОСК'И

MOCKS

Как протестировать функцию, которая взаимодействует с внешним миром (HTTP, файловая система и т.д.)? Неужели на каждый тест будет выполняться отдельный HTTP-запрос на сервер?

Конечно же, нет. Для этого существуют Mock'и.

MOCKS

```
1  import { httpGet } from './http';
2
3  export default function loadUser(id) {
4    // bad practice
5    const data = httpGet(`http://server:8080/users/${id}`);
6    return JSON.parse(data);
7  }
```

MOCKS

```
1  import loadUser from '../src/js/user';
2  import { httpGet } from '../src/js/http';
3
4  jest.mock('../src/js/http');
5
6  beforeEach(() => {
7    jest.resetAllMocks();
8  });
9
10 test('should call loadUser once', () => {
11   httpGet.mockReturnValue(JSON.stringify({}));
12
13   loadUser(1);
14   expect(httpGet).toBeCalledWith('http://server:8080/users/1');
15 });
```

SETUP & TEARDOWN

- `beforeEach`;
- `afterEach`;
- `beforeAll`;
- `afterAll`.

https://jestjs.io/docs/en/setup-teardown

MOCKS

Использование mock'ов - не всегда хорошая идея, т.к. влечёт к избыточному усложнению тестового кода.

JEST EXTENSION

Для VSCode предоставляется плагин Jest за авторством Orta, который в автоматическом режиме перезапускает ваши тесты и отображает статус:

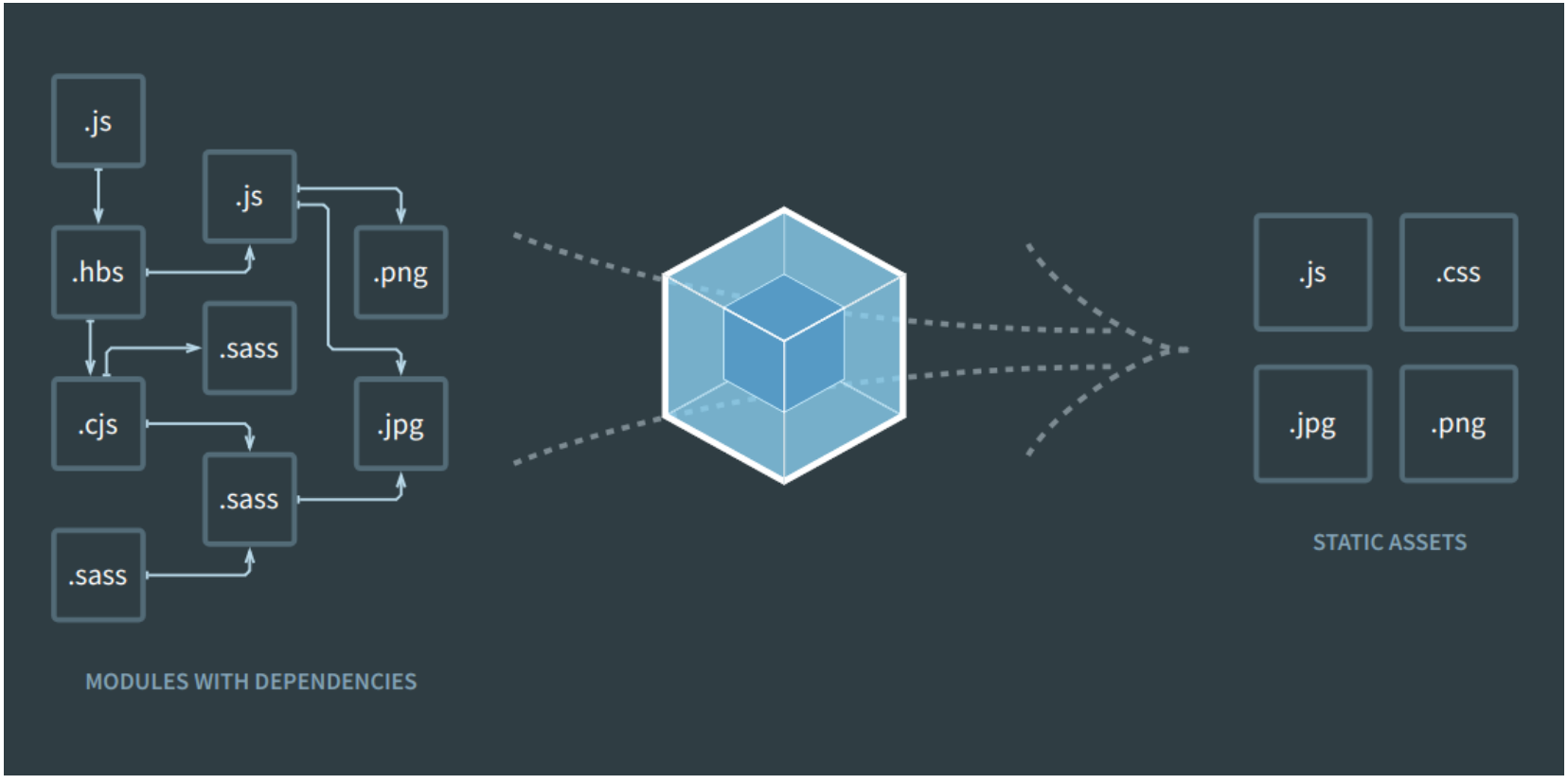
```
test('shoud add two numbers', () => {
  expect(1 + 1).toBe(2);
});
```

А также позволяет отлаживать их:

```
Debug
test('shoud add two numbers', () => {
  expect(2 + 2).toBe(2); // Expected: 2, Received: 4
});
```

WEBPACK

WEBPACK


.js
.js
.hbs
.png
.sass
.cjs
.jpg
.sass
.sass
MODULES WITH DEPENDENCIES
.js
.css
.jpg
.png
STATIC ASSETS

WEBPACK

Webpack - Module Bundler для JS-приложений. Позволяет объединять все ресурсы нашего приложения в Bundle (преобразованные, минимизированные, оптимизированные) и готовые для использования в продакшн-среде.

На сегодняшний день - самый популярный инструмент сборки в мире JS. Содержит интеграции с большинством ругих популярных инструментов.

УСТАНОВКА

Для установки Webpack и поддержки Babel выполним следующую команду:

```
$ npm install --save-dev webpack webpack-cli babel-loader
```

В скриптах заменим `build`:

```
"scripts": {
    ...
    "build": "webpack --mode production"
},
```

СБОРКА

Выполним сборку:

```
$ npm run build
```

Удостоверимся, что появился каталог `dist`, в котором находится минимизированный файл `main.js`.

HTML PLUGIN

Для сборки HTML-файлов нужен отдельный плагин: HTML Plugin

Для установки выполним следующую команду:

```
$ npm install --save-dev html-webpack-plugin html-loader
```

HTML PLUGIN

```
module: {
  rules: [
    {
      test: /\.html$/,
      use: [
        {
          loader: "html-loader"
        }
      ]
    }
  ],
},
plugins: [
  new HtmlWebPackPlugin({
    template: "./src/index.html",
    filename: "./index.html"
  })
]
```

CSS PLUGIN

Для поддержки CSS нам нужен отдельный плагин: Mini CSS Extract Plugin:

```
$ npm  install --save-dev mini-css-extract-plugin css-loader
```

MINI CSS EXTRACT PLUGIN

```
module.exports = {
  module: {
    rules: [
      ...
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader, 'css-loader',
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: '[name].css',
      chunkFilename: '[id].css',
    }),
  ],
```

ПОДКЛЮЧЕНИЕ CSS

Webpack требует использовать спец.синтаксис для построение графа зависимостей (чтобы подключить css):

```
1 | import '../css/styles.css';
```

WEBPACK DEV SERVER

После всех проделанных манипуляций, Live Server нам уже не поможет в разработке, т.к. мы достаточно глубоко интегрировали в свой проект Webpack.

Но решение есть - Webpack Dev Server.

УСТАНОВКА

Для установки выполним следующую команду:

```
$ npm install --save-dev webpack-dev-server
```

И в скриптах заменим `start`:

```
"scripts": {
    "start": "webpack-dev-server --mode development",
    "test": "jest",
    "lint": "eslint .",
    "build": "webpack --mode production"
},
```

ЗАПУСК

Запустим Dev Server и удостоверимся, что Live Reload работает при изменении файлов:

```
$ npm start
```

```
i ｢wds｣: Project is running at http://localhost:8080/
i ｢wds｣: webpack output is served from /
i ｢wdm｣: Hash: 582f4199a90ed6e26bf1
Version: webpack 4.28.3
Time: 941ms
Built at: 2019-01-02 14:54:00
      Asset       Size  Chunks             Chunk Names
./index.html  285 bytes          [emitted]
     main.js    420 KiB    main  [emitted]  main
```

Итоговый настроенный проект вы сможете найти в материалах к этой лекции.

ИТОГИ

ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей, а именно:

1. Jest — позволяет создавать авто-тесты.
2. Webpack — позволяет организовать сборку приложения и настроить dev server.

ВАЖНО

Начиная с сегодняшнего дня во всех домашних заданиях мы будем требовать от вас:

1. Наличия авто-тестов на разрабатываемые функции;
2. 100% покрытия тестируемых функций по строкам;
3. Использования Webpack для сборки ваших проектов.

ПОЛЕЗНЫЕ ССЫЛКИ

- Документация Jest
- Документация Webpack
- Документация Webpack Dev Server

НЕТОЛОГИЯ

Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСАНДР ШЛЕЙКО


a.shleyko+a@yandex.ru


vk.com/shleiko


@dustyo_0