Лекция 11

Test, lint, build, typing, validation, pipeline

By JUSSIAR

Testing

- Зачем они нужны?
- Какие бывают и какие самые удобные?
- Что полезнее всего?
- Фаззинг
- Как строить тесты?
- Базовые принципы TDD
- Моки
- Окружение
- Документация

Зачем нужны тесты?

В первую очередь тесты проверяют, что внесенные изменения не ломают того, что уже было написано. Также они проверяют, что вновь внесенный код и сам работает корректно. Это очень сильное упрощение трактовки и понимания, но по своей сути тесты решают именно эти задачи, попутно помогая разработчикам понимать происходящее в коде быстрее и проще. При этом надо понимать, что тесты должны быть использованы по назначению, а не строго по паттерну и если «в книжке написано», что проверять надо обязательно все и на всех уровнях, то не факт, что это эффективно в конечном итоге.

Какие бывают тесты?

Вариантов представления процесса обеспечения качества довольно много. Если подходить совсем с академической точки зрения, то нужно вводить понятия валидации и верификации, которые являются базовыми и строить теорию далее, но чаще всего теория очень далека от практики в применении тестов, потому что динамика изменения подходов и внесения фич сильно выше, чем это могло бы быть в идеальном мире. Поэтому мы остановимся на самых полезных и применимых в реальности вещах.

Виды тестинга, про которые мы поговорим

- Модульное тестирование
- Интеграционное тестирование
- End-to-end(e2e) тестирование
- Нагрузочное тестирование
- Скриншотное тестирование

Конкретные либы для тестов

jest

selenuim

cypress

playwright

puppeteer

react-test-renderer

Fuzzer

Это подход, про который хочется сказать просто фоном, потому что он не самый частый в своем применении, но очень интересный.

https://habr.com/ru/company/dsec/blog/517596/

Как строить тесты?

В построении тестов мы должны проверить:

- instance
- validation
- corner-cases
- conditional cases
- main-stream

Тестируемый объект

```
function extractFirstAndLastSymbols(string) {
    if (!string) {
        return undefined
    }

    if (!(string instanceof String)) {
        return undefined
    }

    if (string.length === 1) {
        return string[0]
    }

    return string[0] + string[string.length - 1]
}
```

Instance

```
describe( name: 'extractFirstSymbol',  fn: () => {
    test( name: 'should be defined as a function',  fn: () => {
        expect(extractFirstSymbol).toBeInstanceOf(Function)
    })
})
```

Validation

```
describe( name: 'extractFirstSymbol',  fn: () => {
    test( name: 'should be defined as a function',  fn: () => {
        expect(extractFirstSymbol).toBeInstanceOf(Function)
    })

    test( name: 'should return undefined when it received undefined',  fn: () => {
        expect(extractFirstSymbol()).toBeUndefined()
    })

    test( name: 'should return undefined when it received non-string',  fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })
})
```

Corner-cases

```
describe( name: 'extractFirstSymbol', fn: () => {
    test( name: 'should be defined as a function', fn: () => {
        expect(extractFirstSymbol).toBeInstanceOf(Function)
    })

    test( name: 'should return undefined when it received undefined', fn: () => {
        expect(extractFirstSymbol()).toBeUndefined()
    })

    test( name: 'should return undefined when it received non-string', fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return undefined when it received an empty string', fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return one symbol only when it received string with length === 1', fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })
})
```

Conditional cases

```
/**
 * Extractor
 * @param string
 * @returns {string|undefined}
 * @see returns 'A' when 'a' is the first symbol
 */
function extractFirstAndLastSymbols(string) {
    if (!string) {
        return undefined
    }

    if (!(string instanceof String)) {
        return undefined
    }

    if (string.length === 1) {
        return string[0]
    }

    if (string.length[0] === 'a') {
        return 'A' + string[string.length - 1]
    }

    return string[0] + string[string.length - 1]
}
```

```
describe( name: 'extractFirstSymbol', fn: () => {
    test( name: 'should be defined as a function', fn: () => {
        expect(extractFirstSymbol).toBeInstanceOf(Function)
    })

    test( name: 'should return undefined when it received undefined', fn: () => {
        expect(extractFirstSymbol()).toBeUndefined()
    })

    test( name: 'should return undefined when it received non-string', fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return undefined when it received an empty string', fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return one symbol only when it received string with length === 1', fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return \'A\' at first when it received string starts with a', fn: () => {
        expect(extractFirstSymbol( string: 'ab')).toBe( expected: 'Ab')
    })
})
```

Main stream

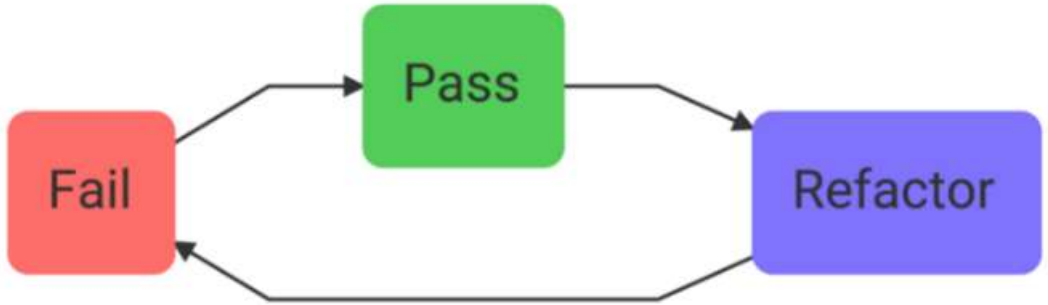
```
describe( name: 'extractFirstSymbol',  fn: () => {
    test( name: 'should be defined as a function',  fn: () => {
        expect(extractFirstSymbol).toBeInstanceOf(Function)
    })

    test( name: 'should return undefined when it received undefined',  fn: () => {
        expect(extractFirstSymbol()).toBeUndefined()
    })

    test( name: 'should return undefined when it received non-string',  fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return undefined when it received an empty string',  fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return one symbol only when it received string with length === 1',  fn: () => {
        expect(extractFirstSymbol( string: null)).toBeUndefined()
    })

    test( name: 'should return \'A\' at first when it received string starts with a',  fn: () => {
        expect(extractFirstSymbol( string: 'ab')).toBe( expected: 'Ab')
    })

    test( name: 'should return rt when it received react',  fn: () => {
        expect(extractFirstSymbol( string: 'react')).toBe( expected: 'rt')
    })
})
```

Базовые принципы TDD

Вот основные принципы применения TDD:

1. Прежде чем писать код реализации некоей возможности, пишут тест, который позволяет проверить, работает ли этот будущий код реализации, или нет. Прежде чем переходить к следующему шагу, тест запускают и убеждаются в том, что он выдаёт ошибку. Благодаря этому можно быть уверенным в том, что тест не выдаёт ложноположительные результаты, это — своего рода тестирование самих тестов.
2. Создают реализацию возможности и добиваются того, чтобы она успешно прошла тестирование.
3. Выполняют, если это нужно, рефакторинг кода. Рефакторинг, при наличии теста, который способен указать разработчику на правильность или неправильность работы системы, вселяет в разработчика уверенность в его действиях.

TDD расшифровывается как Test Driven Development (разработка через тестирование). Процесс, реализуемый в ходе применения этой методологии очень прост:



Тесты выявляют ошибки, тесты завершаются успешно, выполняется рефакторинг

https://habr.com/ru/company/ruvds/blog/450316/

Моки

Здесь важно сказать про удобство использования тестов, про настройки и все, что связано с процессов прогона тестов. Здесь можно удобно строить тестовые кейсы, подставлять моканые данные, апи, фабрики и многое другое. Моки – идеальное место для DI.

Приемы и примеры:

Пример про суб-тесты:

```
it.each(timeDiffSuits)( name: 'should return false when it received date more than 5 min ago',  fn: (suit : number ) => {
    const msecAgoCount: number = timeLimit + suit
    currentDate.setTime(Date.now() - msecAgoCount)
    expect(isCommentEditable(currentDate.toISOString())).toBe( expected: false)
})
```

Пример про дату:

```
let dateNowSpy: jest.SpyInstance

beforeAll( fn: () => {
    // Lock Time
    dateNowSpy = jest.spyOn(Date,  method: 'now').mockImplementation( fn: () => 10 ** 10)
})

afterAll( fn: () => {
    // Unlock Time
    dateNowSpy.mockRestore()
})
```

Окружение

```
"globals": {
    "ts-jest": {
        "diagnostics": false
    },
    "$env": {
        "IS_BROWSER": false,
        "IS_SERVER": true,
        "IS_DEVELOPMENT": true,
        "IS_PRODUCTION": false
    }
},
"testEnvironment": "jsdom",
```

Документация

Важный поинт про тесты – это тот факт, что тесты по своей сути документирую наш код и помогают читать и понимать о чем здесь суть и при каких входных данных какое поведение мы ожидаем.

Lint

Про линтеры нет так много можно сказать. Стоит отметить только, что линтеры сильно упрощают нам жизнь и помогают строго следовать договоренностям.

Самая распространенная либа – eslint. Для правок – prettier.

Build

Сборка - это отдельный этап разработки, в процессе которого производится создание конечного набора исходников, которые будут использоваться при непосредственном обеспечении жизненного цикла продукта.

Выделим несколько моментов, которые важно затронуть:

- Bundle
- Инструменты для js
- Кэширование и хэширование
- CDN

Bundle

Бандл – минифицированная и обобщенная сущность в исходниках, которая композирует в себя весь необходимый функционал. Важно сказать о важности оптимизации размера бандла через минимайзеры, а также централизация и продумываение решений, которые обеспечат дедупликацию внешних пакетов при сборке большого продукта из множества составляющих.

Инструменты

Основным инструментом для сборки в экосистеме JS является webpack.

Из менее удобных и менее актуальных – gulp.

Вспомогательный инструмент – babel.

Кэширование и хэширование

https://webpack.js.org/guides/caching/

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
|- /node_modules
```

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');

  module.exports = {
    entry: './src/index.js',
    plugins: [
      new HtmlWebpackPlugin({
-       title: 'Output Management',
+       title: 'Caching',
      }),
    ],
    output: {
-     filename: 'bundle.js',
+     filename: '[name].[contenthash].js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
  };
```

Running our build script, `npm run build`, with this configuration should produce the following output:

```
...
                    Asset       Size  Chunks                    Chunk Names
main.7e2c49a622975ebd9b7e.js     544 kB       0  [emitted]  [big]  main
                 index.html  197 bytes          [emitted]
...
```

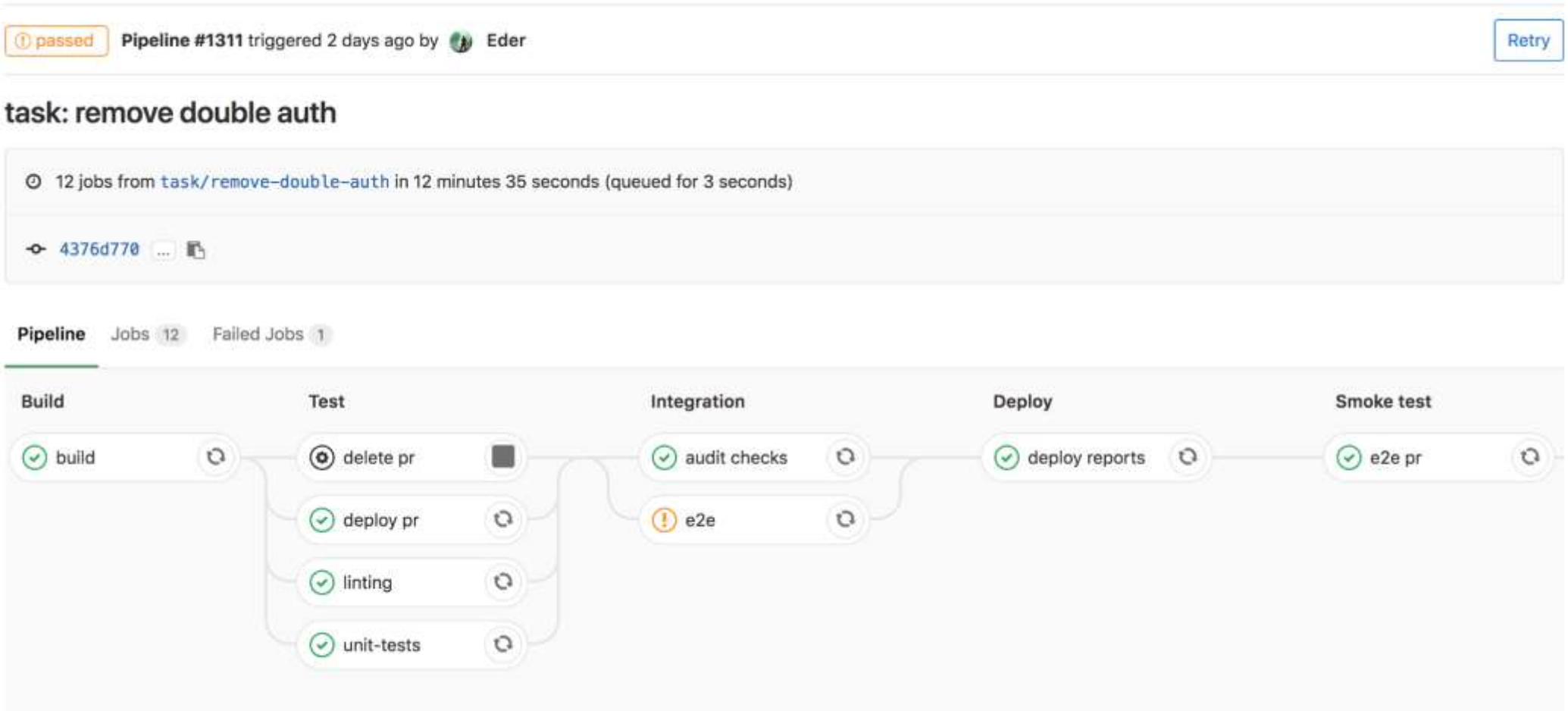
CDN

Очень удобно выкладывать на CDN сборки и далее использовать ссылки на них. Необходимо помнить об этом при разработке и о том, что данное решение достаточно дешевое.

Typing & validation

При использовании типизированного JS, то есть TS можно добавлять еще проверки зависимостей и типизации.

Pipeline


passed
Pipeline #1311 triggered 2 days ago by Eder
Retry
task: remove double auth
12 jobs from task/remove-double-auth in 12 minutes 35 seconds (queued for 3 seconds)
4376d770
Pipeline
Jobs 12
Failed Jobs 1
Build
Test
Integration
Deploy
Smoke test
build
delete pr
deploy pr
linting
unit-tests
audit checks
e2e
deploy reports
e2e pr