По степени изолированности тестируемых частей кода функциональные тесты принято делить на:

* Модульные
* Интеграционные
* Системные

Самые простые и наименее эффективные тесты — модульные. Они проверяют работоспособность конкретных программных модулей, например, функций. Такие тесты проще всего писать, но они не способны помочь проверить, что модули работают вместе. За это уже отвечают интеграционные тесты. У программистов на такие тесты должно быть направлено основное внимание. Их всё ещё достаточно просто писать (если экосистема проекта подготовлена к этому), и они охватывают гораздо большую часть системы.

Наиболее сложные и максимально эффективные — системные тесты. В случае сайтов — это тестирование через браузер. Эти тесты эмулируют поведение настоящего пользователя, ходят по сайту, кликают на ссылки, заполняют и отправляют формы. Сложность этих тестов в том, что им приходится опираться на вёрстку сайта, которая часто и непредсказуемо меняется. Кроме того, в браузере невозможно точно определить, когда закончилось одно действие и началось другое (с точки зрения программы). Именно поэтому такие тесты часто ломаются, их сложно писать и поддерживать.

**Автоматические тесты**

Всё, что требуется от автоматических тестов — повторить проверки, которые мы выполняли, делая ручное тестирование. Для этого достаточно старого доброго if и исключений.

Пример теста:

**if** (capitalize('hello') **!==** 'Hello') { *// Если результат функции не равен ожидаемому значению*

*// Выбрасываем исключение и завершаем выполнение теста*

**throw** **new** Error('Функция работает неверно!');

}

Из примера выше видно, что тесты — это точно такой же код, как и любой другой. Он работает в том же окружении и подчиняется тем же правилам, например, стандартам кодирования. А ещё он может содержать ошибки. Но это не значит, что надо писать тесты на тесты. Избежать всех ошибок невозможно, да и не нужно, иначе стоимость разработки стала бы неоправданно высокой. Обнаруженные ошибки в тестах исправляются, и жизнь продолжается дальше ;)

Структура этой директории зависит от того, на базе чего пишутся тесты, например, на базе какого фреймворка. В простых случаях, она отражает структуру исходного кода. Если предположить, что наша функция capitalize(text) определена в файле src/capitalize.js, то её тест лучше поместить в файл tests/capitalize.test.js. Слово test в имени модуля с тестами, используется только для более явного обозначения цели файла.

Для capitalize() пограничным случаем будет пустая строка:

**if** (capitalize('') **!==** '') {

**throw** **new** Error('Функция работает неверно!');

}

Добавив тест на пустую строку, мы увидим, что вызов показанной в начале урока функции capitalize() завершается с ошибкой. Внутри неё идёт обращение к первому индексу строки без проверки его существования. Исправленная версия кода:

**const** capitalize **=** (text) **=>** {

**if** (text **===** '') {

**return** '';

}

**const** firstChar **=** text[0].toUpperCase();

**const** restSubstring **=** text.slice(1);

**return** `${firstChar}${restSubstring}`;

};

Комбинация всех возможных вариантов поведения функции называется цикломатической сложностью. Это число показывает все возможные пути кода внутри функции. Цикломатическая сложность — хороший ориентир для понимания того, сколько и какие тесты нужно написать.

# Утверждения (Asserts)

Можно заметить, что все проверки строятся одинаковым способом: условие => исключение. Node.js поставляется с модулем *assert*, в котором есть несколько функций, упрощающих написание утверждений:

*// Такой необычный импорт связан с тем,*

*// что assert, экспортируемый по умолчанию, считается устаревшим*

*// Правильно использовать strict*

**import** { strict **as** assert } **from** 'assert';

**import** capitalize **from** '../src/capitalize.js';

*// Проверка сменилась с отрицательной на положительную*

assert(capitalize('') **===** '');

assert(capitalize('hello') **===** 'Hello');

В самом простом случае *assert* используется как функция, которая проверяет истинность переданного значения. Другими словами, assert(true) означает, что всё хорошо, а assert(false) говорит об ошибке. Последний вариант выбрасывает исключение с таким сообщением:

AssertionError [ERR\_ASSERTION]: false == true

Расшифровка сообщения: "Ожидалось, что значением выражения будет истина, но оказалось, что это ложь". Кроме сообщения, выводится бектрейс, по которому можно найти сработавшее утверждение:

// В данном случае assert сработал на 15 строчке файла capitalize.js

at first **(**file:///src/capitalize.js:15:19**)**

at default **(**file:///src/capitalize.js:11:3**)**

at file:///test.js:5:13

С другой стороны, вывод сообщения об ошибке крайне неинформативный. Единственный способ понять, что произошло — открывать код с упавшим утверждением (ещё есть вариант передать сообщение об ошибке последним параметром, но так не делают, потому что это слишком "ручной" способ, требующий больших усилий). Это пытаются исправить с помощью специализированных утверждений, заточенных под конкретные ситуации. Например, при сравнении двух значений подходит функция assert.strictEqual(actual, expected). Перепишем код выше:

**import** { strict **as** assert } **from** 'assert';

*// при использовании strict-режима*

*// проверка equal равносильна strictEqual*

**import** capitalize **from** '../src/capitalize.js';

*// Проверка сменилась с отрицательной на положительную*

assert.equal(capitalize(''), '');

*// Первый параметр actual – то, что пришло*

*// Второй параметр expected – то, что ожидает тест*

*// Правильный порядок аргументов имеет большое значение при анализе ошибки*

assert.equal(capitalize('hello'), 'Hello');

Для сравнения по значению используется ещё одно утверждение: assert.deepEqual(actual, expected). Оно опирается только на содержимое:

assert.deepEqual({}, {}); *// всё ок*

assert.deepEqual({ key: 'value' }, { key: 'value' }); *// всё ок*

assert.deepEqual({ key: 'value' }, { key: 'another value' }); *// Boom!*

На самом деле правила проверки этой функции достаточно сложны. Подробнее об этом можно прочитать в [документации](https://nodejs.org/api/assert.html#assert_assert_deepstrictequal_actual_expected_message)

Для тестирования негативных сценариев предназначены функции assert.notStrictEqual(actual, expected) и assert.notDeepStrictEqual(actual, expected). Они тестируют то, что значения не равны. Эти утверждения используются крайне редко, но знать о них всё равно полезно:

assert.notDeepEqual({ a: 1 }, { a: '1' }); *// OK!*

# power-assert

В выводе видно само утверждение и результат проверки. Но непонятно, что за объект user и какая у него структура. Для получения этой информации придётся включаться в отладку. Но вместо этого можно воспользоваться библиотекой *power-assert*:

**import** assert **from** 'power-assert';

*// Весь код остаётся тем же самым*

**const** user **=** {

name: 'Madonna',

friends: ['Kate', 'Michel'],

email: 'madonna@example.com',

};

*// Интерфейс библиотеки power-assert на 100% совместим со встроенным модулем assert.*

assert(user.name **===** 'Michel');

И посмотреть вывод:

AssertionError **[**ERR\_ASSERTION]: *# test.js:10*

assert**(**user.name **===** 'Michel'**)**

| | |

| | false

| "Madonna"

Object**{**name:"Madonna",friends:#Array#,email:"madonna@example.com"**}**

--- **[**string] 'Michel'

+++ **[**string] user.name

@@ -1,6 +1,7 @@

M

-ichel

+adonna

Попробуйте остановиться и внимательно изучить этот вывод. Что здесь показано? *power-assert* облегчает отладку настолько, насколько это вообще физически возможно. Он показывает значение каждого объекта и результат каждой операции, входящей в выражение, переданное в функцию *assert*. Кроме того, в конце он сравнивает строки и говорит, в чём конкретно было различие между ними.

Вот ещё один интересный пример из документации:

**import** assert **from** 'power-assert';

**const** array **=** [1, 2, 3];

**const** zero **=** 0;

**const** two **=** 2;

assert(array.indexOf(zero) **===** two);

*// AssertionError [ERR\_ASSERTION]: # test.js:7*

*//*

*// assert(array.indexOf(zero) === two)*

*// | | | | |*

*// | | | | 2*

*// | -1 0 false*

*// [1,2,3]*

*//*

*// [number] two*

*// => 2*

*// [number] array.indexOf(zero)*

*// => -1*

# Jest

Напишем наш первый тест. Создайте файл **\_\_tests\_\_/index.test.js** со следующим содержимым:

**import** reverse **from** '../src/index.js';

test('reverse', () **=>** {

expect(reverse('hello')).toEqual('olleh');

expect(reverse('')).toEqual('');

});

Далее мы разберём структуру этого файла, а пока попробуем запустить тест на выполнение:

*# Jest поддерживает ECMAScript модули в экспериментальном режиме*

NODE\_OPTIONS**=**--experimental-vm-modules npx jest

PASS \_\_tests\_\_/index.test.js

✓ reverse **(**11ms**)**

Test Suites: 1 passed, 1 total

Tests: 1 passed, 1 total

Snapshots: 0 total

Time: 1.166s

Ran all test suites.

**Структура**

Давайте ещё раз посмотрим на файл с тестом:

**import** reverse **from** '../src/index.js';

test('reverse', () **=>** {

expect(reverse('hello')).toEqual('olleh');

expect(reverse('')).toEqual('');

});

Для тестов Jest предоставляет две глобальные функции: test и expect. Они доступны без какого-либо импорта, так как Jest делает их глобальными функциями.

Функция test нужна для описания конкретного теста и его проверок. Самих тестовых функций может быть любое количество. Первым параметром эта функция принимает произвольную строчку, которая должна описывать сам тест. Эта строчка потом отображается на экране во время запуска тестов для упрощения отладки.

NODE\_OPTIONS**=**--experimental-vm-modules npx jest

PASS \_\_tests\_\_/index.test.js

✓ reverse **(**11ms**)** *# название теста*

Второй параметр — функция, внутри которой описан проверочный код. Обратите внимание на то, что этот код не выполняется сразу. Функция test добавляет его внутрь Jest, который уже решает, как и когда запускать тесты. Это позволяет проводить различные оптимизации, например, выполнять тесты параллельно.

Самое необычное в этом коде — проверки. Jest использует "матчеры" (matchers). Это утверждения, имеющие особую структуру, напоминающую обращение к объекту. Общий принцип работы матчеров такой:

* Вызывается функция expect(), куда передаётся актуальное (фактическое) значение.
* На результате, возвращаемом функцией expect(), вызывается подходящий матчер, например, toEqual.

Код с матчерами похож на обычные предложения на английском языке. Это сделано специально, чтобы их могли читать даже не программисты:

*// Ожидается, что результат выражения reverse('hello') равен 'olleh'*

expect(reverse('hello')).toEqual('olleh');

# Матчеры (Expectations)

*// Проверка равенства по ссылке*

*// assert.equal([1, 2], [1, 2])*

expect([1, 2]).toBe([1, 2]); *// false*

*// Проверка равенства по значению*

*// assert.deepEqual([1, 2], [1, 2])*

expect([1, 2]).toEqual([1, 2]); *// true*

Предположим, что функция возвращает массив и мы хотим проверить его размер. Для этого можно воспользоваться матчером toBe:

**const** data **=** [1, 2, 3];

*// take берет первые n элементов*

*// assert.equal(take(data, 2).length, 2)*

expect(take(data, 1).length).toBe(2);

Этот матчер прекрасно справится с задачей. Но в случае ошибки его вывод не слишком информативен:

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

Expected: 2

Received: 1

Поэтому лучше взять специализированный матчер для проверки размера массива:

expect(take(data, 1)).toHaveLength(2);

Тогда вывод расскажет гораздо больше:

expect**(**received**)**.toHaveLength**(**expected**)**

Expected length: 2

Received length: 1

Received array: **[**1]

Благодаря тому, что в expect передаётся сам массив, а не его длина, у Jest появляется возможность выводить содержимое массива в случае ошибки. Это, опять же, упрощает отладку.

Ниже пример некоторых популярных матчеров, полезных в ежедневном тестировании:

expect(**null**).toBeNull();

*// Проверяет значение на truthy (любое значение, которое приводится к true)*

expect(**true**).toBeTruthy();

*// Точное сравнение с true*

expect(**true**).toBe(**true**);

expect(**undefined**).toBeUndefined();

*// Проверка, что массив содержит элемент*

expect([1, 2, 3]).toContain(2);

*// Проверка, что строка содержит подстроку*

expect('hello, world').toMatch('hello');

*// Проверяет, что в объекте есть свойство с определённым значением*

expect({ key: 'value' }).toHaveProperty('key', 'value');

Кроме того, к любому матчеру можно применить модификатор not, который инвертирует поведение матчера:

expect(**null**).not.toBeNull(); *// not null*

expect(**undefined**).not.toBeUndefined(); *// not undefined*

expect([1, 2, 3]).not.toContain(2); *// not contain 2*

expect('hello, world').not.toMatch('hello'); *// not match hello*

Особняком стоит матчер toMatchObject. Он используется, когда нас в тестах интересует не весь объект, а только какая-то его часть:

**const** user **=** {

firstName: 'tolya',

lastName: 'petrov',

age: '33',

};

*// Тест пройдёт успешно, так как проверяется только firstName*

expect(user).toMatchObject({ firstName: 'tolya' });

## Тестируем основную функциональность

**import** makeStack **from** '../src/stack.js';

**const** stack **=** makeStack();

stack.isEmpty(); *// true*

stack.push(1); *// (1)*

stack.push(2); *// (1, 2)*

stack.push(3); *// (1, 2, 3)*

stack.isEmpty(); *// false*

stack.pop(); *// 3. В стеке (1, 2)*

stack.pop(); *// 2. В стеке (1)*

stack.pop(); *// 1. В стеке пусто*

stack.isEmpty(); *// true*

Теперь напишем первый тест. Первый тест всегда должен проверять позитивный сценарий — тот, в котором задействована основная функциональность тестируемого компонента:

**import** makeStack **from** '../src/stack.js';

test("stack's main flow", () **=>** {

**const** stack **=** makeStack();

*// Добавляем два элемента в стек и затем извлекаем их*

stack.push('one');

stack.push('two');

expect(stack.pop()).toEqual('two');

expect(stack.pop()).toEqual('one');

});

Этот тест проверяет, что правильно работают два основных метода без учёта пограничных случаев. Для этого внутри теста выполняются два матчера, которые по очереди проверяют извлекаемые значения из стека.

В интернете можно встретить мнение, что несколько проверок в рамках одного теста это неправильно. Что тесты нужно детализировать максимально подробно и создавать новый тест на каждую проверку.

test("stack's main flow", () **=>** {

**const** stack **=** makeStack();

stack.push('one');

stack.push('two');

expect(stack.pop()).toEqual('two');

});

test("stack's main flow", () **=>** {

**const** stack **=** makeStack();

stack.push('one');

stack.push('two');

stack.pop();

expect(stack.pop()).toEqual('one');

});

Такой подход нередко приводит к серьёзному раздуванию кода и дублированию. А выгода не очевидна. Что по-настоящему надо выделять в отдельный тест, так это другой сценарий, которому нужны другие данные и который выполняет другую последовательность действий.

**let** now;

*// Запускается перед каждым тестом*

beforeEach(() **=>** {

now **=** Date.now(); *// текущий timestamp*

});

test('first example', () **=>** console.log(now));

test('second example', () **=>** console.log(now));

*// console.log \_\_tests\_\_/index.test.js:9*

*// 1583871515943*

*//*

*// console.log \_\_tests\_\_/index.test.js:10*

*// 1583871515950*

beforeEach(callback) принимает на вход функцию, внутри которой выполняется инициализирующее действие. Оно не обязательно приводит к созданию переменных. Возможно, инициализация заключается в подготовке файловой системы, например, создании файлов. Даже если нам нужно выполнить код один раз перед всеми тестами, его все равно нужно выполнять не на уровне модуля, а внутри хука beforeAll(callback). Этот хук запускается ровно один раз перед всеми тестами, расположенными в одном модуле.

## Глубокая вложенность

Jest позволяет группировать тесты в блоки describe:

describe('User', () **=>** {

test('should be valid', () **=>** { */\* ... \*/* });

});

Они помогают структурировать сложные тесты и задать для каждого блока describe свой собственный beforeEach. Хотя такая возможность бывает полезна, но очень легко начать использовать её во вред:

describe('', () **=>** {

describe('...', () **=>** {

describe('...', () **=>** {

test('should be valid', () **=>** { */\* ... \*/* })

});

});

});

Глубокая иерархия тестов очень тяжело поддаётся анализу и фиксирует структуру. Из-за этого возникают сложности при добавлении новых проверок. Становится непонятно, к чему она относится. Это проблема любых иерархий, которые рассматривают систему только с одной точки зрения.