

[🏠](#) → [Язык JavaScript](#) → [Качество кода](#)

 7 июня 2022 г.

Автоматическое тестирование с использованием фреймворка Mocha

Далее у нас будут задачи, для проверки которых используется автоматическое тестирование. Также его часто применяют в реальных проектах.

Зачем нам нужны тесты?

Обычно, когда мы пишем функцию, мы легко можем представить, что она должна делать, и как она будет вести себя в зависимости от переданных параметров.

Во время разработки мы можем проверить правильность работы функции, просто вызвав её, например, из консоли и сравнив полученный результат с ожидаемым.

Если функция работает не так, как мы ожидаем, то можно внести исправления в код и запустить её ещё раз. Так можно повторять до тех пор, пока функция не станет работать так, как нам нужно.

Однако, такие «ручные перезапуски» – не лучшее решение.

При тестировании кода ручными перезапусками легко упустить что-нибудь важное.

Например, мы работаем над функцией `f`. Написали часть кода и решили протестировать. Выясняется, что `f(1)` работает правильно, в то время как `f(2)` – нет. Мы вносим в код исправления, и теперь `f(2)` работает правильно. Вроде бы, всё хорошо, не так ли? Однако, мы забыли заново протестировать `f(1)`. Возможно, после внесения правок `f(1)` стала работать неправильно.

Это – типичная ситуация. Во время разработки мы учитываем множество различных сценариев использования. Но сложно ожидать, что программист станет вручную проверять каждый из них после любого изменения кода. Поэтому легко исправить что-то одно и при этом сломать что-то другое.

Автоматическое тестирование означает, что тесты пишутся отдельно, в дополнение к коду. Они по-разному запускают наши функции и сравнивают результат с ожидаемым.

Behavior Driven Development (BDD)

Давайте начнём с техники под названием [Behavior Driven Development](#) или, коротко, BDD.

BDD – это три в одном: и тесты, и документация, и примеры использования.

Чтобы понять BDD – рассмотрим практический пример разработки.

Разработка функции возведения в степень — «pow»: спецификация

Допустим, мы хотим написать функцию `pow(x, n)`, которая возводит `x` в целочисленную степень `n`. Мы предполагаем, что `n ≥ 0`.

Эта задача взята в качестве примера. В JavaScript есть оператор `**`, который служит для возведения в степень. Мы сосредоточимся на процессе разработки, который также можно применять и для более сложных задач.

Перед тем, как начать писать код функции `pow`, мы можем представить себе, что она должна делать, и описать её.

Такое описание называется *спецификацией* (specification). Она содержит описания различных способов использования и тесты для них, например:

```
1 describe("pow", function() {
2
3   it("возводит в степень n", function() {
4     assert.equal(pow(2, 3), 8);
5   });
6
7 });
```

Спецификация состоит из трёх основных блоков:

`describe("заголовок", function() { ... })`

Какую функциональность мы описываем. В нашем случае мы описываем функцию `pow`. Используется для группировки рабочих лошадок – блоков `it`.

`it("описание", function() { ... })`

В первом аргументе блока `it` мы *человеческим языком* описываем конкретный способ использования функции, а во втором – пишем функцию, которая тестирует данный случай.

`assert.equal(value1, value2)`

Код внутри блока `it`, если функция работает верно, должен выполняться без ошибок.

Функции вида `assert.*` используются для проверки того, что функция `pow` работает так, как мы ожидаем. В этом примере мы используем одну из них – `assert.equal`, которая сравнивает переданные значения и выбрасывает ошибку, если они не равны друг другу. Существуют и другие типы сравнений и проверок, которые мы добавим позже.

Спецификация может быть запущена, и при этом будет выполнена проверка, указанная в блоке `it`, мы увидим это позднее.

Процесс разработки

Процесс разработки обычно выглядит следующим образом:

1. Пишется начальная спецификация с тестами, проверяющими основную функциональность.
2. Создаётся начальная реализация.
3. Для запуска тестов мы используем фреймворк [Mocha](#) (подробнее о нём чуть позже). Пока функция не готова, будут ошибки. Вносим изменения до тех пор, пока всё не начнёт работать так, как нам нужно.
4. Теперь у нас есть правильно работающая начальная реализация и тесты.
5. Мы добавляем новые способы использования в спецификацию, возможно, ещё не реализованные в тестируемом коде. Тесты начинают «падать» (выдавать ошибки).
6. Возвращаемся на шаг 3, дописываем реализацию до тех пор, пока тесты не начнут завершаться без ошибок.
7. Повторяем шаги 3-6, пока требуемая функциональность не будет готова.

Таким образом, разработка проходит *итеративно*. Мы пишем спецификацию, реализуем её, проверяем, что тесты выполняются без ошибок, пишем ещё тесты, снова проверяем, что они проходят и т.д.

Давайте посмотрим этот поток разработки на нашем примере.

Первый шаг уже завершён. У нас есть спецификация для функции `pow`. Теперь, перед тем, как писать реализацию, давайте подключим библиотеки для пробного запуска тестов, просто чтобы убедиться, что тесты работают (разумеется, они завершатся ошибками).

Спецификация в действии

В этой главе мы будем пользоваться следующими JavaScript-библиотеками для тестов:

- **Mocha** – основной фреймворк. Он предоставляет общие функции тестирования, такие как `describe` и `it`, а также функцию запуска тестов.
- **Chai** – библиотека, предоставляющая множество функций проверки утверждений. Пока мы будем использовать только `assert.equal`.
- **Sinon** – библиотека, позволяющая наблюдать за функциями, эмулировать встроенные функции и многое другое. Нам она пригодится позднее.

Эти библиотеки подходят как для тестирования внутри браузера, так и на стороне сервера. Мы рассмотрим вариант с браузером.

Полная HTML-страница с этими библиотеками и спецификацией функции `pow`:

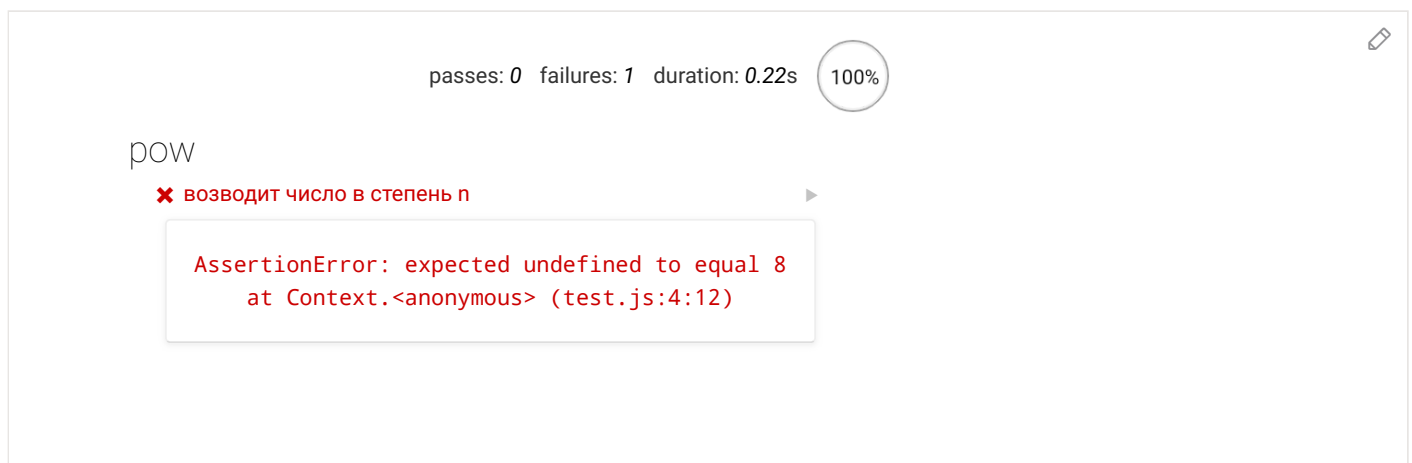
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <!-- добавим стили mocha для отображения результатов -->
5   <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2
6   <!-- добавляем сам фреймворк mocha -->
7   <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></s
8   <script>
9     // включаем режим тестирования в стиле BDD
10    mocha.setup('bdd');
11  </script>
12  <!-- добавим chai -->
13  <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></scr
14  <script>
15    // chai предоставляет большое количество функций. Объявим assert глобально
16    let assert = chai.assert;
17  </script>
18 </head>
19
20 <body>
21
22   <script>
23     function pow(x, n) {
24       /* Здесь будет реализация функции, пока пусто */
25     }
26   </script>
27
28   <!-- скрипт со спецификацией (describe, it...) -->
29   <script src="test.js"></script>
30
31   <!-- элемент с id="mocha" будет содержать результаты тестов -->
32   <div id="mocha"></div>
33
34   <!-- запускаем тесты! -->
35   <script>
36     mocha.run();
```

```
37     </script>
38 </body>
39
40 </html>
```

Условно страницу можно разделить на пять частей:

1. Тег `<head>` содержит сторонние библиотеки и стили для тестов.
2. Тег `<script>` содержит тестируемую функцию, в нашем случае – `pow`.
3. Тесты – в нашем случае внешний скрипт `test.js`, который содержит спецификацию `describe("pow", ...)`, представленную выше.
4. HTML-элемент `<div id="mocha">` будет использован фреймворком Mocha для вывода результатов тестирования.
5. Запуск тестов производится командой `mocha.run()`.

Результаты:



The screenshot shows the Mocha test runner interface. At the top, it displays the test statistics: "passes: 0 failures: 1 duration: 0.22s" and a progress indicator showing "100%". Below this, the function name "pow" is listed. A red error message is shown: "✖ возводит число в степень n". A tooltip box contains the full error message: "AssertionError: expected undefined to equal 8 at Context.<anonymous> (test.js:4:12)".

Пока что тест завершается ошибкой. Это логично, потому что у нас пустая функция `pow`, так что `pow(2,3)` возвращает `undefined` вместо `8`.

На будущее отметим, что существуют более высокоуровневые фреймворки для тестирования, такие как [karma](#) и другие. С их помощью легко сделать автозапуск множества тестов.

Начальная реализация

Давайте напишем простую реализацию функции `pow`, чтобы пройти тесты.

```
1 function pow(x, n) {
2   return 8; // :) сжульничаем!
3 }
```

Вау, теперь всё работает!

passes: 1 failures: 0 duration: 0.36s

100%

pow

✓ возводит в степень n



Улучшаем спецификацию

Конечно, мы сжульничали. Функция не работает. Попытка посчитать `pow(3, 3)` даст некорректный результат, однако тесты проходят.

...Такая ситуация вполне типична, она случается на практике. Тесты проходят, но функция работает неправильно. Наша спецификация не идеальна. Нужно дополнить её тестами.

Давайте добавим ещё один тест, чтобы посмотреть, что `pow(3, 3) = 27`.

У нас есть два пути организации тестов:

1. Первый – добавить ещё один `assert` в существующий `it`:

```
1 describe("pow", function() {
2
3   it("возводит число в степень n", function() {
4     assert.equal(pow(2, 3), 8);
5     assert.equal(pow(3, 3), 27);
6   });
7
8 });
```

2. Второй – написать два теста:

```
1 describe("pow", function() {
2
3   it("2 в степени 3 будет 8", function() {
4     assert.equal(pow(2, 3), 8);
5   });
6
7   it("3 в степени 3 будет 27", function() {
8     assert.equal(pow(3, 3), 27);
9   });
10
11 });
```

Принципиальная разница в том, что когда один из `assert` выбрасывает ошибку, то выполнение `it` блока тут же прекращается. Таким образом, если первый `assert` выбросит ошибку, результат работы второго `assert` мы уже не узнаем.

Разделять тесты предпочтительнее, так как мы получаем больше информации о том, что конкретно пошло не так.

Помимо этого есть одно хорошее правило, которому стоит следовать.

Один тест проверяет одну вещь.

Если вы посмотрите на тест и увидите в нём две независимые проверки, то такой тест лучше разделить на два более простых.

Давайте продолжим со вторым вариантом.

Результаты:

passes: 1 failures: 1 duration: 0.28s 100%

pow

✓ 2 в степени 3 будет 8

✗ 3 в степени 3 будет 27

AssertionError: expected 8 to equal 27
at Context.<anonymous> (test.js:8:12)

Как мы и ожидали, второй тест провалился. Естественно, наша функция всегда возвращает `8`, в то время как `assert` ожидает `27`.

Улучшаем реализацию

Давайте напишем что-то более похожее на функцию возведения в степень, чтобы заставить тесты проходить.

```
1 function pow(x, n) {
2   let result = 1;
3
4   for (let i = 0; i < n; i++) {
5     result *= x;
6   }
7
8   return result;
9 }
```

Чтобы убедиться, что эта реализация работает нормально, давайте протестируем её на большем количестве значений. Чтобы не писать вручную каждый блок `it`, мы можем генерировать их в цикле `for`:

```
1 describe("pow", function() {
2
3   function makeTest(x) {
4     let expected = x * x * x;
5     it(`${x} в степени 3 будет ${expected}`, function() {
6       assert.equal(pow(x, 3), expected);
7     });
8   }
9
10  for (let x = 1; x <= 5; x++) {
11    makeTest(x);
12  }
```

```
12   }  
13  
14   });
```

Результат:

passes: 5 failures: 0 duration: 0.18s 100%

pow

✓ 1 в степени 3 будет 1

✓ 2 в степени 3 будет 8

✓ 3 в степени 3 будет 27

✓ 4 в степени 3 будет 64

✓ 5 в степени 3 будет 125

Вложенные блоки describe

Мы собираемся добавить больше тестов. Однако, перед этим стоит сгруппировать вспомогательную функцию `makeTest` и цикл `for`. Нам не нужна функция `makeTest` в других тестах, она нужна только в цикле `for`. Её предназначение – проверить, что `pow` правильно возводит число в заданную степень.

Группировка производится вложенными блоками `describe`:

```
1 describe("pow", function() {  
2  
3   describe("возводит x в степень 3", function() {  
4  
5     function makeTest(x) {  
6       let expected = x * x * x;  
7       it(`${x} в степени 3 будет ${expected}`, function() {  
8         assert.equal(pow(x, 3), expected);  
9       });  
10    }  
11  
12    for (let x = 1; x <= 5; x++) {  
13      makeTest(x);  
14    }  
15  
16  });  
17  
18  // ... другие тесты. Можно писать и describe, и it блоки.  
19 });
```

Вложенные `describe` образуют новую подгруппу тестов. В результатах мы можем видеть дополнительные отступы в названиях.

passes: 5 failures: 0 duration: 0.25s

100%



pow

возводит x в степень 3

- ✓ 1 в степени 3 будет 1
- ✓ 2 в степени 3 будет 8
- ✓ 3 в степени 3 будет 27
- ✓ 4 в степени 3 будет 64
- ✓ 5 в степени 3 будет 125



В будущем мы можем написать новые `it` и `describe` блоки на верхнем уровне со своими собственными вспомогательными функциями. Им не будет доступна функция `makeTest` из примера выше.

i before/after и beforeEach/afterEach

Мы можем задать `before/after` функции, которые будут выполняться до/после тестов, а также функции `beforeEach/afterEach`, выполняемые до/после каждого `it`.

Например:

```
1 describe("тест", function() {
2
3   before(() => alert("Тестирование началось – перед тестами"));
4   after(() => alert("Тестирование закончилось – после всех тестов"));
5
6   beforeEach(() => alert("Перед тестом – начинаем выполнять тест"));
7   afterEach(() => alert("После теста – заканчиваем выполнение теста"));
8
9   it('тест 1', () => alert(1));
10  it('тест 2', () => alert(2));
11
12 });
```

Порядок выполнения будет таким:

- 1 Тестирование началось – перед тестами (before)
- 2 Перед тестом – начинаем выполнять тест (beforeEach)
- 3 1
- 4 После теста – заканчиваем выполнение теста (afterEach)
- 5 Перед тестом – начинаем выполнять тест (beforeEach)
- 6 2
- 7 После теста – заканчиваем выполнение теста (afterEach)
- 8 Тестирование закончилось – после всех тестов (after)

[Открыть пример в песочнице.](#)

Обычно `beforeEach/afterEach` и `before/after` используются для инициализации, обнуления счётчиков или чего-нибудь ещё между тестами (или группами тестов).

Расширение спецификации

Основная функциональность `pow` реализована. Первая итерация разработки завершена. Когда мы закончим отмечать и пить шампанское, давайте продолжим работу и улучшим `pow`.

Как было сказано, функция `pow(x, n)` предназначена для работы с целыми положительными значениями `n`.

Для обозначения математических ошибок функции JavaScript обычно возвращают `NaN`. Давайте делать также для некорректных значений `n`.

Сначала давайте опишем это поведение в спецификации.

```
1 describe("pow", function() {
2
3   // ...
4
5   it("для отрицательных n возвращает NaN", function() {
6     assert.isNaN(pow(2, -1));
7   });
8
9   it("для дробных n возвращает NaN", function() {
10    assert.isNaN(pow(2, 1.5));
11  });
12
13 });
```

Результаты с новыми тестами:

passes: 5 failures: 2 duration: 0.31s 100%

pow

✗ для отрицательных n возвращает NaN ▶

AssertionError: expected 1 to be NaN
at assert.isNaN (https://cdnjs.cloudflare.c
at Context.<anonymous> (test.js:19:12)

✗ для дробных n возвращает NaN ▶

AssertionError: expected 4 to be NaN
at assert.isNaN (https://cdnjs.cloudflare.c
at Context.<anonymous> (test.js:23:12)

ВОЗВОДИТ x в степень 3

✓ 1 в степени 3 будет 1 ▶

✓ 2 в степени 3 будет 8 ▶

✓ 3 в степени 3 будет 27 ▶

✓ 4 в степени 3 будет 64 ▶

✓ 5 в степени 3 будет 125 ▶

Новые тесты падают, потому что наша реализация не поддерживает их. Так работает BDD. Сначала мы добавляем тесты, которые падают, а уже потом пишем под них реализацию.

Другие функции сравнения

Обратите внимание на `assert.isNaN`. Это проверка того, что переданное значение равно `NaN`.

Библиотека [Chai](#) содержит множество других подобных функций, например:

- `assert.equal(value1, value2)` – проверяет равенство `value1 == value2`.
- `assert.strictEqual(value1, value2)` – проверяет строгое равенство `value1 === value2`.
- `assert.notEqual`, `assert.notStrictEqual` – проверяет неравенство и строгое неравенство соответственно.
- `assert.isTrue(value)` – проверяет, что `value === true`
- `assert.isFalse(value)` – проверяет, что `value === false`
- ...с полным списком можно ознакомиться в [документации](#)

Итак, нам нужно добавить пару строчек в функцию `pow`:

```
1 function pow(x, n) {  
2   if (n < 0) return NaN;  
3   if (Math.round(n) !== n) return NaN;  
4  
5   let result = 1;  
6  
7   for (let i = 0; i < n; i++) {  
8     result *= x;  
9   }  
10  
11   return result;  
12 }
```

Теперь работает, все тесты проходят:

passes: 7 failures: 0 duration: 0.35s

100%

pow

✓ если n - отрицательное число, результат будет NaN ▶

✓ если n не число, результат будет NaN ▶

возводит x в степень 3

✓ 1 в степени 3 будет 1 ▶

✓ 2 в степени 3 будет 8 ▶

✓ 3 в степени 3 будет 27 ▶

✓ 4 в степени 3 будет 64 ▶

✓ 5 в степени 3 будет 125 ▶

[Открыть готовый пример в песочнице.](#)

Итого

В BDD сначала пишут спецификацию, а потом реализацию. В конце у нас есть и то, и другое.

Спецификацию можно использовать тремя способами:

1. Как **Тесты** – они гарантируют, что функция работает правильно.
2. Как **Документацию** – заголовки блоков `describe` и `it` описывают поведение функции.
3. Как **Примеры** – тесты, по сути, являются готовыми примерами использования функции.

Имея спецификацию, мы можем улучшить, изменить и даже переписать функцию с нуля, и при этом мы будем уверены, что она продолжает работать правильно.

Это особенно важно в больших проектах, когда одна функция может быть использована во множестве мест. Когда мы вносим в такую функцию изменения, у нас нет никакой возможности вручную проверить, что она продолжает работать правильно во всех местах, где её используют.

Не имея тестов, людям приходится выбирать один из двух путей:

1. Внести изменения, и неважно, что будет. Потом у наших пользователей станут проявляться ошибки, ведь мы наверняка что-то забудем проверить вручную.
2. Или же, если наказание за ошибки в коде серьёзное, то люди просто побоятся вносить изменения в такие функции. Код будет стареть, «зарастать паутиной», и никто не захочет в него лезть. Это нехорошо для разработки.

Автоматическое тестирование кода позволяет избежать этих проблем!

Если проект покрыт тестами, то вышеупомянутые проблемы не возникают. После любых изменений мы можем запустить тесты и увидеть результаты огромного количества проверок, сделанных за секунды.

Кроме того, код, хорошо покрытый тестами, как правило, имеет лучшую архитектуру.

Это естественно, ведь такой код легче менять и улучшать. Но не только по этой причине.

Для написания тестов нужно организовать код таким образом, чтобы у каждой функции была ясно поставленная задача и точно определены её аргументы и возвращаемое значение. А это означает, что мы получаем хорошую архитектуру с самого начала.

В реальности это не всегда так просто. Иногда сложно написать спецификацию до того, как будет написана реализация, потому что не всегда чётко понятно, как та или иная функция должна себя вести. Но в общем и целом написание тестов делает разработку быстрее, а итоговый продукт более стабильным.

Далее по книге мы встретим много задач с тестами, так что вы увидите много практических примеров.

Написание тестов требует хорошего знания JavaScript. Но мы только начали учить его. Не волнуйтесь. Пока вам не нужно писать тесты, но вы уже умеете их читать и поймёте даже более сложные примеры, чем те, что были представлены в этой главе.

✓ Задачи

Что не так с этим тестом?

важность: 5

Что не так в нижеприведённом тесте функции `pow` ?

```
1 it("Возводит x в степень n", function() {
2   let x = 5;
3
4   let result = x;
5   assert.equal(pow(x, 1), result);
6
7   result *= x;
8   assert.equal(pow(x, 2), result);
```

```
9
10   result *= x;
11   assert.equal(pow(x, 3), result);
12 });
```

P.S. Тест не содержит синтаксических ошибок и успешно проходит.

решение



Тест демонстрирует один из соблазнов, с которым сталкиваются разработчики при их написании.

У нас тут, по сути, три теста, но они написаны как одна функция с тремя проверками.

Иногда так проще писать, но если произойдёт ошибка, то гораздо сложнее понять, что пошло не так.

Если ошибка происходит посередине сложного потока выполнения, то нам придётся выяснять, какие данные были в этом месте. По сути, придётся *отлаживать тест*.

Гораздо лучше разбить тест на несколько блоков `it` и ясно описать входные и ожидаемые на выходе данные.

Примерно так:

```
1  describe("Возводит x в степень n", function() {
2    it("5 в степени 1 будет 5", function() {
3      assert.equal(pow(5, 1), 5);
4    });
5
6    it("5 в степени 2 будет 25", function() {
7      assert.equal(pow(5, 2), 25);
8    });
9
10   it("5 в степени 3 будет 125", function() {
11     assert.equal(pow(5, 3), 125);
12   });
13 });
```

Мы заменили один `it` на `describe` и группу блоков `it`. Теперь, если какой-либо из блоков завершится неудачно, мы точно увидим, с какими данными это произошло.

Также мы можем изолировать один тест и запускать только его, написав `it.only` вместо `it`:

```
1  describe("Возводит x в степень n", function() {
2    it("5 в степени 1 будет 5", function() {
3      assert.equal(pow(5, 1), 5);
4    });
5
6    // Mocha будет запускать только этот блок
7    it.only("5 в степени 2 будет 25", function() {
8      assert.equal(pow(5, 2), 25);
9    });
10
11   it("5 в степени 3 будет 125", function() {
12     assert.equal(pow(5, 3), 125);
```

```
13     });  
14 });
```

[Предыдущий урок](#)[Следующий урок](#)

Поделиться

[Карта учебника](#)

Проводим курсы по JavaScript и фреймворкам.



💬 Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите [на GitHub](#).
- Для одной строки кода используйте тег `<code>` , для нескольких строк кода — тег `<pre>` , если больше 10 строк — ссылку на песочницу ([plnkr](#), [JSBin](#), [codepen...](#))
- Если что-то непонятно в статье — пишите, что именно и с какого места.