Автоматические тесты при помощи chai и mocha

Зачем нужны тесты?

- При написании функции мы обычно представляем, что она должна делать, какое значение — на каких аргументах выдавать.
- В процессе разработки мы, время от времени, проверяем, правильно ли работает функция. Самый простой способ проверить — это запустить её, например, в консоли, и посмотреть результат.
- Если что-то не так — поправить, опять запустить — посмотреть результат… И так — «до победного конца».
- Но такие ручные запуски — очень несовершенное средство проверки.
- **Когда проверяешь работу кода вручную — легко его «недотестировать».**
- Например, пишем функцию f. Написали, тестируем с разными аргументами. Вызов функции f(a) — работает, а вот f(b) — не работает. Поправили код — стало работать f(b), вроде закончили. Но при этом забыли заново протестировать f(a) — упс, вот и возможная ошибка в коде.
- **Автоматизированное тестирование — это когда тесты написаны отдельно от кода, и можно в любой момент запустить их и проверить все важные случаи использования.**

BDD — поведенческие тесты кода

- Мы рассмотрим методику тестирования, которая входит в BDD — Behavior Driven Development. Подход BDD давно и с успехом используется во многих проектах.
- BDD — это не просто тесты. Это гораздо больше.
- **Тесты BDD — это три в одном: И тесты И документация И примеры использования одновременно.**
- Впрочем, хватит слов. Рассмотрим примеры.

Разработка pow: спецификация

- Допустим, мы хотим разработать функцию pow(x, n), которая возводит x в целую степень n, для простоты n≥0.
- Ещё до разработки мы можем представить себе, что эта функция будет делать и описать это по методике BDD.
- Это описание называется *спецификация* (или, как говорят в обиходе, «спека») и выглядит так:

```
describe("pow", function() {

  it("возводит в n-ю степень", function() {
    assert.equal(pow(2, 3), 8);
  });

});
```

- У спецификации есть три основных строительных блока, которые вы видите в примере выше:
- **describe(название, function() { ... })**
- Задаёт, что именно мы описываем, используется для группировки «рабочих лошадок» — блоков it. В данном случае мы описываем функцию pow.
- **it(название, function() { ... })**
- В названии блока it *человеческим языком* описывается, что должна делать функция, далее следует *тест*, который проверяет это.
- **assert.equal(value1, value2)**
- Код внутри it, если реализация верна, должен выполняться без ошибок.
- Различные функции вида assert.* используются, чтобы проверить, делает ли pow то, что задумано. Пока что нас интересует только одна из них — assert.equal, она сравнивает свой первый аргумент со вторым и выдаёт ошибку в случае, когда они не равны. В данном случае она проверяет, что результат pow(2, 3) равен 8.

Поток разработки

- Как правило, поток разработки таков:
- Пишется спецификация, которая описывает самый базовый функционал.
- Делается начальная реализация.
- Для проверки соответствия спецификации мы задействуем одновременно фреймворк, в нашем случае Mocha вместе со спецификацией и реализацией. Фреймворк запускает все тесты it и выводит ошибки, если они возникнут. При ошибках вносятся исправления.
- Спецификация расширяется, в неё добавляются возможности, которые пока, возможно, не поддерживаются реализацией.
- Идём на пункт 2, делаем реализацию, и так далее, до победного конца.
- Разработка ведётся *итеративно*, один проход за другим, пока спецификация и реализация не будут завершены.
- В нашем случае первый шаг уже завершён, начальная спецификация готова, хорошо бы приступить к реализации. Но перед этим проведём «нулевой» запуск спецификации, просто чтобы увидеть, что уже в таком виде, даже без реализации — тесты работают.

Пример в действии

- Для запуска тестов нужны соответствующие JavaScript-библиотеки.
- Мы будем использовать:
- Mocha — эта библиотека содержит общие функции для тестирования, включая describe и it.
- Chai — библиотека поддерживает разнообразные функции для проверок. Есть разные «стили» проверки результатов, с которыми мы познакомимся позже, на текущий момент мы будем использовать лишьassert.equal.
- Sinon — для эмуляции и хитрой подмены функций «заглушками», понадобится позднее.
- Эти библиотеки позволяют тестировать JS не только в браузере, но и на сервере Node.JS. Здесь мы рассмотрим браузерный вариант, серверный использует те же функции.
- Пример HTML-страницы для тестов:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <!-- подключаем стили Mocha, для отображения результатов -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/2.1.0/mocha.css">
  <!-- подключаем библиотеку Mocha -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/2.1.0/mocha.js"></script>
  <!-- настраиваем Mocha: предстоит BDD-тестирование -->
  <script>
    mocha.setup('bdd');
  </script>
  <!-- подключаем chai -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/2.0.0/chai.js"></script>
  <!-- в chai есть много всего, выносим assert в глобальную область -->
  <script>
    var assert = chai.assert;
  </script>
</head>
```

```
<body>
  <script>
    function pow(x, n) {
      /* код функции, пока что пусто */
    }
  </script>
  <!-- в этом скрипте находятся спеки -->
  <script src="test.js"></script>
  <!-- в элементе с id="mocha" будут результаты тестов -->
  <div id="mocha"></div>
  <!-- запустить тесты! -->
  <script>
    mocha.run();
  </script>
</body>
</html>
```

- Эту страницу можно условно разделить на четыре части:
- Блок <head> — в нём мы подключаем библиотеки и стили для тестирования, нашего кода там нет.
- Блок <script> с реализацией спецификации, в нашем случае — с кодом для pow.
- Далее подключаются тесты, файл test.js содержит describe("pow", ...), который был описан выше. Методыdescribe и it принадлежат библиотеке Mocha, так что важно, что она была подключена выше.
- Элемент будет использоваться библиотекой Mocha для вывода результатов. Запуск тестов инициируется командой mocha.run().
- Результат срабатывания:
- Пока что тесты не проходят, но это логично — вместо функции стоит «заглушка», пустой код.
- Пока что у нас одна функция и одна спецификация, но на будущее заметим, что если различных функций и тестов много — это не проблема, можно их все подключить на одной странице. Конфликта не будет, так как каждый функционал имеет свой блок describe("что тестируем"...). Сами же тесты обычно пишутся так, чтобы не влиять друг на друга, не делать лишних глобальных переменных.

Начальная реализация

- Пока что, как видно, тесты не проходят, ошибка сразу же. Давайте сделаем минимальную реализацию pow, которая бы работала нормально:
- function pow() {
- return 8; // :) мы - мошенники!
- }
- О, вот теперь работает:

Исправление спецификации

- Функция, конечно, ещё не готова, но тесты проходят. Это ненадолго :)
- Здесь мы видим ситуацию, которая (и не обязательно при ленивом программисте!) бывает на практике — да, есть тесты, они проходят, но увы, функция работает неправильно.
- **С точки зрения BDD, ошибка при проходящих тестах — вина спецификации.**
- В первую очередь не реализация исправляется, а уточняется спецификация, пишется (падающий) тест.
- Сейчас мы расширим спецификацию, добавив проверку на pow(3, 4) = 81.
- Здесь есть два варианта организации кода:
- Первый вариант — добавить assert в тот же it:

```
describe("pow", function() {
  it("возводит в n-ю степень", function() {
    assert.equal(pow(2, 3), 8);
    assert.equal(pow(3, 4), 81);
  });
});
```

Второй вариант — сделать два теста:

```
describe("pow", function() {
  it("при возведении 2 в 3ю степень результат 8", function() {
    assert.equal(pow(2, 3), 8);
  });
  it("при возведении 3 в 4ю степень равен 81", function() {
    assert.equal(pow(3, 4), 81);
  });
});
```

- Их принципиальное различие в том, что если assert обнаруживает ошибку, то он тут же прекращает выполнение блоки it. Поэтому в первом варианте, если вдруг первый assert «провалился», то про результат второго мы никогда не узнаем.
- **Таким образом, разделить эти тесты может быть полезно, чтобы мы получили больше информации о происходящем.**
- Кроме того, есть ещё одно правило, которое желательно соблюдать.
- **Один тест тестирует ровно одну вещь.**
- Если мы явно видим, что тест включает в себя совершенно независимые проверки — лучше разбить его на два более простых и наглядных.
- По этим причинам второй вариант здесь предпочтительнее.
- Результат:
- Как и следовало ожидать, второй тест не проходит. Ещё бы, ведь функция всё время возвращает 8.

Уточнение реализации

Придётся написать нечто более реальное, чтобы тесты проходили:

```
function pow(x, n) {
  var result = 1;

  for (var i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Чтобы быть уверенными, что функция работает верно, желательно протестировать её на большем количестве значений. Вместо того, чтобы писать блоки `it` вручную, мы можем сгенерировать тесты в цикле `for`:

```
describe("pow", function() {

  function makeTest(x) {
    var expected = x * x * x;
    it("при возведении " + x + " в степень 3 результат: " + expected, function()
{
      assert.equal(pow(x, 3), expected);
    });
  }

  for (var x = 1; x <= 5; x++) {
    makeTest(x);
  }

});
```

Вложенный describe

Функция `makeTest` и цикл `for`, очевидно, нужны друг другу, но не нужны для других тестов, которые мы добавим в дальнейшем. Они образуют единую группу, задача которой — проверить возведение в n-ю степень.

Будет правильно выделить их, при помощи вложенного блока `describe`:

```
describe("pow", function() {

  describe("возводит x в степень n", function() {

    function makeTest(x) {
      var expected = x * x * x;
      it("при возведении " + x + " в степень 3 результат: " + expected,
function() {
        assert.equal(pow(x, 3), expected);
      });
    }

    for (var x = 1; x <= 5; x++) {
      makeTest(x);
    }

  });

  // ... дальнейшие тесты it и подблоки describe ...
});
```

- Вложенный describe объявит новую «подгруппу» тестов, блоки it которой запускаются так же, как и обычно, но выводятся с подзаголовком, вот так:
- В будущем мы сможем в добавить другие тесты it и блоки describe со своими вспомогательными функциями

before/after и beforeEach/afterEach

В каждом блоке `describe` можно также задать функции `before/after`, которые будут выполнены до/после запуска тестов, а также `beforeEach/afterEach`, которые выполняются до/после каждого `it`.

Например:

```
describe("Тест", function() {

  before(function() { alert("Начало тестов"); });
  after(function() { alert("Конец тестов"); });

  beforeEach(function() { alert("Вход в тест"); });
  afterEach(function() { alert("Выход из теста"); });

  it('тест 1', function() { alert('1'); });
  it('тест 2', function() { alert('2'); });

});
```

Последовательность будет такой:

```
Начало тестов
Вход в тест
1
Выход из теста
Вход в тест
2
Выход из теста
Конец тестов
```

Открыть пример с тестами в песочнице

Как правило, `beforeEach/afterEach` (`before/each`) используют, если необходимо произвести инициализацию, обнулить счётчики или сделать что-то ещё в таком духе между тестами (или их группами).

Расширение спецификации

- Базовый функционал pow описан и реализован, первая итерация разработки завершена. Теперь расширим и уточним его.
- Как говорилось ранее, функция pow(x, n) предназначена для работы с целыми неотрицательными n.
- В JavaScript для ошибки вычислений служит специальное значение NaN, которое функция будет возвращать при некорректных n.
- Добавим это поведение в спецификацию

```
describe("pow", function() {

  // ...

  it("при возведении в отрицательную степень результат NaN", function() {
    assert(isNaN(pow(2, -1)));
  });

  it("при возведении в дробную степень результат NaN", function() {
    assert(isNaN(pow(2, 1.5)));
  });

});
```

Другие assert

- Обратим внимание, в спецификации выше использована проверка не assert.equal, как раньше, aassert(выражение). Такая проверка выдаёт ошибку, если значение выражения при приведении к логическому типу не true.
- Она потребовалась, потому что сравнивать с NaN обычным способом нельзя: NaN не равно никакому значению, даже самому себе, поэтому assert.equal(NaN, x) не подойдёт.
- Кстати, мы и ранее могли бы использовать assert(value1 == value2) вместо assert.equal(value1, value2). Оба этих assert проверяют одно и тоже.
- Однако, между этими вызовами есть отличие в деталях сообщения об ошибке.
- При «упавшем» assert в примере выше мы видим "Unspecified AssertionError", то есть просто «что-то пошло не так», а при assert.equal(value1, value2) — будут дополнительные подробности: expected 8 to equal 81.

- **Поэтому рекомендуется использовать именно ту проверку, которая максимально соответствует задаче.**
- Вот самые востребованные assert-проверки, встроенные в Chai:
- assert(value) — проверяет что value является true в логическом контексте.
- assert.equal(value1, value2) — проверяет равенство value1 == value2.
- assert.strictEqual(value1, value2) — проверяет строгое равенство value1 === value2.
- assert.notEqual, assert.notStrictEqual — проверки, обратные двум предыдущим.
- assert.isTrue(value) — проверяет, что value === true
- assert.isFalse(value) — проверяет, что value === false
- …более полный список — в документации
- В нашем случае хорошо бы иметь проверку assert.isNaN, но, увы, такого метода нет, поэтому приходится использовать самый общий assert(...). В этом случае для того, чтобы сделать сообщение об ошибке понятнее, желательно добавить к assert описание.

Все вызовы assert позволяют дополнительным последним аргументом указать строку с описанием ошибки, которое выводится, если assert не проходит.

Добавим описание ошибки в конец наших assert'ов:

```
describe("pow", function() {

  // ...

  it("при возведении в отрицательную степень результат NaN", function() {
    assert(isNaN(pow(2, -1)), "pow(2, -1) не NaN");
  });

  it("при возведении в дробную степень результат NaN", function() {
    assert(isNaN(pow(2, 1.5)), "pow(2, 1.5) не NaN");
  });

});
```

- Теперь результат теста гораздо яснее говорит о том, что не так
- В коде тестов выше можно было бы добавить описание и к assert.equal, указав в конце: assert.equal(value1, value2, "описание"), но с равенством обычно и так всё понятно, поэтому мы так делать не будем.

Итого

- Итак, разработка завершена, мы получили полноценную спецификацию и код, который её реализует.
- Задачи выше позволяют дополнить её, и в результате может получиться что-то в таком духе:

```
describe("pow", function() {

  describe("возводит x в степень n", function() {

    function makeTest(x) {
      var expected = x * x * x;
      it("при возведении " + x + " в степень 3 результат: " + expected,
function() {
        assert.equal(pow(x, 3), expected);
      });
    }

    for (var x = 1; x <= 5; x++) {
      makeTest(x);
    }

  });
```

```
  it("при возведении в отрицательную степень результат NaN", function() {
    assert(isNaN(pow(2, -1)), "pow(2, -1) не NaN");
  });

  it("при возведении в дробную степень результат NaN", function() {
    assert(isNaN(pow(2, 1.5)), "pow(2, -1.5) не NaN");
  });

  describe("любое число, кроме нуля, в степени 0 равно 1", function() {

    function makeTest(x) {
      it("при возведении " + x + " в степень 0 результат: 1", function() {
        assert.equal(pow(x, 0), 1);
      });
    }

    for (var x = -5; x <= 5; x += 2) {
      makeTest(x);
    }

  });

  it("ноль в нулевой степени даёт NaN", function() {
    assert(isNaN(pow(0, 0)), "0 в степени 0 не NaN");
  });

});
```

- Эту спецификацию можно использовать как:
- **Тесты**, которые гарантируют правильность работы кода.
- **Документацию** по функции, что она конкретно делает.
- **Примеры** использования функции, которые демонстрируют её работу внутри it.
- Имея спецификацию, мы можем улучшать, менять, переписывать функцию и легко контролировать её работу, просматривая тесты.
- Особенно важно это в больших проектах.
- Бывает так, что изменение в одной части кода может повлечь за собой «падение» другой части, которая её использует. Так как всё-всё в большом проекте руками не перепроверишь, то такие ошибки имеют большой шанс остаться в продукте и вылезти позже, когда проект увидит посетитель или заказчик.
- Чтобы избежать таких проблем, бывает, что вообще стараются не трогать код, от которого много что зависит, даже если его ну очень нужно переписать. Жизнь пробивается тонкими росточками там, где должен цвести и пахнуть новый функционал.

- **Код, покрытый автотестами, являет собой полную противоположность этому!**
- Даже если какое-то изменение потенциально может порушить всё — его совершенно не страшно сделать. Ведь есть масса тестов, которые быстро и в автоматическом режиме проверят работу кода и, если что-то падает — это можно будет легко локализовать и поправить.
- **Кроме того, код, покрытый тестами, имеет лучшую архитектуру.**
- Конечно, это естественное следствие того, что его легче менять и улучшать. Но не только.
- Чтобы написать тесты, нужно разбить код на функции так, чтобы для каждой функции было чётко понятно, что она получает на вход, что делает с этим и что возвращает. Это означает ясную и понятную структуру с самого начала.
- Конечно, в реальной жизни всё не так просто. Зачастую написать тест сложно. Или сложно поддерживать тесты, поскольку код активно меняется. Сами тесты тоже пишутся по-разному, при помощи разных инструментов.