تست خودکار با Mocha

تست خودکار ، در توسعه کد استفاده شده و در پروژه‌ها به صورت گسترده از آن استفاده می‌شود.

چرا به تست نیاز داریم؟

وقتی فانکشنی می‌نویسیم معمولا می‌توانیم تصور می‌کنیم که چه کاری انجام خواهد داد و چه ورودی‌هایی منجر به چه خروجی‌هایی می‌شود.در طول پروژه‌ها پروژه می‌توانیم فانکشن‌ها را اجرا کنیم تا ببینیم آیا خروجی‌های مدنظر ما را تولید می‌کنند یا خیر. برای مثال می‌توانیم این کار را در کنسول نیز انجام دهیم. اگر خطایی رخ دهد ، آن را برطرف می‌کنیم و دوباره فانکشن را اجرا می‌کنیم و نتیجه را چک می‌کنیم. آنقدر این کار را انجام می‌دهیم تا دیگر خطایی پیدا نکنیم.

ولی اینگونه تست کردن‌های دستی ، خیلی جالب نیستند.

هنگامی که به صورت دستی کدی را تست می‌کنید ، به احتمال زیاد چیزی از قلم خواهد افتاد.

فرض کنید می‌خواهیم فانکشنی به نام f بسازیم. کد آن را نوشته سپس آن را تست می‌کنیم : f(1) کار می‌کند ولی f(2) به درستی کار نمی‌کند. کد را تصحیح کرده و اکنون f(2) نیز کار می‌کند. به نظر شما کامل شد؟ ولی یادمان رفت که دوباره f(1) را تست کنیم. در این گونه موارد امکان بروز خطا وجود دارد.

این اتفاق خیلی معمول است. هنگامی که کدی را توسعه می‌دهیم ، حالت‌های زیادی را در ذهن داریم ولی خیلی بعید است که برنامه نویس بعد از هر تغییر تمام آن حالات را چک کند. پس ممکن است به راحتی با تصحیح یک خطا ، دیگری به خطا برخورد کند.

تست خودکار به این معنی است که تست‌ها ، مجزا از کد اصلی نوشته می‌شوند. این تست‌ها ، کد ما را به روش‌های مختلف اجرا می‌کنند و نتیجه را با چیزی که انتظار می‌رود مقایسه می‌کنند.

توسعه رفتار محور – Behavior Driven Development (BDD)

با تکنیکی به نام BDD شروع می‌کنیم.

برای فهم بهتر BDD ، با نمونه‌های عملی ، آن را توضیح خواهیم داد.

فرض کنید میخواهیم فانکشن pow(x,n) را به شکلی بنویسیم که عدد صحیح x را به توان n برساند. فرض میکنیم $n \geq 0$.

این نمونه فقط یک مثال است. عملگر ** در جاوا اسکریپت وجود دارد که همین کار را انجام می‌دهد ولی در اینجا ، تمرکز ما بر روی جریان و فرآیند توسعه کد است که از این روش در نمونه‌های پیچیده‌تر نیز می‌توان استفاده کرد.

قبل از اینکه کد pow را بنویسیم ، می‌توانیم تصور کنیم که این کد چه کاری انجام خواهد داد و آن را شرح بدهیم.

این شرح ، specification یا به اختصار spec نامیده شده و شامل موارد استفاده همراه با تست آنها می‌شود. مثلا :

```
describe("pow", function() {
  it("raises to n-th power", function() {
    assert.equal(pow(2, 3), 8);
  });
});
```


هر spec سه بخش اصلی دارد که در مثال بالا آن را مشاهده می‌کنید :

: describe("title", function() { ... })

چه عملکردی را داریم شرح می‌دهیم. در مثال ما ، ما فانکشن pow را شرح می‌دهیم. از بلاک‌های it هم برای گروه‌بندی workerها استفاده می‌کنیم.

: it("use case description", function() { ... })

در عنوان بلاک it به زبان آدمیزاد توضیح می‌دهیم که این مورد استفاده خاص چیست و آرگومان دوم هم فانکشنی است که آن تست را انجام می‌دهد.

: assert.equal(value1, value2)

اگر کد پیاده‌سازی شده صحیح باشد ، کد داخل بلاک it باید بدون خطا اجرا شود.

فانکشن‌های assert.* برای این استفاده می‌شوند که چک کنیم pow آنطور که انتظار می‌رود کار می‌کند یا خیر. در این مثال از یکی از آنها استفاده کردیم : assert.equal . این متد ، آرگومان‌ها را مقایسه کرده و اگر برابر نباشند اعلام می‌کند که خطایی رخ داده است. در این مثال نتیجه pow(2,3) را با 8 مقایسه می‌کند. مقایسه‌های دیگری نیز وجود دارند که در ادامه خواهیم دید.

specها می‌توانند اجرا شوند و تست‌های داخل بلاک it را اجرا کنند. در ادامه به صورت عملی انجام خواهیم داد.

جریان توسعه

جریان توسعه معمولا به شکل زیر است :

۱– یک spec اولیه به همراه تست‌هایی برای عملکردهای ابتدایی آن نوشته می‌شود.

۲– پیاده‌سازی اولیه صورت می‌گیرد.

۳– برای چک کردن اینکه آیا کار می‌کند یا خیر ، از فریم ورکی به نام Mocha استفاده می‌کنیم که specها را اجرا می‌کنند. اگر تست‌ها درست اجرا نشوند ، خطا نمایش داده می‌شود. در این صورت کدها را اصلاح کرده تا بالاخره درست کار کنند.

۴– اکنون به یک پیاده‌سازی ابتدایی به همراه تست‌های آن رسیدیم.

۵– اکنون موارد جدیدی به spec اضافه می‌کنیم که احتمالا هنوز پیاده سازی نشده‌اند. به همین خاطر تست‌ها خطا می‌دهند.

۶– به مرحله ۳ برمی‌گردیم و پیاده‌سازی را تکمیل می‌کنیم تا تست‌ها به درستی اجرا شده و خطایی بروز ندهد.

۷– مراحل ۳ تا ۶ را آنقدر تکرار می‌کنیم تا فانکشن مورد نظر آماده شود.

پس توسعه یک فرآیند iterative است. یعنی spec را می‌نویسیم ، پیاده‌سازی می‌کنیم ، مطمن می‌شویم که تست‌ها را پاس می‌کند ، سپس تست‌های بیشتری می‌نویسیم و مطمن می‌شویم که آنها را نیز پاس می‌کند و همینطور الی آخر. در نهایت ما هم کدی صحیح و قابل اجرا و هم تست‌های آن را داریم.

بیایید این مراحل را در مثال عملی خودمان بررسی کنیم.

مرحله اول تقریبا انجام شد :spec ابتدایی برای pow را نوشتیم. حال قبل از اینکه پیاده‌سازی را انجام دهیم ، اجازه بدهید از تعدادی کتابخانه جاوااسکریپت استفاده کنیم تا ببینیم چیزی که نوشتیم کار می‌کند یا خطا می‌دهد. (خطا خواهند داد)

اجرای spec ها

در این آموزش از کتابخانه‌های جاوا اسکریپتی زیر برای تست استفاده می‌کنیم:

Mocha – به عنوان فریم ورک اصلی : این کتابخانه فانکشن‌های معمول تست مثل describe و it و فانکشن اصلی اجرای تست‌ها را داخل خودش دارد.

Chai : کتابخانه‌ای که assertهای زیادی داخل خودش دارد که به ما اجازه می‌دهد از آنها استفاده کنیم. در حال حاضر فقط به assert.equal نیاز داریم.

Sinon : کتابخانه ای برای تحت نظر قرار دادن فانکشن‌ها ، تقلید فانکشن‌های داخلی و غیره. در ادامه به آن نیاز خواهیم داشت.

از این کتابخانه‌ها هم در سمت مرورگر و هم در سمت سرور می‌توان استفاده نمود. در اینجا ما در سمت مرورگر از آن استفاده می‌کنیم.

سورس کامل یک صفحه وب به همراه این کتابخانه‌ها به شکل زیر است:

```
<!DOCTYPE html>
<html>
<head>
  <!-- add mocha css, to show results -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
  <!-- add mocha framework code -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></script>
  <script>
    mocha.setup('bdd'); // minimal setup
  </script>
  <!-- add chai -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></script>
  <script>
    // chai has a lot of stuff, let's make assert global
    let assert = chai.assert;
  </script>
</head>
<body>

  <script>
    function pow(x, n) {
      /* function code is to be written, empty now */
    }
  </script>
  <!-- the script with tests (describe, it...) -->
  <script src="test.js"></script>

  <!-- the element with id="mocha" will contain test results -->
  <div id="mocha"></div>
  <!-- run tests! -->
  <script>
    mocha.run();
  </script>
</body>
</html>
```

صفحه بالا را می‌توان به پنج بخش تقسیم کرد :

۱– <head> : اضافه کردن کتابخانه‌های ثانویه و استایل‌ها برای کارهای تستی

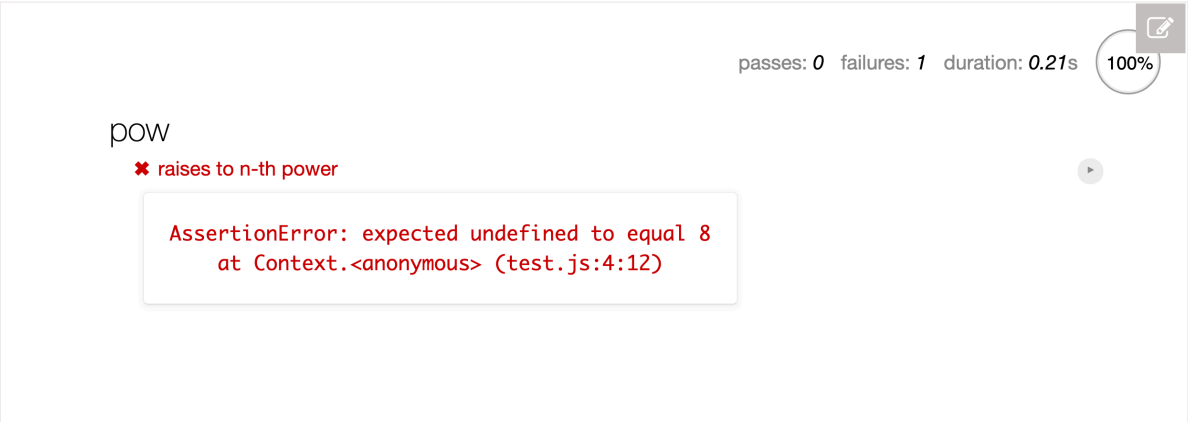
۲– <script> : حاوی فانکشنی که می‌خواهیم تست کنیم . در مثال ما : pow

۳– تست‌ها : در مثال‌ما اسکریپت خارجی به نام test.js که شامل describe("pow",...) توضیح داده شده از صفحات قبل است .

۴– المان که برای نمایش خروجی Mocha به کار می‌رود .

۵– تست‌ها بوسیله دستور mochar.run() اجرا می‌شوند .

نتیجه :



در حال حاضر تست fail می‌شود چرا که خطایی وجود دارد . فانکشن pow ما خالی بوده و هنوز چیزی داخلش ننوشتیم به همین خاطر به جای ۸ مقدار undefined را برمی‌گرداند .

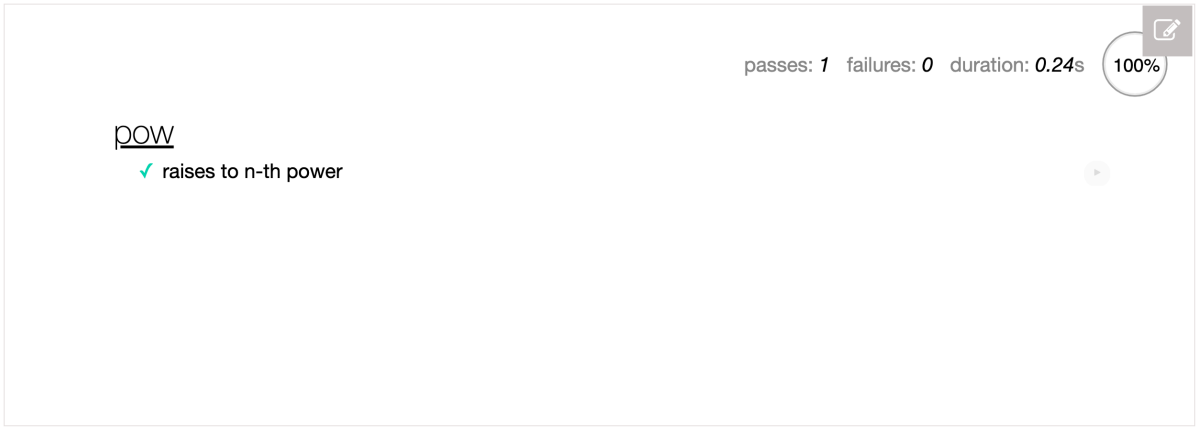
به عنوان یک نکته جانبی این موضوع را نیز در نظر داشته باشید که تسترهای سطح بالایی مثل karma وجود دارند که عملیات خودکار سازی تست‌ها را ساده‌تر می‌کنند .

پیاده‌سازی اولیه

بیایید با هم یک پیاده‌سازی اولیه از pow انجام دهیم که تست‌ها را pass می‌کند :

```
function pow(x, n) {
  return 8; // :) we cheat!
}
```

واو! تست ما کار می‌کند!



کاری که انجام دادیم یک تقلب بود. فانکشن درست کار نمی‌کند بلکه تحت هر شرایطی عدد ۸ را برمی‌گرداند که استثنا برای مثال pow(2,3) درست کار می‌کند ولی برای باقی محاسبات مثل pow(3,4) درست کار نخواهد کرد. ولی اگر بخواهیم روراست باشیم این موضوع بسیار معمول بوده و بسیار اتفاق می‌افتد که تست‌ها pass شوند ولی فانکشن درست کار نکند. spec ما ناقص است و باید use caseهای بیشتری به آن اضافه کنیم. بیایید تست دیگری اضافه کنیم که pow(3,4) = 81 را بررسی نماید.

به ۲ طریق می‌توانیم این کار را انجام دهیم:

۱- روش اول : assertهای بیشتری به it موجود اضافه کنیم :

```
describe("pow", function() {
  it("raises to n-th power", function() {
    assert.equal(pow(2, 3), 8);
    assert.equal(pow(3, 4), 81);
  });
});
```


۲- روش دوم : دو تا تست مجزا بسازیم :

```
describe("pow", function() {
  it("2 raised to power 3 is 8", function() {
    assert.equal(pow(2, 3), 8);
  });
  it("3 raised to power 4 is 81", function() {
    assert.equal(pow(3, 4), 81);
  });
});
```

تفاوت این دو روش در این است که هنگامی که assert با خطا مواجه شود ، بلاک it متوقف شده و تمام می‌شود. پس در روش اول اگر assert اول با خطا مواجه شود ، نتیجه assert بعدی را نخواهیم دید .

جداسازی تست‌ها مفیدتر است چرا که اطلاعات بیشتری از نحوه کار فانکشن در اختیار ما قرار می‌دهد پس روش دوم پیشنهاد می‌شود. در کنار همه اینها ، یک قانون دیگر وجود دارد که توصیه می‌شود رعایت کنید :

هر تست فقط برای بررسی یک چیز استفاده شود.

اگر دیدید تستی شامل دو چک کردن مستقل می‌باشد ، بهتر است آن را به دو تست ساده‌تر تبدیل کنید . پس بیایید با روش دوم ادامه دهیم.

نتیجه :

همانگونه که انتظار داشتیم تست ما با خطا مواجه شد چرا که فانکشن pow تحت هر شرایطی ۸ برمی‌گرداند در صورتی که در تست دوم باید ۸۱ بشود.

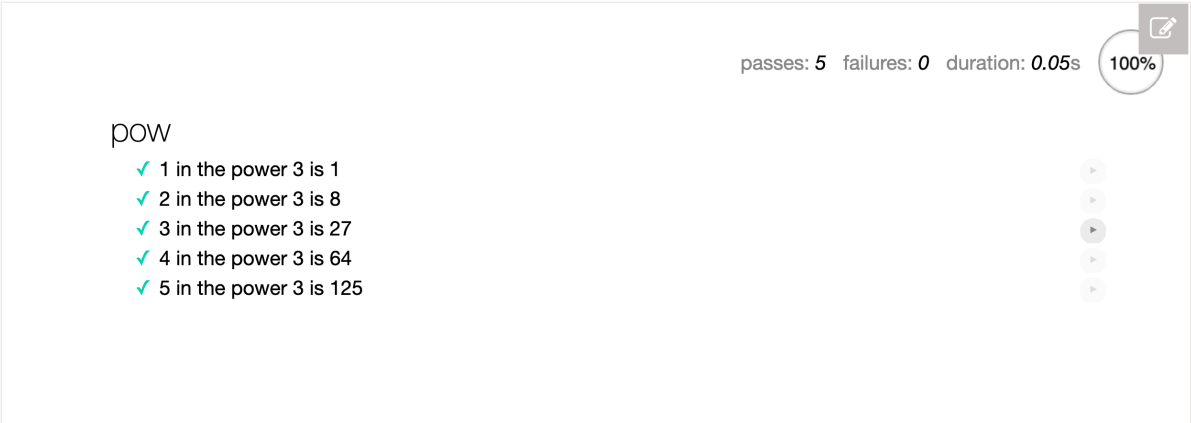
بهبود و تکمیل پیاده‌سازی فانکشن

حال فانکشن خود را به صورت واقعی می‌نویسیم :

```
function pow(x, n) {
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

برای اینکه مطمن بشویم فانکشن به صورت صحیح کار می‌کند بیایید تست‌های بیتشری انجام دهیم. به جای اینکه بلاک‌های it را به صورت دستی بنویسیم ، می‌توانیم آنها را داخل حلقه for تولید کنیم:

```
describe("pow", function() {
  function makeTest(x) {
    let expected = x * x * x;
    it(`${x} in the power 3 is ${expected}`, function() {
      assert.equal(pow(x, 3), expected);
    });
  }
  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }
});
```

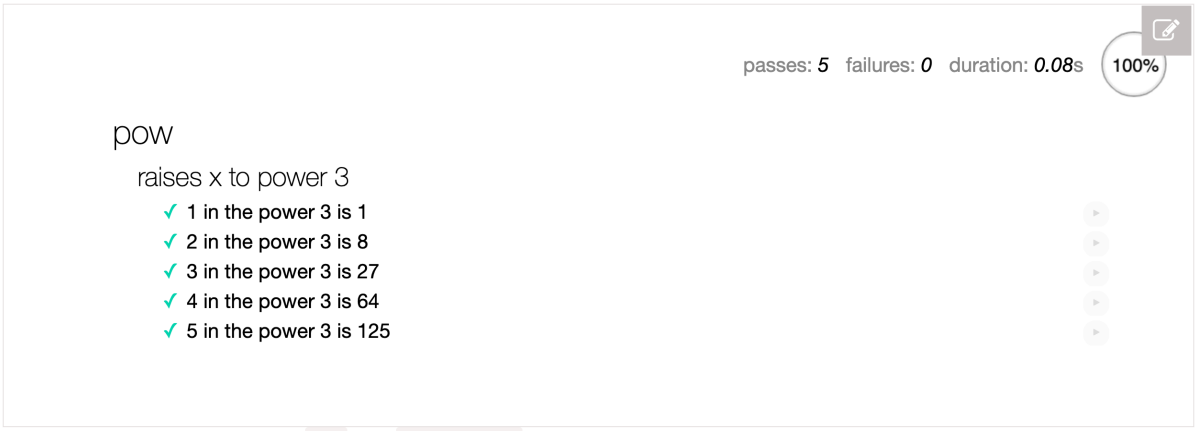
describeهای تودرتو

حال می‌خواهیم بازهم تست‌های بیشتری بنویسیم. ولی قبل از آن باید فانکشن کمکی makeTest و حلقه for را داخل یک گروه قرار دهیم. در تست‌های دیگر به makeTest نیاز نداریم و فقط در حلقه for از آن استفاده می‌کنیم. گروه بندی بوسیله یک describe درونی دیگر انجام می‌شود.

```
describe("pow", function() {
  describe("raises x to power 3", function() {
    function makeTest(x) {
      let expected = x * x * x;
      it(`${x} in the power 3 is ${expected}`, function() {
        assert.equal(pow(x, 3), expected);
```

```
    });
  }
  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }
 });
 // ... more tests to follow here, both describe and it can be added
});
```


describe درونی یک زیرگروه جدید از تست‌ها را تعریف می‌کند . در خروجی ، نتیجه تست درونی را با عنوان جداگانه‌ای می‌بینیم :



در آینده می‌توانیم بلاک‌های describe و it دیگری نیز به بالاترین سطح اضافه کنیم که فانکشن‌های کمکی خودشان را دارند و به makeTest دسترسی ندارند .

before/after و beforeEach/afterEach

می‌توانیم فانکشن‌های before/after تعریف کنیم تا قبل یا بعد از هر تست اجرا شوند ، همچنین می‌توانیم فانکشن‌های beforeEach/afterEach تعریف کنیم تا قبل یا بعد از هر بلاک it اجرا شوند .

```
describe("test", function() {
  before(() => alert("Testing started – before all tests"));
  after(() => alert("Testing finished – after all tests"));
  beforeEach(() => alert("Before a test – enter a test"));
  afterEach(() => alert("After a test – exit a test"));
  it('test 1', () => alert(1));
  it('test 2', () => alert(2));
});
```

خروجی :

```
Testing started – before all tests (before)
Before a test – enter a test (beforeEach)
1
After a test – exit a test   (afterEach)
Before a test – enter a test (beforeEach)
2
After a test – exit a test   (afterEach)
Testing finished – after all tests (after)
```


معمولا از این توابع برای مقدار دهی اولیه یا صفر سازی کانترها یا چیزهای دیگر بین تست‌ها استفاده می‌شود.

توسعه spec

عملکرد اصلی pow کامل و اولین مرحله توسعه انجام شد. حال به سراغ مراحل بعدی توسعه آن برویم.

همانطور که می‌دانیم فانکشن pow(x,n) باید با مقادیر صحیح و مثبت n کار کند. هنگام بروز خطا در محاسبات ریاضی ، جاوا اسکریپت معمولا NaN برمی‌گرداند. بیاید همین کار را برای مقادیر نادرست n نیز انجام دهیم.

ابتدا spec آن را توسعه بدهیم!

```
describe("pow", function() {
  // ...
  it("for negative n the result is NaN", function() {
    assert.isNaN(pow(2, -1));
  });
  it("for non-integer n the result is NaN", function() {
    assert.isNaN(pow(2, 1.5));
  });
});
```

نتیجه به صورت شکل صفحه بعد نمایش داده خواهد شد :

تست‌های جدید fail خواهند شد چرا که پیاده سازی فانکشن pow آنها را پوشش نمی‌دهد . BDD اینگونه کار می‌کند : ابتدا تست‌های fail را می‌نویسیم ، سپس فانکشن را طوری توسعه می‌دهیم که تست‌ها fail نشوند .

سایر assertionها

توجه داشته باشید که assert.isNaN ، NaN بودن را چک می‌کند .

assertionهای دیگری در Chai موجود است برای مثال :

assert.equal(value1, value2) – checks the equality value1 == value2.

assert.strictEqual(value1, value2) – checks the strict equality value1 === value2.

assert.notEqual, assert.notStrictEqual – inverse checks to the ones above.

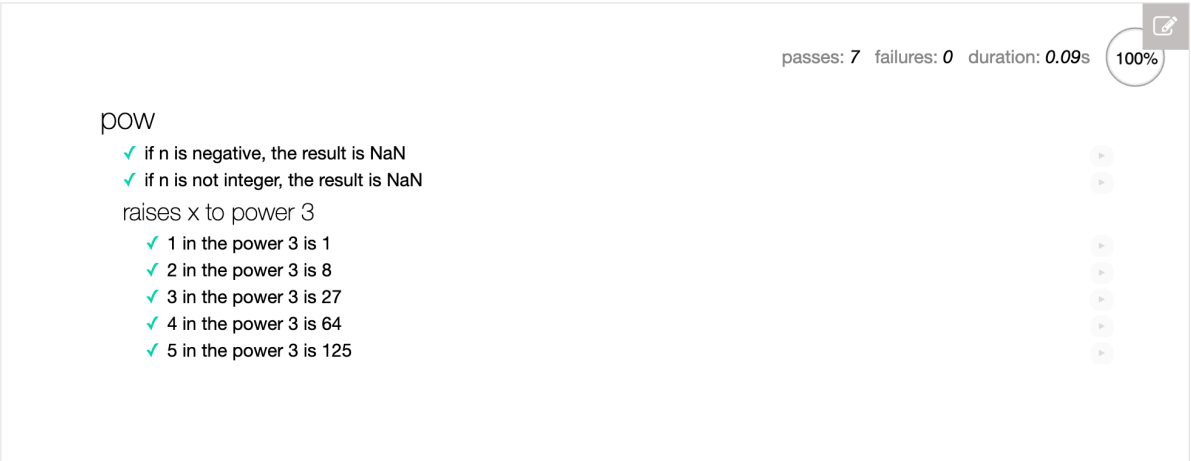
assert.isTrue(value) – checks that value === true

assert.isFalse(value) – checks that value === false

بنابراین ، برای تکمیل pow باید چند خط به آن اضافه کنیم :

```
function pow(x, n) {
  if (n < 0) return NaN;
  if (Math.round(n) != n) return NaN;
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```


حال تمامی تست‌ها pass می‌شود :



خلاصه :

در BDD ابتدا تست‌ها نوشته شده سپس پیاده سازی انجام می‌شود . در آخر هم spec و هم کد اصلی را داریم .

spec را به سه طریق می‌توان استفاده کرد :

۱– به عنوان Test : آنها تضمین می‌کنند که کد درست کار کند .

۲– به عنوان Doc : عناوین describe و it عملکرد فانکشن را شرح می‌دهند .

۳– به عنوان Example : تست‌ها ، نمونه‌های واقعی از نحوه استفاده از فانکشن ها هستند .

با specها به راحتی می‌توانیم کدها را ارتقا ، تغییر و یا حتی از صفر بنویسیم و مطمن باشیم که همچنان به صورت صحیح کار می‌کند . این موضوع مخصوصا در پروژه‌های بزرگ که فانکشن‌های نوشته شده در جاهای متفاوتی استفاده می‌شود کاربرد دارد . در چنین موقعیتی اگر تغییری روی کد اصلی انجام شود ، هیچ راهی وجود دارد ندارد تا مطمن شویم که کد همه جا درست کار می‌کند .

بدون تست‌ها ، دو راه پیش روی ما وجود دارد :

۱– بدون توجه به اثرات جانبی ، تغییرات را انجام دهیم و منتظر بشینیم تا باگ‌های متفاوتی بروز پیدا کند .

۲– اگر بروز خطا ، دردسر بزرگی به همراه دارد ، کدها را تغییر نداده و بروزرسانی نکنیم.

تست خودکار کمک می‌کند که این مشکلات برطرف شوند !

اگر پروژه همراه با تست انجام شود ، چنین مشکلاتی بروز نخواهد کرد. بعد از هر تغییر ، کافی است تست‌ها را اجرا کنیم تا حالات متفاوتی در کسری از ثانیه اجرا شود.

کدی که همراه با تست نوشته شود ، ساختار بهتری دارد.

به طور طبیعی ، کدی که همراه با تست خودکار نوشته شود را راحت‌تر می‌توان اصلاح کرد و ارتقا داد. ولی دلیل دیگری نیز وجود دارد. برای نوشتن تست‌ها ، کد باید به صورتی سازماندهی شود که وظیفه هر فانکشن ، ورودی و خروجی آن مشخص باشد و این به معنای ساختار و معماری خوب از ابتدای کار است.
در واقعیت ، تست نویسی کار ساده ای نمی‌باشد. گاهی اوقات نوشتن spec قبل از کد واقعی کار دشواری است چرا که هنوز رفتار دقیق فانکشن معلوم نیست. ولی به صورت کلی ، نوشتن تست ، سرعت توسعه را افزایش داده و کد نوشته شده را پایدارتر می‌کند.
در آموزش‌های بعدی ، taskهای مختلفی را همراه با تست آنها خواهید دید.
لازمه نوشتن تست‌ها ، دانش مناسبی از جاوا اسکریپت است. ولی ما هنوز در ابتدای راه هستیم. پس برای اینکه راحت‌تر پیش برویم ، فعلا نیازی ندارید که تست بنویسید ولی باید بتوانید آنها را بخوانید ، هر چند که ممکن است کمی پیچیده‌تر از مثال‌های این بخش باشند.