Unit testing with vanilla JavaScript: The very basics

 aurel kurtula Nov 25 '17 *Updated on Nov 28, 2017* • 9 min read

#javascript #testing

In my last tutorial I covered the basics of JavaScript testing, or rather I illustrated what it is and how it can be achieved. But JavaScript testing is better done with a framework. So in this tutorial I'm going to test a simple Todo app using Jasmine, "a behavior-driven development framework for testing JavaScript code".

I found it to be very easy when thinking that it simply exists to give structure and more robustness to our testing, especially when compared to the previous vanilla approach.

Setting up the project

We are going to build a basic todo app. It will have two components. One which will control the data and one which will inject the data to the DOM.

For the sake of simplicity we are not going to use any build tool. We'll just have four files:

- `index.html` - The actual app will be rendered and served to the client from here.
- `ToDo.js` - We'll write our application code here.

- `SpecRunner.html` - Test results are going to be displayed here.
- `ToDoSpec.js` - Using Jasmine we'll test the code we write in `ToDo.js` here.

For a bigger application we'd structure those files differently of course but for simplicity those are all in the root folder. Further, talking about CSS here would be overkill, but clearly you'd use css to style the code in the index page.

The `index.html` is going to be empty, everything is going to be injected via JavaScript.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Todo</title>
</head>
<body>
</body>
<script src="ToDo.js"></script>
</html>
```

The `SpecRunner.html` is like wise empty but we'll link to Jasmine files, followed by both `ToDo.js` and `ToDoSpec.js` . The reason being that `ToDoSpec.js` will need to read the methods from `ToDo.js` in order to check if they behave the way we want them to.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Testing with Jasmine</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/li
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.8.0
```

```
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.8.0
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.8.0
    <script src="ToDo.js"></script>
    <script src="ToDoSpec.js"></script>
</head>
<body>
</body>
</html>
```

That's it for the boiler plate. Now let's think a bit about what we want our app to do.

The checklist of things we need our app to do

Testing the functionality, this is the checklist:

- should add an item
- should delete an item
- should mark item as complete

Testing DOM manipulation:

- should inject initial HTML
- should display new item
- should trigger form and add item to todo array

By the end of this tutorial Jasmine will keep track of the above checklist, it will look like this:

When researching I heard about different approaches on testing. One that stuck was the "test first" approach. Which means writing the test then implementing the code that would pass the test. But as I wrote the code for this tutorial I had to do abit a both. However, no matter what approach, I feel that one of the immediate benefits of testing along the way, means that it will force us to think of modularity very seriously.

The basics of Jasmine structure

In my previous tutorial I used if statements to check if my functions did what I needed to do, Jasmine does a similar thing but this way:

```
describe('Testing the functionality, this is the checklist', ()=>{
  it('should add an item', ()=>{
    //...
  })
  it('should delete an item', ()=>{
    //...
  })
  it('should mark item as complete', ()=>{
    //...
  })
})
```

Note how it matches our checklist and the screenshot above. Then we'll group the "Testing DOM manipulation" tests into another `describe` group.

Now lets start tackeling each test.

Should add an item test and implementation

The todo list is going to be an object, it will then have methods as part of its prototype to modify the todo list.

In `ToDoSpec.js` we'll start the first test.

```
describe('Testing the functionality, this is the checklist', ()=>{
  it('should add an item', ()=>{
    let todo = new ToDo();
    let item = {
     title: "get milk",
     complete: false
   }
    const done = todo.addTodo(item)
    expect(todo.getItems().length).toBe(1);
  })
})
```

For the first test, we try to create an instance of `ToDo()` object, then pass a hard coded (fake list) item object to `todo.addTodo`, then **the most important part:** we check if it worked, by checking if our item is correctly stored. In plain English, we are asking Jasmine to "expect" `todo.getItems().length` to return the length of the items array, and for it to be `1` (since) we just added one item in an otherwise empty array (at this point we do not care if it's an array, but that's what it will be).

Open `SpecRunner.html` in the browser. We'll obviously get an error. It will say "ToDo is not defined".

Let's pass that test

In `ToDoSpec.js` we are trying to test the production code which will be stored in `ToDo.js` . So open that file and let's try to fix the errors in the test.

The first thing the test tries to do is instantiate `ToDo` object. Create that then refresh the `SpecRunner.html` in the browser

```
function ToDo(){
  this.todo = [];
}
```

Now the `ToDoSpec.js` tries to run `todo.addTodo` , which does not exist.

Lets write the entire code we need to pass the test:

```
function ToDo(){
  this.todo = [];
}
ToDo.prototype.addTodo= function(item){
  this.todo.push(item)
}
ToDo.prototype.getItems= function(){
  return this.todo
}
```

That passes the test. We have the `addTodo`, `getItems` methods (otherwise known as getter and setters).

Should delete an item test and implementation

The implementation of each test and functionality will follow the same pattern, we create the test then the method which passes the test

```
it('should delete an item', ()=>{
  let todo = new ToDo();
  let item = {
    id: 1,
    title: "get milk 1",
    complete: false
   }
   let item2 = {
    id: 2,
    title: "get milk 2",
    complete: false
   }
   todo.addTodo(item)
   todo.addTodo(item2)
  todo.delete(2)
  expect(todo.getItems()[todo.getItems().length-1].id).toBe(1);
})
```

To test the delete feature, we need to add an item, then be able to delete it. We are adding two items to test that the `delete` method is actually deleting the one we want.

We now need to create the `delete` method over at `ToDo.js`

```
ToDo.prototype.delete = function(id){
 this.todo =  this.todo.filter(item => item.id !== id)
}
```

As we planned in the test, we filter through the items and remove items that don't have the `id` which is passed.

Should mark item as complete test and implementation

We want to be able to change the property of `complete` from `false` to `true`. Again, to be sure that it's done right I'm adding to items and trying to change one of them to complete(the more I think about it, the less I think it's required but it makes mee feel safe that it actually works).

```
  it('should mark item as complete', function(){
    let todo = new ToDo();
    let item = {
      id: 1,
      title: "get milk 1",
      complete: false
     }
     let item2 = {
      id: 2,
      title: "get milk 2",
      complete: false
     }
     todo.addTodo(item)
     todo.addTodo(item2)
     todo.complete(2)
    expect(todo.getItems().find(item => item.id == 2).complete).toBe(
  })
```

Above we expect the item by `id` of `2` to have the property `complete` to be set to true.

The actual `todo.complete` method will be:

```
ToDo.prototype.complete = function(id){
  this.todo.find(item => item.id == id).complete = true;
}
```

Refactoring the code

As it can be seen We are initializing the `ToDo` object on every test. Jasmine allows us to run some code before every test.

At the top of all our tests, we can add the code that is clearly being duplicated

```
describe('Testing the functionality, this is the checklist', ()=>{
  let todo, item, item2;
  beforeEach(function(){
    todo = new ToDo();
    item = {
      id: 1,
      title: "get milk 1",
      complete: false
     }
     item2 = {
      id: 2,
      title: "get milk 2",
      complete: false
     }
  })
  //...
})
```

Pretty cool! Of course, we would then remove those duplicated snippets from each test case.

And there we have them, All the tests we planned to check from the "Testing the functionality" pass with flying (green) colours!

Testing DOM manipulation

In this batch of tests, we want to make sure that DOM injections work as we expect.

For these new set of tests, we use a new `describe` method. We also make use to the `beforeEach` method to instantiate the `DomManipulation` object (we'll have to create it) and we create a dummy item (which we'll use later)

```
describe('Testing DOM manipulation', function(){
  let Dom, item, todo;
  beforeEach(function(){
     todo = new ToDo();
    Dom = new DomManipulation();
    item = {
      complete: false,
      id : 1,
      title: 'some Title'
    }
 })
// it methods will go here ...
})
```

Interestingly, if we refresh the browser, still pointing to `SpecRunner.html` , we would not see an error even though

`DomManipulation` does not exist. Which proves, `beforeEach` really runs only if we have a test. Let's create the first.

should initialise HTML

If you recall, we don't have anything in the `index.html` . I chose this approach so that I can test-drive this framework. So we need to create the DOM nodes. That's the first test.

```
it('should initialise HTML', function(){
    const form = document.createElement('form');
    const input = document.createElement('input')
    const ul = document.createElement('ul')
    input.id = "AddItemInput"
    form.id="addItemForm"
    form.appendChild(input);
    expect(Dom.init().form).toEqual(form)
    expect(Dom.init().ul).toEqual(ul)
  })
```

Above we want to make sure that `Dom.init()` creates the correct DOM nodes. **Note that we can have multiple expectations**, we want `Dom.init()` to produce a form and an unordered list.

In `ToDo.js` we can create `DomManipulation` and it's `init` method

```
function DomManipulation(){}
DomManipulation.prototype.init = function(){
  const form = document.createElement('form');
  const input = document.createElement('input')
  const ul = document.createElement('ul')
  input.id = "AddItemInput"
  form.id="addItemForm"
```

```
  form.appendChild(input);
  return {
    form, ul
  }
}
```

should create item

When a user submits an item we want a list DOM element to be created. Since this is testing the reaction of the element and not the form submition, we faked the data, pretending it came from the form (`item` is the object we created in `beforeEach` method).

```
it('should create item', function(){
    const element = Dom.displayItem(item);
    const result = document.createElement('li');
    result.innerText = item.title
    expect(element).toEqual(result)
  })
```

`Dom.displayItem` should create the exact element we created in the test. So let's create that method:

```
DomManipulation.prototype.displayItem = function(item){
  const li = document.createElement('li');
  li.innerText = item.title
  return li;
}
```

should trigger form and add item to todo array

This was by far the hardest part for me to accept. **I feel as though it's a hack!**

We need to check if the form is submitted and that the input is added to the todo array (from the previous implementation).

Since the tests are automated, and that we do not have access to the original DOM, the form, input and trigger has to be faked! Let's have a look at the test.

```
it('should trigger form and add item to todo array', function(){
  const form = document.createElement('form');
  form.innerHTML= `<input value="get milk" />
    <button type="submit" />`;
  document.body.appendChild(form)
  const ul = document.createElement('ul');
  Dom.addTodoEvent(
    form,
    todo.addTodo.bind(todo),
    ul)
  form.getElementsByTagName('button')[0].click();
  document.body.removeChild(form)
  expect(todo.todo[0].title).toEqual('get milk')
})
```

We create the form and a hard-coded input, which the user would otherwise add. Then the form is injected to the DOM! **That's the only way to trigger the event**. Then we run `Dom.addTodoEvent` passing it the form, the `todo.addTodo` method and an un ordered list.

Finally we "fake" the form submition, and **Remove the form from the DOM** (otherwise it would be seen in the browser, when loading `SpecRunner.html`).

At the end, we expect an item to be added, with the same title we added to the form's input.

I feel that there must be a better way than adding and removing DOM elements like that!

Finally, let's create the `DomManipulation.prototype.addTodoEvent` which the above test expects

```
DomManipulation.prototype.addTodoEvent = function(form, createTodo, u
  const displayItem = this.displayItem;
  const id = new Date().getUTCMilliseconds();
  form.addEventListener('submit', function(e){
        e.preventDefault();
        const input = document.querySelector('input').value
        const item = {complete: false,id : id, title: input}
        createTodo(item);
        unorderedList.appendChild(displayItem(item))
    })
}
```

The `addTodoEvent` processes the form. It requires the form, the method which processes the form's output, and the DOM which should be changed.

Conclusion

I really like this. In the long run, it would make the process of adding functionality or modifying existing code a lot easier. Also, the more I'll use the "test first" approach, the more modular my code will end up being. I still feel uneasy that I might be missing

something by adding and removing DOM elements like in the last test though, what do you think?
