Jasmine - 5

Wednesday, 19 May 2021 22:02

Spy Matchers - Jasmine Spies

Spies create test doubles and help us isolate dependencies for true unit testing

Test Doubles

A test double is an object that can stand in for a real object in a test, similar to how a stunt double stands in for an actor in a movie

Spies

Jasmine has test double functions called spies
A spy can stub any function and tracks calls to it and all arguments
A spy only exists in the describe or it block in which it is defined, and will be removed after each spec

Spy Matchers

There are special matchers for interacting with spies

toHaveBeenCalled
toHaveBeenCalledWith
toHaveBeenCalledTimes

Section notes

How to spy on a property (getter or setter)

Downloadable slides and notes for this section are at the downloads section at the bottom of this page.

Testing when methods are called, or stubbing their responses with a predetermined canned behavior, are some common scenarios we find ourselves in when it comes to unit testing our JavaScript code.

As JavaScript ES6 adoption increases by developers through the use of compilers (babel, traceur, typescript, etc…) as well as browsers increase support to the not so new features ES6 offers, we find that we need new ways to complete some of these testing tasks: we usually know where we want to go but not how to get there.

Stubbing properties is one of these situations. Let's take a look:

Suppose we have `Person` class:

```
1.  class Person {
2.    constructor(firstName, lastName) {
3.      this.firstName = firstName;
4.      this.lastName = lastName;
5.    }
6.
7.    get fullName() {
8.      return `${this.firstName} ${this.lastName}`;
9.    }
10. }
```

Here we have the ES6 class syntax, with a getter property. The `get fullName()` syntax binds an object property `fullName` to a

`fullName()` syntax binds an object property `fullName` to a function that will be called when that property is looked up.

So when we have an object with a prototype that links to `Person` (an object instance of `Person`) and we read its `fullName` property, we are actually calling the method bound to the `fullName` property and getting what that method returns, like this:

```
11. const person = new Person('John', 'Doe');
12.
13. person.fullName;
14. // => "John Doe"
```

In the past, stubbing these getters with jasmine or even spying on them wasn't easy. But in early 2017, a new method was added: `spyOnProperty`.

When we look at the signature, we notice that it is very similar to the `spyOn` method but with a third optional parameter:

```
spyOnProperty(object, propertyName, accessType)
```

Where

- `object` is the target object where you want to install the spy on.
- `propertyName` is the name of the property that you will replace with the spy.
- `accessType` is an optional parameter. It is the `propertyName` type. its value can be either `'get'` or `'set'` and it defaults to `get`.

So, to spy on our `fullName` property in the `person` object, we could just write:

could just write:

`spyOnProperty(person, 'fullName', 'get')` or even better, just `spyOnProperty(person, 'fullName')` will do it. Easy.

As I mentioned earlier, it is just like `spyOn` so we can assign its returned reference to a new variable, we can stub the response and we can assert calls:

```
15.  const spy = spyOnProperty(person, 'fullName').and.returnValue(
16.    'dummy stubbed name'
17.  );
18.
19.  expect(person.fullName).toBe('dummy stubbed name');
20.  expect(spy).toHaveBeenCalled();
```

We can also use `callThrough` to execute the method in test and get its real returned value:

```
21.  // Installs our spy
22.  const spy = spyOnProperty(person, 'fullName').and.callThrough();
23.
24.  // Here we expect 'John Doe', what the real method returns.
25.  expect(person.fullName).toBe('John Doe');
26.
27.  // Still we can assert calls on spy
28.  expect(spy).toHaveBeenCalled();
```

Or `callFake` to execute and return any implementation needed for our method:

```
29.  const spy = spyOnProperty(
30.    person,
31.    'fullName'
32.  ).and.callFake(function() {
33.    // Perform some operations needed for this specific test
34.    let someString = 'Fake';
35.    let someResult = 'result';
36.
37.    return someString + someResult;
38.  });
```

```
});

expect(person.fullName).toBe('Fakeresult');
expect(spy).toHaveBeenCalled();
```

Or simply do nothing when we just care about the calls and not the actual result:

```
const spy = spyOnProperty(person, 'fullName');
const result = person.fullName;
expect(spy).toHaveBeenCalled();
```

What about object properties with other values?

Let's write the `Person` definition in ES5 (without the class syntax), written as an *Immedietaly-Invoked Function Expression (IIFE)*:

```
var Person = (function () {
  function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  Object.defineProperty(Person.prototype, 'fullName', {
    get: function () {
      return this.firstName + ' ' + this.lastName;
    },
    enumerable: true,
    configurable: true
  });

  return Person;
})();
```

When we write specs against this code using `spyOnProperty`, everything works as expected.

Our `get` is defined through the `Object.defineProperty` method. This is the mechanism used to install that property on the `Person`'s prototype.

Functions are ultimately objects in JavaScript, and objects have prototypes, so the code above is just defining a new property on the prototype for the `Person` constructor function that our IIFE

prototype for the `Person` constructor function that our IIFE returns.

Imagine now that instead we have a `person` object literal with the same property that we want to spy on, but this time it is just a property, no methods bound to it:

```
var person = {
  fullName: 'John Doe' // Just a string
}
```

By calling`spyOnProperty(person, 'fullName')` in our specs, we get an exception: *Property fullName does not have access type get*. But why?

Under the hood, `spyOnProperty` is getting the property descriptor (a precise description of the property) and checking if the access type `get|set` for that property descriptor exists. The bird's-eye view inside jasmine would look like this:

```
const descriptor = Object.getOwnPropertyDescriptor(
  person,
  'fullName'
);

if(!descriptor[accessType]) {
  // throw new Error(...);
}
```

When jasmine checks the `fullName` property descriptor for the `person` literal, `accessType='get'`, so an Error is thrown as `descriptor['get']` does not exist for this property.

An instance of `Person` (object with a prototype linked to the function that the IIFE returns) passes that check as `descriptor['get']` exists.

After the check, the descriptor is used to create the actual spy and its strategies (spy and restore). `spyOnProperty` is implemented

its strategies (spy and restore). `spyOnProperty` is implemented with a function behind the property defined through *Object.defineProperty*. So, you cannot use `spyOnProperty` to install spies on traditional object literal properties as these don't use *Object.defineProperty*, and this is why jasmine performs the descriptor check and throws an Error before trying to set the spy.

But don't get confused when using `get` inside an object literal, that works just fine:

```
var person = {
  get fullName() {}, // can use spyOnProperty
}
```

`spyOnProperty` works as `get` binds a method to the property, `get` is part of the property descriptor and this `person` literal could be defined as:

```
var person = Object.defineProperties({}, {
  fullName: {
    get: function get() {},
    configurable: true,
    enumerable: true
  }
});
```

Originally published on Medium by Juan Lizarazo