12
The Future – Jasmine 2.0

Whenever a new version of a library is released we always feel the rush to start using it. It is an understandable feeling, driven by our need to use the cutting edge.

It is in this spirit that this chapter has been written. To demonstrate what the future holds for Jasmine 2.0, why you should still be using 1.3.1 now, and how to perform a migration when the right time comes.

Release candidate 2

At the time of writing of this book, Jasmine 2.0 was still a release candidate, which means it was still an unstable release, and some of the concepts presented here might have changed in the final release of the library.

As a good piece of advice, it might not be a good idea to use the 2.0 version on any production code until it has reached the final release, as there might be some bugs, and the API might change.

The official release notes for this release can be seen at `https://github.com/pivotal/jasmine/blob/v2.0.0.rc2/release_notes/20rc1.md`.

Breaking changes

There are a lot of small breaking changes in this release. Most importantly, in the way custom matchers are created, which for instance, breaks the jasmine-jquery library. For this reason alone, porting the entire application specs to Jasmine 2.0 is not currently feasible.

To guide you through the major changes of this release, we are going to take a baby-step approach, and see how we would make changes in isolation. Be sure to check the attached source files to understand better how these changes apply in the project.

At the end of this chapter, we are going to see a solution to roll an incremental migration from 1.3.1 to 2.0 that could be applied once Jasmine 2.0 reaches its final release.

New syntax to create custom matchers

Jasmine 2.0 comes with a new way of creating custom matchers. A lot of refactoring has been done under the hood, and the most important change is that internally Jasmine uses this same infrastructure to create its own built-in matchers.

The concept of custom matchers was first presented in the *Chapter 2*, *Your First Spec*, in the form of the `toBeAGoodInvestment` matcher:

```
this.addMatchers({
  toBeAGoodInvestment: function() {
    var investment = this.actual;
    var what = this.isNot ? 'bad' : 'good';

    this.message = function() {
      return 'Expected investment to be a '+ what +' investment';
    };

    return investment.get('isGood');
  }
});
```

Back then, it was pretty simple to define a custom matcher, a function that when invoked would either return a Truthy or Falsy result to indicate either a success or failure of the expectation.

We would get the matcher's parameters via function arguments, and the actual value via the `this.actual` attribute.

Further on, we would use the `this.isNot` to check whether the matcher was being called as a negation and the `this.message` to define custom error messages.

So how does it change in the 2.0 release?

Given here is the previous matcher written in the new syntax:

```
jasmine.Expectation.addMatchers({
  toBeAGoodInvestment: function() {
    return {
      compare: function (actual) {
        var pass = actual.isGood();
        var what = pass ? 'bad' : 'good';

        return {
          pass: pass,
          message: 'Expected investment to be a '+ what +' investment'
        };
      }
    };
  }
});
```

The first thing you notice is the new `jasmine.Expectation.addMatchers` function to add matchers. It still expects an object with new matchers to add, but the actual matcher definition is quite different.

The function passed as the value of the `toBeAGoodInvestment` attribute is now expected to return an object containing a single attribute named `compare`.

This `compare` attribute is going to contain a function with the comparison logic that determines if this matcher expectation was either a success or a failure. It receives everything it needs via arguments, the first being as always the `actual`, and the remainder as the other parameters passed to the matcher when invoked.

The result of the `compare` function must be an object with two attributes:

- `pass`: This is a Truthy or Falsy value indicating the result of the comparison
- `message`: This is the error message to be shown if the pass value is to be considered a failure

To better understand how the error message works, let's take our example; if the **pass** value is `true`, this means that the investment is good, but the matcher was called as a negation (`expect(actual).not.toBeAGoodInvestment()`), then the error message must be **Expected investment to be a bad investment**.

It is a simple implementation, but can be a little less intuitive than the previous solution of using the `this.isNot` attribute.

New syntax for asynchronous specs

Inspired by **Mocha** (the testing framework) and the way it does asynchronous testing, Jasmine now comes with a much simpler solution for testing asynchronous code.

As we had seen in *Chapter 4*, *Asynchronous Testing – AJAX*, Jasmine provided two global functions, `runs` and `waitsFor` to implement asynchronous specs. In its 2.0 incarnation, Jasmine has a much simpler solution.

Let's take a look at the spec here ported to Jasmine 2.0:

```
describe("Stock", function() {
  var stock;

  beforeEach(function() {
    stock = new Stock({
      symbol: 'AOUE'
    });
  });

  describe("when fetched", function() {
    beforeEach(function(done) {
      stock.fetch({
        success: function () {
          done();
        }
      });
    });

    it("should update its share price", function() {
      expect(stock.sharePrice).toEqual(20.18);
    });
  });
});
```

Instead of having the `waitsFor` function with a timeout and error message, we have a single `done` callback argument that when present, will make Jasmine consider its corresponding block (`beforeEach` in the preceding example) as asynchronous, and wait for the `done` callback to be invoked before running any other block (`it` in the preceding example).

This is now the only way to carry out asynchronous testing in Jasmine 2.0. There is no longer something such as the `runs` function.

New syntax for spies

The syntax to create new spies remains the same in the new release, with the familiar `spyOn`, `jasmine.createSpy`, and `jasmine.createSpyObj` functions.

What has changed is what you can do with those spies once they have been created.

As an example, in Jasmine 1.3.1 you could tell a spy to return a fake value with the `andReturn` function as follows:

```
jasmine.createSpy('mySpy').andReturn('stubbed value');
```

In Jasmine 2.0, to achieve the same result you would write:

```
jasmine.createSpy('mySpy').and.callReturn('stubbed value');
```

Since they are very similar, we can write a table to map the different functions between releases:

1.3.1	2.0.0-rc2
`andCallThrough`	`and.callThrough`
`andReturn`	`and.callReturn`
`andCallFake`	`and.callFake`
`andThrow`	`and.callThrow`

Migrating from 1.3.1 to 2.0.0

There will come a time when Jasmine 2.0 gets its final release and you might want to start writing tests using the new and shiny version of the library.

It turns out that to do this, you don't need to rewrite all your tests in a single batch, but instead do a more incremental and sane approach. That is possible because of another change that happened in this new release; it no longer pollutes the global namespace.

But you might be thinking, "the examples showed in this chapter were all using global functions". And you are right! They are global functions, but they are not being defined by Jasmine itself, instead by the simple `boot.js` file.

This is a new file added to the standalone release that sets up the global Jasmine functions to work in the browser environment.

You can check that by opening this file. In there you can see a number of variables being created and attached to the `window` object, therefore making them global:

```
window.jasmine = jasmineRequire.core(jasmineRequire);

var env = jasmine.getEnv();

var jasmineInterface = {
  describe: function(description, specDefinitions) {
    return env.describe(description, specDefinitions);
  },

  it: function(desc, func) {
    return env.it(desc, func);
  }
  // other jasmine functions: beforeEach, afterEach...
};

extend(window, jasmineInterface);
```

With that in mind, we could change this `boot.js` file to instead create these global functions with different names.

We could, for instance, make all 2.0 functions with a trailing underscore. Something such as `describe_`.

At the `SpecRunner.html` file we could refer to both versions of the library and have new tests written using the new global functions:

```
describe_("Player", function() {
  var player;
  var song;

  beforeEach_(function() {
    player = new Player();
    song = new Song();
  });

  it_("should be able to play a Song", function() {
    player.play(song);
    expect_(player.currentlyPlayingSong).toEqual(song);

    //demonstrates use of custom matcher
    expect_(player).toBePlaying(song);
  });
});
```

And the old ones would still work.

Then slowly, we could change spec by spec to use these new functions. And once we had ported everything, it is all a matter of creating a small script to update all the spec files to remove the trailing underscore, restore the `boot.js` file to its original implementation and have the application ported to the 2.0 release.

Be sure to check the attached code for a working scenario.

Summary

In this chapter, we have learned about the upcoming changes of Jasmine 2.0 and how we could perform a migration from 1.3.1 to 2.0.

We have seen how the 2.0 release breaks compatibility with third-party libraries, such as `jasmine-jquery`, and that it might be a good idea to wait for the final release of the 2.0 version before we start porting legacy code.

It was demonstrated that the best approach is to continue development in the stable release, and in the future use the migration process explained earlier to move to the new release once it becomes stable.