One of the benefits of using AngularJS is the ability to unit test the JavaScript code in a complex application. Unit testing is incredibly easy for trivial cases when controllers and models are declared in global scope. However unit testing is slightly more challenging for objects defined inside of Angular modules because of the need to bootstrap modules, work with a dependency injector, and deal with the subtleties of nested functional code.

Let’s try to test the following controller defined in a module:

```
(function (app) {

    var SimpleController = function ($scope) {

        $scope.x = 3;
        $scope.y = 4;
        $scope.doubleIt = function () {
            $scope.x *= 2;
            $scope.y *= 2;
        };
    };

    app.controller("SimpleController",
              ["$scope", SimpleController]);

}(angular.module("myApp")));
```

We’ll be using AngularJS mocks and Jasmine in an HTML page, which requires the following scripts:

- jasmine.js

- jasmine-html.js

- angular.js

- angular-mocks.js

- simpleController.js (where the controller lives)

It’s important to **include the Jasmine scripts before including angular-mocks**, as angular-mocks will enable some additional features when Jasmine is present (notably the helper methods *module* and *inject*).

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

    beforeEach(module('myApp'));

    describe("SimpleController", function() {
        var scope;
        beforeEach(inject(function($rootScope, $controller) {
            scope = $rootScope.$new();
            $controller("SimpleController", {
                $scope: scope
            });
        }));

        it("should double the numbers", function() {
            scope.doubleIt();
            expect(scope.x).toBe(6);
        });
    });
});
```

The *module* method used in the first *beforeEach* (which you can also invoke as *angular.mocks.module*) will initialize and configure the myApp module and its dependencies.

The *inject* method in the second *beforeEach* (*angular.mocks.inject*) takes a function that requires dependencies. Behind the scenes, the *inject* method will set up an $injector and use it to invoke the function with the right dependencies. In this test, we need the application's $rootScope object and the $controller service. We are going to use the $controller service to have complete oversight over the instantiation of the SimpleController and keep a scope object around that we can write asserts against.

Once the setup code is done, the *it* tests are relatively easy to write (and read). The trick is figuring out what to inject, what to instantiate directly, and (a topic for future posts) what to mock or fake by hand.