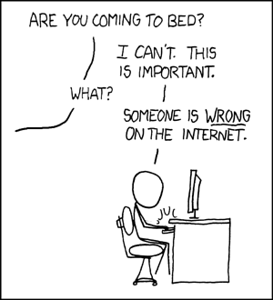
* Mocha and Jasmine are testing frameworks to test your code: they can use different assertion libraries, reporters, etc...
* should.js is an assertion library - It works from IE9 onward and any other browser - so you need a testing framework to use it
* chai is an assertion library "ecosystem": you can add plugins (I'm looking at you sinonJS) or just use it plain with its own version of should.js or expect.js - also here you need a testing framework in order to use it
* karma is a testing environment that leverages the power of Mocha and Jasmine to test cross-browser, do E2E testing, and more
* [Karma](http://karma-runner.github.io/0.12/index.html) is an example of a **test runner**
* [Mocha](http://mochajs.org/) is an example of a **testing framework**
* [Chai](http://chaijs.com/) is an example of an **assertion library**
* Karma: Test Runner
* Karma is a type of test runner which creates a fake server, and then spins up tests in various browsers using data derived from that fake server. Karma is only a test runner, and requires a testing framework such as Mocha to plug into it in order to actually run tests.
* Test Runners work on the highest level of abstraction out of all testing software. All the other testing software takes place within the test runner. As a result, it is necessary to configure the test runner to work with the other testing software which plug into it. In our example of using Karma as a test runner, **karma init** is the command which creates the **karma.conf.js** file. This file is where all our test runner configurations will be.
* The header of each karma configuration file follows this structure:
* module.exports = function(config) {
* config.set({
* basePath: '../..',
* frameworks: ['mocha']
* As you can see, this configuration file offers us a hook to include our testing framework, which exist on a lower level of abstraction than our test runner.
* Additionally, Karma uses other technologies such as the [minimatch library](https://github.com/isaacs/minimatch), which uses regex-like syntax to filter the correct files to use. The **basePath**property determines the root path Karma uses to find the actual specs, which exist on a lower level of abstraction.
* Mocha: Testing Framework
* The following file uses Mocha as a testing framework, and Chai as an assertion library:
* *describe('the todo.App', function() {*
* *context('the todo object', function(){*
* *it('should have all the necessary methods', function(){*
* *var msg = "method should exist";*
* *expect(todo.util.trimTodoName, msg).to.exist;*
* *expect(todo.util.isValidTodoName, msg).to.exist;*
* *expect(todo.util.getUniqueId, msg).to.exist;*
* *});*
* *});*
* *});*
* How to Distinguish Between Mocha and Chai
* How can we distinguish between which level of abstraction each part of the testing suite is occurring on? In other words, we are now working on a lower level than a test runner, we are actually writing tests now. We are using a lot of methods which are native to JavaScript. How do we distinguish which method is from which piece of testing software? It was easy to differentiate between a test runner and a framework. But now we are throwing a framework and an assertion library all in the same file. It is becoming muddled as to which method is a result of which piece of software.
* We can distinguish between framework (Mocha) methods and assertion library (Chai) methods by looking at the contents of the **it** block. Methods outside the **it** block are generally derived from the testing framework. Everything within the **it** block is code coming from the assertion library.**beforeEach**, **describe**, **context**, **it**, are all methods extending from Mocha. **expect**, **equal**, and **exist**, are all methods extending from Chai.
* afterEach(function() {
* $httpBackend.verifyNoOutstandingExpectation();
* $httpBackend.verifyNoOutstandingRequest();
* $window.localStorage.removeItem('com.shortly');
* });
* it('should have a signup method', function() {
* expect($scope.signup).to.be.a('function');
* });
* All the methods concerned with the testing framework are occurring outside the **it** block, and all methods concerned with the assertion library are occurring inside the it block. Therefore we can conclude that anything occurring inside the it block is indeed occurring on a lower level of abstraction than the testing framework. Or in terms of our classification schema, everything occurring inside the it blocks is either part of an assertion library or a part of a testing plugin. The notion that anything inside the it block is occurring on a lower level of abstraction than the testing framework is only a heuristic, that is- it is merely a rule of thumb.
* 
* There are many technologies out there and unlimited edge cases to predict, but as a generality it would be fair to view the it block as the interface between two different levels of abstraction in testing.
* Chai: Assertion Library
* Chai is an assertion library that plugs into Mocha. Up until now we have just been concerned with using a test runner for automation, and using a testing framework for setup, teardown, and structure. But now we are getting into the meat and potatoes. The assertion library is what actually runs the specs and determines whether any given condition is valid or not. Ultimately, every test is ran by methods which are derived from our assertion library. It is worth mentioning though, not every framework needs an external assertion library. [Jasmine](http://jasmine.github.io/) for example, has it's own assertion library builtin. Mocha is just structured in such a way where it **does** need an external assertion library. This makes Mocha more difficult to setup initially, but offers much greater flexibility than frameworks which use a builtin assertion library such as Jasmine.

Importance of call back

What is a callback? A callback is a function to be executed after another function is executed. Sounds tongue-twisted? Normally if you want to call function do\_b after functiondo\_a the code looks something like

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | function do\_a(){    console.log( '`do\_a`: this comes out first');  }    function do\_b(){    console.log( '`do\_b`: this comes out later' );  }    do\_a();  do\_b(); |

Result

|  |  |
| --- | --- |
| 1  2 | `do\_a`: this comes out first  `do\_b`: this comes out later |

However javascript is an event driven language. If do\_a takes longer than do\_b, the result of do\_b comes out first than do\_a;

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13 | function do\_a(){    // simulate a time consuming function    setTimeout( function(){      console.log( '`do\_a`: this takes longer than `do\_b`' );    }, 1000 );  }    function do\_b(){    console.log( '`do\_b`: this is supposed to come out after `do\_a` but it comes out before `do\_a`' );  }    do\_a();  do\_b(); |

Result

|  |  |
| --- | --- |
| 1  2 | `do\_b`: this is supposed to come out after `do\_a` but it comes out before `do\_a`  `do\_a`: this takes longer than `do\_b` |

So how do we make sure do\_b comes out after do\_a in that situation? This is where callbacks comes in handy.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17 | function do\_a( callback ){    setTimeout( function(){      // simulate a time consuming function      console.log( '`do\_a`: this takes longer than `do\_b`' );        // if callback exist execute it      callback && callback();    }, 3000 );  }    function do\_b(){    console.log( '`do\_b`: now we can make sure `do\_b` comes out after `do\_a`' );  }    do\_a( function(){    do\_b();  }); |

Result

|  |  |
| --- | --- |
| 1  2 | `do\_a`: this takes longer than `do\_b`  `do\_b`: now we can make sure `do\_b` comes out after `do\_a` |