## The Chai.js library allows 3 syntaxes for writing expectations.

value.should.XXX

expect(value).XXX

assert.XXX(value)

Normally we use expect() method like this

**expect(valueUnderTest).foo.bar.whatever;**

### For Example:

const actual = `There is no God but Allah and Hazrat Muhammad (saw) is the messenger of Allah`;

const expected = `Hazrat Muhammad (saw)`;

chai.expect(actual).to.contains(expected);

### Properties

**Chai.js includes 13 assertion properties:** ok, true, false, null, undefined, NaN, exist, empty, arguments, itself, sealed, frozen, and finite.

**Chai includes 15 cosmetic properties:** to, be, been, is, that, which, and, has, have, with, at, of, same, but, and does.

**Chai includes 7 flagging properties**: not, deep, nested, own, ordered, any, and all

The string ‘foo’ is expected to be undefined

**expect(‘foo’).undefined; // fails**

Cosmetic Properties for human readers of the code

**expect(‘foo’).to.be.undefined; // fails**

**expect(‘foo’).to.not.be.undefined; // passes**

### Method

A method is an assertion property which takes one or more additional values.

**expect(‘foo’).to.equal(‘bar’); // fails**

**Chai includes quite a few methods:** a (aliased as an), include (aliased as includes, contain, and contains), equal, eql, above, below, least, most, within, instanceof, property, ownPropertyDescriptor, lengthOf, match, string, keys, throw, respondTo, satisfy, closeTo, members, oneOf, change, increase, decrease, by, and fail.

**Chai includes 4 methods which can be treated as properties**: a (aliased as an), include (aliased as includes, contain, and contains), length, lengthOf

It’s possible to mix methods and properties.

**expect(‘foo’).to.equal(‘foo’).and.not.be.undefined; // passes**

Flagging properties adds a flag which applies to all properties and methods which come after the flagged property in the expectation chain. This means order matters.

**expect(‘foo’).to.not.be.undefined.and.equal(‘foo’) // fails**

In the above not this which causes both the undefined and equal to expect that the value under test is not undefined, and not equal to the string ‘foo’.

### Chainable Method

A chainable method combines many of the previous types into one keyword. Chainable methods can be used either as a method, giving it a value, or as a property, or giving it no values.

**expect([1, 2, 3]).to.include(3); // passes**

**expect({ a: 1, b: 2 }).to.include.keys('a'); // passes**

## Testing Arrays and Objects with Chai.js

When it comes to testing arrays and objects with Chai.js sometimes the selection of flagging properties and assertions becomes confusing. nested? deep? own? include? All?

### Equality

Making assertions on arrays and object.

**expect([1, 2, 3]).to.equal([1, 2, 3]); // fails**

The equality being expressed in the example above is actually a core mechanic of Javascript, not of Chai.js.

$ node

> [1,2,3] === [1,2,3]

false

Javascript equality is strict; because each copy has it’s own address in memory, Javascript’s strict equality considers them not equal as they do not share the same identity.

### Deep Equality

Chai.js solves this problem by providing a second equality assertion, eql. The Eql is based on the [deep-eql project](https://github.com/chaijs/deep-eql).

**expect([1, 2, 3]).to.eql([1, 2, 3]); // passes**

Deep equality compares sameness to all depths.

**expect([{a:1}, {b:2}]).to.eql([{a:1}, {b:4}]); // fails**

**expect([{a:1}, {b:2}]).to.eql([{a:1}, {b:2}]); // passes**

Similar to arrays, comparing two objects will use strict or deep equality.

**expect({ a: 1 }).to.equal({ a: 1 }); // fails**

**expect({ a: 1 }).to.eql({ a: 1 }); // passes**

### Unordered Deep Equality

Deep equality is an excellent approach, but it enforces both order and contents when comparing arrays. Often the case arises in testing where the order doesn’t matter, but the contents do

**expect([1,2,3]).to.eql([3,2,1]); // fails**

While *eql* compares content and enforces order, members only compares content allowing assertions that only care about values being present

**expect([1,2,3]).to.have.members([3,2,1]); // passes**

**expect([1,2,3]).to.have.members([1,2,3]); // passes**

**Danger Zone**: A common mistake is to write the above assertion with *include* instead of *have*.

**expect([1,2,3]).to.include.members([3,2,1]); // passes**

The distinction between *include* and *have* is an important one. Have is a cosmetic property. It does nothing to the expectation but make it easier to read. However include is a chainable method. It is setting a flag which changes the behavior of members and the length of the elements doesn’t matter.

**expect([1,2,3,4]).to.include.members([3,2,1]); // passes**

**expect([1,2,3,4]).to.have.members([3,2,1]); // fails because length is matter**

Between ordering and the exactness of members, there’s a 2x2 truth table of which matcher to use in what instances.

1. Order Wholeness Matters — .to.have.ordered.members like eql

2. Unordered Wholeness Matters — .to.have.members

3. Unordered Membership Matters — .to.include.members

4. Ordered Membership Matters — Impossible case

For unordered array membership Chai have two flagging properties any and all

### Any and All

And and all can be included if it makes the expectation easier to read, but are effectively acting as cosmetic properties. The following two are functionally equivalent.

**expect([1,2,3]).to.have.members([3,2,1]); // passes**

**expect([1,2,3]).to.have.all.members([3,2,1]); // passes**

Similarly, with an array of primitive values (non-objects) it’s possible to write the same expectation with *eql* or *.ordered.members*.

**expect([1,2,3]).to.eql([1,2,3]); // passes**

**expect([1,2,3]).to.have.ordered.members([1,2,3]); // passes**

### 60 Fathoms Deep Equality

The difference between choosing *eql* and *.ordered.members* becomes more obvious when comparing arrays of objects. Mentioned before, eql is an equality assertion in Chai.js which will perform a deep equal instead of a strict equal. And third way to compare two arrays of primitive values is to use the flagging property *deep*.

**expect([1, 2, 3]).to.deep.equal([1, 2, 3]); // passes**

While the expectation above is functionally equivalent to using **eql**, the **deep** flag makes the difference when mixed with other assertions, such as members.

**expect([ {a:1} ]).to.have.deep.members([ {a:1} ]); // passes**

**expect([ {a:1} ]).to.have.members([ {a:1} ]); // fails**

### Object Property Equality

When comparing objects, sometimes it’s only important what properties those objects have, not what the value of the properties are.

**const obj = { a: 1, b: 2 };**

**expect(obj.c).to.not.be.undefined;**

**AssertionError**: expected undefined not to be undefined

As expected, Chai provides the keys and property assertions which can assert the existence of a single property (property) or multiple properties (keys) on an object.

**expect({ a: 1, b: 2 }).to.have.property('b'); // passes**

**expect({ a: 1, b: 2 }).to.have.keys([‘a’, ‘b’]); // passes**

If the value matters, the property assertion can be used with a second parameter, the expected value.

**expect({ a: 1, b: 2 }).to.have.property(‘b’, 2); // passes**

### Deep Object Property Equality

An alternative way to test for the existence of a property is with the include assertion. Include is acting in the same capacity that it did with arrays, checking that the given properties and values are under tests, not that the given whole properties and values are under test.

**expect({ a: 1, b: 2 }).to.include({ b: 2 }); // passes**

* 1. **Important Concept**: By default, all assertions in Chai are performing a strict equality comparison.
  2. Thus, asserting that an array or object of objects has a member object will cause those two objects
  3. to be compared strictly. So in this case it interacts with the deep flag.

**expect({ a: { c: 3 } }).to.include({ a: { c: 3 } }); // fails**

**expect({ a: { c: 3 } }).to.deep.include({ a: { c: 3 } }); // passes**

*deep.include* is useful in some instances, but for anything more than a level or so deep can become unwieldy. For really deep inspections, the nested flagging property can be used. Nested signals that in all places where the property name (key) would have been, it is now a property path. This is particularly helpful when working with extremely complex JSON structures.

**const obj = {**

**query: {**

**bool: {**

**filter: {**

**term: { id: '12345' }**

**}**

**}**

**}**

**};**

// passes

**expect(obj).to.have.nested.property('query.bool.filter.term.id');**

// passes (works with deep as well)

**expect(obj).to.have.deep.nested**

**.property('query.bool.filter.term', { id: '12345' });**

Something interesting about nested is that arrays can also be referenced by path as well.

**const obj = {**

**query: {**

**bool: {**

**filter: [{**

**term: { id: '12345' }**

**}]**

**}**

**}**

**};**

// passes

**expect(obj).to.have.nested.property('query.bool.filter[0].term.id');**

The nested flagging property works with both the property and keys assertion.