TinyTest

Source code of released version | Source code of development version

---

After reading this, you'll know:

1. What Tinytest is used for.
2. How to include tests in your package.
3. How to structure your test file(s).
4. When to use synchronous or asynchronous tests.
5. How to run your tests.
6. What assertions and utility methods are available to you.

What is Tinytest?

Tinytest is the official Meteor package test runner.

It's used to test *local* packages. You cannot test non-local packages (for example packages you have included using `meteor add some:package-name`) unless you clone their repos and make them available locally. The assumption is, of course, that you are testing packages under development.

Tinytest is specifically designed for *unit testing*. From wikipedia:

> *... Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.*

> *... Ideally, each test case is independent of the others. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation.*

The object of a unit test is to prove functionality you have written. It is not to prove third-party or other package or library code which may form a part of your codebase. This may require that you head up your tests with *stubs, mocks/spies* or *fakes* which satisfy the demands of your code, but are non-functional (or have limited functionality). Alternatively, instead of placing the stubs in your test file, you could put them into a separate `stubs.js` file, which you `api.addFiles`, or even create and add a stubs package which you then `api.use`.

From martinfowler.com:

> *Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).*

> *Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.*

> *Mocks [are] objects pre-programmed with expectations which form a specification of the calls they are expected to receive.*

For a comprehensive way of including spies, stubs and other fun stuff, check out practicalmeteor:munit and smithy:describe, both of which wrap a host of useful functionality around Tinytest, including the more widely used `describe ... it ... expect` syntax. This README covers standard Tinytest, so you should refer to the documentation for those if you prefer their syntax and functionality.

Adding Tinytest to Your Package

Tinytest is included through configuration in the `package.js` file using `Package.onTest()`.

```
// Define test section
Package.onTest(function(api) {
  // We will need the following packages to run our tests.
  // Note that we also need to include the package to be tested (myname:mypackage).
  api.use(['tinytest', 'underscore', 'ecmascript', 'myname:mypackage']);
  // In v1.2 test-packages no longer include any packages globally.
  // You may need to make some exports global for your tests to run, for example:
  api.imply('underscore');
  // This file contains the tests we want to run (these will run on the client and
the server)
  api.addFiles('tests.js');
  // You can add as many test files as you want and choose where they are to run:
  api.addFiles('server-tests.js', 'server');
  // and these only run on the client:
  api.addFiles('client-tests.js', 'client');
});
```

Structuring the Test File

The test file is normally written in Javascript/ES5 (but may be anything if the appropriate transpiler is included with `api.use('some-transpiler');` within `Package.onTest`). Examples in this README assume ES2015, for which you will have to include an `api.use('ecmascript');` line in your `Package.onTest`.

> *Note that currently the `package.js` file itself is processed as Javascript/ES5. This cannot be changed.*

A "test" contains the code you want to test, plus zero or more *assertions* wrapped within a `Tinytest.add` or `Tinytest.addAsync` callback. The `name` of the test is specified in the `add` or `addAsync` call. An assertion is a call to a method used to verify an actual result (or some property of an actual result) vs an expected result. For example, "is the result === 1?", or "is the result a JavaScript `Number`?".

If a test has no assertions it is assumed to pass unless an exception is caught. However, it is more usual to explicitly specify assertions.

In any single test all assertions must pass for the test as a whole to pass.

The `name` of the test may also be used to generate a hierarchical (grouped) result page. Hierarchical levels are split using " - " as a separator in the name. The name of the package should be at the head of the `name`:

```
'mypackage - functional tests - test 1'
'mypackage - functional tests - test 2'
'mypackage - performance tests - test 1'
'mypackage - performance tests - test 2'
```

Would be good `name`s to use for reporting test results in a hierarchy.

Multiple test files may be included in `package.js`. They will be processed in the order declared. However, test results are presented in the order in which they appear in their groupings. The pass/fail status of tests is updated in

order, including waiting for *asynchronous* tests to complete.

Synchronous and Asynchronous Tests

Tests may be *synchronous* or *asynchronous*.

A synchronous test is one in which the flow of code *does not* need to wait for the result of an external action to be made available through an on-completion callback.

Conversely, an asynchronous test is one in which the flow of code *does* need to wait for the result of an external action to be made available through an on-completion callback.

Synchronous tests are more common, but asynchronous tests are useful occasionally. With the introduction of Meteor Promises (promises running in fibers), we can now successfully write synchronous tests containing asynchronous code. However, examples of this usage are outside the scope of this section of the README.

Synchronous Tests

```
Tinytest.add(name, (test) => {
  // test body
});
```

Asynchronous Tests

```
Tinytest.addAsync(name, (test, onComplete) => {
  someAsyncRequest((error, result) => {
    // test body
    onComplete(); // invoke when async function completes.
  }
});
```

You can avoid having to call the `onComplete` callback explicitly by returning a `Promise` from the test function, which conveniently enables `async` test functions:

```
Tinytest.addAsync(name, async (test) => {
  test.equal(shouldReturnFoo(), "foo");
  const bar = await shouldReturnBarAsync();
  test.equal(bar, "bar");
});
```

Executing the Tests

Tests are executed by running the (development) application with the `test-packages` command. By default, all packages are tested, but specific packages can be tested by specifying them by name.

`meteor test-packages` runs tests on all (local) packages in the app.

`meteor test-packages her:package his:package` runs tests only on `her:package` and `his:package`.

The results are presented on the browser (port 3000) as usual. When run as shown, the file watcher runs, so code may be edited and the tests will be re-run automatically. If this is undesirable, you may add the `--run-once` switch.

Executing only specific tests

You can replace temporarily `Tinytest.add` or `Tinytest.addAsync` by `Tinytest.only` or `Tinytest.onlyAsync` so only the tests added using `only*` are going to be executed.

This is helpful when only a few tests are failing, so you can focus on them.

Assertions

The `test` object in the above callbacks is used to add assertions to the test.

```
Tinytest.add('mypackage - basic tests', (test) => {

  // Get the result from our function under test.
  const result = myFunction(1);

  // Do the test ... calling myFunction with 1 should return 2.
  test.equal(result, 2);
});
```

Tests may include multiple assertions. The total number of successes and failures for each test are reported.

Assertions with optional fail messages

An optional `message` may be added to the assertions which will change a failure report from `fail - <failure-test>` to `fail - <failure-test> - message <optional message>`

equal

`test.equal(actual, expected[, message[, not]]);`

`not` is a boolean (`true` / `false`) parameter which may be used to switch the sense (`equal` becomes `notEqual`). Used internally by `notEqual` . **Don't use it in your tests - it just makes your code harder to grok.**

Compares by type and value, so 2 is not equal to "2", for example. Is a better choice for checking something is strictly `true` or `false` than `test.isTrue` or `test.isFalse` (see below).

notEqual

`test.notEqual(actual, expected[, message]);`

Internally, calls `equal` with the `not` flag set to `true` .

matches

`test.matches(actual, regexp[, message]);`

Checks `regexp.test(actual)`

notMatches

```
test.notMatches(actual, regexp[, message]);
```

Checks `!regexp.test(actual)`

isTrue

```
test.isTrue(actual[, message]);
```

Checks that `actual` is truthy (does not check type is `Boolean`).

isFalse

```
test.isFalse(actual[, message]);
```

Checks that `actual` is falsey (does not check type is `Boolean`).

isNull

```
test.isNull(actual[, message]);
```

Checks `actual === null`

isNotNull

```
test.isNotNull(actual[, message]);
```

Checks `!(actual === null)`

isUndefined

```
test.isUndefined(actual[, message]);
```

Checks `actual === undefined`

isNotUndefined

```
test.isNotUndefined(actual[, message]);
```

Checks `!(actual === undefined)`

isNaN

```
test.isNaN(actual[, message]);
```

Checks `isNaN(actual)`

isNotNaN

```
test.isNotNaN(actual[, message]);
```

Checks `!(isNaN(actual))`

length

```
test.length(obj, expected_length[, message]);
```

Checks `obj.length === expected_length`

instanceOf

`test.instanceOf(obj, klass[, message]);`

Checks `obj instanceof klass`

notInstanceOf

`test.notInstanceOf(obj, klass[, message]);`

Checks `!(obj instanceof klass)`

include

`test.include(haystack, needle[, message[, not]]);`

`haystack` may be an `array`, `object` or `string`. Correspondingly, `needle` may be an `element` (simple elements only), `key` (method or property name) or `substring`.

`not` is a boolean (`true` / `false`) parameter which may be used to switch the sense (`include` becomes `notInclude`). Used internally by `notInclude`. **Don't use it in your tests - it just makes your code harder to grok.**

notInclude

`test.notInclude(haystack, needle[,message]);`

Internally, calls `include` with the `not` flag set to `true`.

_stringEqual

`test._stringEqual: function (actual, expected[, message]);`

EXPERIMENTAL way to compare two strings that results in a nicer display in the test runner, e.g. for multiline strings.

Assertions without optional fail messages

`test.throws(func, expected);`

`expected` can be:

- `undefined` : accept any exception.
- `string` : pass if the string is a substring of the exception message.
- `regexp` : pass if the exception message passes the regexp.
- `function` : call the function as a predicate with the exception.

Note: Node's `assert.throws` also accepts a constructor to test whether the error is of the expected class. But since JavaScript can't distinguish between constructors and plain functions and Node's `assert.throws` also accepts a predicate function, if the error fails the `instanceof` test with the constructor then the constructor is then treated as a predicate and called (!)

The upshot is, if you want to test whether an error is of a particular class, use a predicate function.

Utility Methods

runId

`test.runId();`

Returns a unique string id for this test (for example 'ZmXxMPyoWGFy5wEiB').

exception

`test.exception(exception);`

Should only be used with asynchronous tests

Call this to fail the test with an exception that occurs inside asynchronous callbacks in tests. If you call this function, you should make sure that:

- the test doesn't call its callback (onComplete function).
- the test function doesn't directly raise an exception.

fail

`test.fail(doc);`

Can be used to output a detailed message about a failure with path and value.

Example call:

```
test.fail({
          type: 'match-error-path',
          message: "The path of Match.Error doesn't match.",
          pattern: JSON.stringify(pattern),
          value: JSON.stringify(value),
          path: err.path,
          expectedPath,
        });
```

expect_fail

`test.expect_fail();`

May be used to change a fail to a qualified pass. The test is counted as a pass, but clicking on it reveals that the underlying test failed:

```
- expected_fail — assert_equal - expected xxx - actual yyy - not
```