TESTING GENERATORS

Read on to learn more about the testing helpers Yeoman add to ease the pain of unit testing a generator.

The examples below assume you use Mocha (http://mochajs.org/) in BDD mode. The global concept should apply easily to your unit testing framework of choice.

Organizing your tests

It is important to keep your tests simple and easily editable.

Usually the best way to organize your tests is to separate each generator and sub-generator into its own `describe` block. Then, add a `describe` block for each option your generator accepts. And then, use an `it` block for each assertion (or related assertion).

In code, you should end up with a structure similar to this:

```
describe('backbone:app', function () {
  it('generates a project with require.js', function () {
      // assert the file exist
      // assert the file uses AMD definition
  });

  it('generates a project with webpack');
});
```

Test helpers

Yeoman provide test helpers methods. They're contained inside the `yeoman-test` package.

```
var helpers = require('yeoman-test');
```

You can check the full helpers API here (https://github.com/yeoman/yeoman-test).

The most useful method when unit testing a generator is `helpers.run()`. This method will return a RunContext (https://github.com/yeoman/yeoman-test/blob/master/lib/run-context.js) instance on which you can call method to setup a directory, mock prompt, mock arguments, etc.

```
var path = require('path');

it('generate a project', function () {
  // The object returned acts like a promise, so return it to wait until the
  return helpers.run(path.join(__dirname, '../app'))
    .withOptions({ foo: 'bar' })      // Mock options passed in
    .withArguments(['name-x'])        // Mock the arguments
    .withPrompts({ coffee: false })   // Mock the prompt answers
    .withLocalConfig({ lang: 'en' }) // Mock the local config
    .then(function() {
      // assert something about the generator
    });
})
```

Sometimes you may want to construct a test scenario for the generator to run with existing contents in the target directory. In which case, you could invoke `inTmpDir()` with a callback function, like so:

```
var path = require('path');
var fs = require('fs-extra');

helpers.run(path.join(__dirname, '../app'))
  .inTmpDir(function (dir) {
    // `dir` is the path to the new temporary directory
    fs.copySync(path.join(__dirname, '../templates/common'), dir)
  })
  .withPrompts({ coffee: false })
  .then(function () {
    assert.file('common/file.txt');
  });
```

You can also perform asynchronous task in your callback:

```
var path = require('path');
var fs = require('fs-extra');

helpers.run(path.join(__dirname, '../app'))
  .inTmpDir(function (dir) {
    var done = this.async(); // `this` is the RunContext object.
    fs.copy(path.join(__dirname, '../templates/common'), dir, done);
  })
  .withPrompts({ coffee: false });
```

The run Promise will resolve with the directory that the generator was run in. This can be useful if you want to use a temporary directory that the generator was run in:

```
helpers.run(path.join(__dirname, '../app'))
  .inTmpDir(function (dir) {
    var done = this.async(); // `this` is the RunContext object.
    fs.copy(path.join(__dirname, '../templates/common'), dir, done);
  })
  .withPrompts({ coffee: false })
  .then(function (dir) {
    // assert something about the stuff in `dir`
  });
```

If your generator calls `composeWith()`, you may want to mock those dependent generators. Using `#withGenerators()`, pass in array of arrays that use `#createDummyGenerator()` as the first item and a namespace for the mocked generator as a second item:

```
var deps = [
  [helpers.createDummyGenerator(), 'karma:app']
];
return helpers.run(path.join(__dirname, '../app')).withGenerators(deps);
```

If you hate promises, you can use the `'ready'`, `'error'`, and `'end'` Events emitted:

```
helpers.run(path.join(__dirname, '../app'))
  .on('error', function (error) {
    console.log('Oh Noes!', error);
  })
  .on('ready', function (generator) {
    // This is called right before `generator.run()` is called
  })
  .on('end', done);
```

You can also run a generator importing it as a module. This is usefull if the source code of your generator is transpiled.

You will need to provide the following settings to `run`:

- `resolved`: Path to the generator, e.g. `../src/app/index.js`
- `namespace`: Namespace of the generator, e.g. `mygenerator:app`

```
var MyGenerator = require('../src/app');

helpers.run(MyGenerator, {
  resolved: require.resolve(__dirname, '../src/app/index.js'),
  namespace: 'mygenerator:app'
});
```

Assertions helpers

Yeoman extends the native assert module (https://nodejs.org/api/assert.html) with generator related assertions helpers. You can see the full list of assertions helpers on the `yeoman-assert` repository (https://github.com/yeoman/yeoman-assert).

Require the assertion helpers:

```
var assert = require('yeoman-assert');
```

Assert files exists

```
assert.file(['Gruntfile.js', 'app/router.js', 'app/views/main.js']);
```

`assert.noFile()` assert the contrary.

Assert a file content

```
assert.fileContent('controllers/user.js', /App\.UserController = Ember\.Objec
```

`assert.noFileContent()` assert the contrary.