Testing Techniques in Node.js

Nicole Rauch and Andreas Leidig, SPA2014, Wednesday 2nd July, 2014 9:15am-12:00pm

Introduction

Software Craftsmanship Communities in Austria, Germany… See web site

Started to develop the site in March 2013 as a community platform with two main contributors (Nicole and Andreas). It has been live since August 2013. Layered Architecture – Node.js and mongoDB. 3-tier, stateless. No back-end notion of session, but the user ID and language are context for the front-end.

Workshop Mechanics

A problem will be presented that was encountered in developing the site. In pairs or threes, think about solutions and try out ideas in code. Finally present solution to the group. A virtual machine is provided that includes a running server configuration.

Testing Tools

Problem 1: App Structure

index.js invokes service.allMembers
membersService.js
memberStore.js
persistence.js
mongoDB

Node.js “require” promises to instantiate a singleton module. But: require creates one singleton PER IMPORT PATH. So if different modules are not in the same location, they could require the same module by different relative paths.

In production code (folder “lib”): require(‘./membersService’);
Test code (folder “test”): require(‘../../lib/membersService’);

Stubbing is therefore not so simple.

Nicole and Andreas discovered the proxyquire module, which is intended to resolve this.

var app = proxyquire(‘../../lib/members/*index.js*’, {‘./membersService’: membersServiceStub});

Next, they wanted to stub out the memberStore. This can be done by two proxyquire statements. In general, you can use this for any number of layers.

This approach really does not scale, particularly if there is fan-out to the different layers. How can we fix this?

Suggested approaches

- Wrap the real “require” in a custom “require” method to allow easy substitutions
- Create a symlink to make the paths the same
    - Might cause a problem in the build server, which is in a cloud server
- Normalise all the paths
- Service broker approach using symbolic names for each required module
- Pass all the required paths to the module under test as parameters
    - Doesn’t really scale

Selected Solution: Dependency Injection

The Cool Beans framework supplies this with minimal overhead. Just define all the singleton beans in a json file to plug into nconf. Then write:

```
function createConfiguration() {
        nconf.defaults({
                //…
                beans: new Beans(‘./config/beans.json’);
        });
        return nconf;
}

var service = require(‘nconf’).get(‘beans’).get(‘membersService’);
```

Suggestion from floor: could also look into wire.js.

With Cool Beans, you just write the following for a module you want to stub (and none of the modules dependent on it):

```
beforeEach(function () {
        sinon.stub(membersPersistence, ‘list’, function (sortOrder, callback) {
                callback(null, [testMember]);
        }
});
```

Sinon is another library, which specifically caters for singletons.

Problem 2: Application Configuration

How can we provide a standardised test configuration with minimal setup? Usually, production config differs from testing – logins, logging, privileges, paths…

A configuration file should define the IDs of users with superuser privileges. Use nconf.js (of course) to read it.

For testing, use

```
beforeEach (function(done) {
        nconf.set(‘superuser’, ‘Bobby’, function() {
                done();
        }
});
```

What can you do to improve this solution?

Suggestions

- Helper class to add a superuser ID to the configuration
- Configuration service

Model Solution: Helper

Helper function configureForTest.js can override configuration parameters that are needed for testing.

```
var nconf=require(‘nconf’);
nconf.overrides({ superuser: [‘Charlie’] });
module.exports = require(‘../configure’);
```

and in caller:

```
var beans = require(‘../../testutil/configureForTest’).get(‘beans’);
```

How can you guard against configuration errors in the test configuration? Review carefully. At least you can see easily what is different between production config and test config. You can also override configuration parameters for individual test suites or test cases (multiple layers).

Problem 3: Hide Your Resources

Combine CoolBeans with nconf to replace some beans with fake implementations. This ensures that, for example, a failing test cannot accidentally corrupt a database.

Warning: you can still code tests in a way that requires the production configuration without allowing the test overrides.

Problem 4: HTML Tests

How can we check that the pages are rendered correctly without starting a full-fledged server?

We want to test the rendered HTML, together with

- Relevant middleware (i18n and user privileges) work correctly

“Current user” is part of the session and is managed by a third-party module “passport”. Authentication is done by an OpenID provider (working offline, you have to provide your own). Test setup for a test of a vertical slice of the system therefore becomes very long-winded.

Model Solution: minimal express app with the required middleware

createApp takes a memberID parameter, which if supplied, means the express app acts as if that user is logged in. The user object is stubbed accordingly.

Benefits:

- Easily set up isolated parts of the application for testing.
- Rely on

(NB it turns out that the test passed unexpectedly when the path was changed in '/edit/hada' – Nicole worked out that the stubbed member class wasn't checking the user ID requested against the nickname of the logged-in member).

Problem 5: Database Tests

How to test a database integration, given that our test setup stubs the database? Some logic will work directly within the database – queries for all activities in a specific timeframe, for example.

Solution: a second test configuration in which we connect to a test database. See testbeansWithDB.json. The test database is rebuilt for each test run. Individual test cases can call configureForTestWithDB instead of configureForTest to use the test database.

Can provide unit tests to verify the database operations in isolation, if desired.

Problem 6: Database Race Conditions

How can you test contention between users? Two people try to edit the same information.

In the absence of transaction protection in mongoDB, the initial approach was to use the findAndModify operation to implement optimistic locking with version counters and automatic retries where feasible (e.g. register me for a specific activity). Users who lose are asked to redo their operation.

How can we test that there will be no data loss caused by race conditions? How can we test that the retry (if applicable) succeeds without data loss?

You have to somehow bring about an artificial race condition. Answer was to modify getById, which is called within the saveWithVersion method. So within the test setup, make a modification A and persist it, then within the test make a modification B and attempt to persist it (which should fail).

Retry is just a slight variation on that – the outcome should not be mod A but mod A + mod B.

Suggestion from the floor is to use CouchDB instead of mongoDB. Versioning is built in.

Problem 7: DOM testing

How can we test front-end code that requires a DOM, if all we have is jade template files in the back-end (but no HTML files)?
We want to unit-test the client-side jquery validations. These tests rely heavily on DOM elements.
We have no static HTML files to provide a DOM for testing.

Possible Approaches

Save the HTML from a browser and save it as a resource for the test? That goes out of date if the jade template changes. Also, you have to adjust all the paths to scripts loaded by the page.

Selected Solution: extract the form to a Jade mix-in

This can be included easily from other jade files.
Create a new jade file that includes all the form mixins.
Create a dummy file for all required res.locals.
Create a grunt target (gruntfile.js) which configures the build target for the forms. Its jade/compile target creates the HTML form from the jade template with the stubbed dependencies.

Resources

Softwerkskammer will be maintained as a living demo. Contributions and suggestions are invited, as is plagiarism.

Original code: https://github.com/softwerkskammer/Agora
Examples and slides: https://github.com/NicoleRauch/NodeTestingTechniques
Blog: http://pboop.wordpress.com/
Mail: nicole.rauch@msg-gillardon.de; andreas.leidig@msg-gillardon.de