**Unit Testing ui-router Configuration**

## 注意：使用中遇到的坑

#### 1.karma.config.js配置

a.在files中添加 “angular-ui-router.js”的路径

b.添加config文件的路径

c.添加 test 的文件的路径

#### 2. 将config独立到一个模块中

angular.module('someApp.routing', ['ui.router'])

.config(function($stateProvider, ...){

...

});

// app.js

angular.module('someApp', ['someApp.routing', ...])

其中，将ui.router的依赖放到config模块中

【参考文献】

1. <https://github.com/nikaspran/example-ui-router-testing（已经下载到本地>：E:\gitSource\jsSample\UiRouterUnitTesting）

**Why?**

Angular app configuration is code, so one could argue there is no real reason not to test it. If you misconfigure your application, it’s no less threatening than a bug in some controller or service.

End-to-end testing can (and should) be used to verify major functionality. It is, however, rather more difficult to write and maintain. In contrast, unit tests let you check the details at impressive speeds.

Unit testing the .config() blocks in your Angular app seems a daunting task at first. You can’t easily access the providers (tests run after the config phase is over) and there is usually a lot of stuff that gets bootstrapped along even for the tiniest of tests.

There is however a rather painless way of testing at least the routing of your app, which is usually where a lot of the important logic happens. In this post, we’ll be looking into how you might do unit-testing if you’re using [ui-router](https://github.com/angular-ui/ui-router). Your experience may vary with other libraries.

**Separate routing from the rest of your configuration**

The key concept in unit testing anything is making it independently testable and ui-router configuration is no exception.

To do this, you should first extract anything related to ui-router into a separate module, then require that in your main app module. It’s a simple refactoring that gives immediate benefits even without the tests - in my experience, ui-router can drown out everything else in the configuration phase. This makes it easy to see at a glance the high-level logic of your app, makes the rest of your modules easier to test (you don’t have to bootstrap the entire routing if you don’t want to).

I feel including ui-router as a dependency in the routing submodule is also wise as it keeps the implementation details hidden. In theory, you could replace ui-router with a different routing library in the future and there would be less to change. It would look something like this:

// routing.js

angular.module('someApp.routing', ['ui.router'])

.config(function($stateProvider, ...){

...

});

// app.js

angular.module('someApp', ['someApp.routing', ...])

...

**Helpful utilities**

Here’s handful of key utilities that we’ll be using to streamline our unit tests. You’ll find these scattered about in the examples below.

* mockTemplate(url, [template])

Since ui-router will by default try to retrieve your views, we use this tiny function to mock a template for a specific route.

function mockTemplate(templateRoute, tmpl) {

$templateCache.put(templateRoute, tmpl || templateRoute);

}

* goTo(url)

Literally takes you to the specified URL and then does a $digest. Simply to make tests look more readable.

function goTo(url) {

$location.url(url);

$rootScope.$digest();

}

* goFrom(url).toState(state, [params])

This is used mainly to check onEnter and onExit blocks, as you actually need to do the state transition. We prime the $location so that the ui-router does not immediately go somewhere different on$scope.$digest().

function goFrom(url) {

return {toState: function (state, params) {

$location.replace().url(url); //Don't actually trigger a reload

$state.go(state, params);

$rootScope.$digest();

}};

}

* resolve(value).forStateAndView(state, [view])

Resolve blocks are a bit tricky to test, so this is what I’ve been using personally. It’s a bit weird, but it essentially lets you execute the resolve as if you were ui-router. It uses $injector to get you the fully wired up version of the resolve result.

function resolve(value) {

return {forStateAndView: function (state, view) {

var viewDefinition = view ? $state.get(state).views[view] : $state.get(state);

return $injector.invoke(viewDefinition.resolve[value]);

}};

}

**URL routing**

One of the most important things to check is whether your routing matches your expected state transitions. This is where you test all your$urlRouterProvider .when(), .rule() and .otherwise() blocks. This is also where you can check whether your state url mappings are what you expect them to be.

Since actions speak louder than words, here’s an example. Say you have the following configuration:

...

.config(function ($urlRouterProvider, $stateProvider) {

$urlRouterProvider

.when('', '/home')

.when('/', '/home')

.otherwise(function ($injector) {

$injector.get('$state').go('404', {}, { location: false });

});

$stateProvider

.state('home', {

url: '/home',

templateUrl: 'views/home.html'

})

.state('404', {

templateUrl: 'views/404.html'

});

});

The test structure for URL routing would then look something like this:

describe('path', function () {

describe('when empty', function () {

beforeEach(mockTemplate.bind(null, 'views/home.html'));

it('should go to the home state', function () {

goTo('');

expect($state.current.name).toEqual('home');

});

});

describe('/', function () {

beforeEach(mockTemplate.bind(null, 'views/home.html'));

it('should go to the home state', function () {

goTo('/');

expect($state.current.name).toEqual('home');

});

});

describe('/home', function () {

beforeEach(mockTemplate.bind(null, 'views/home.html'));

it('should go to the home state', function () {

goTo('/home');

expect($state.current.name).toEqual('home');

});

});

describe('otherwise', function () {

beforeEach(mockTemplate.bind(null, 'views/404.html'));

it('should go to the 404 state', function () {

goTo('someNonExistentUrl');

expect($state.current.name).toEqual('404');

});

it('should not change the url', function () {

var badUrl = '/someNonExistentUrl';

goTo(badUrl);

expect($location.url()).toEqual(badUrl);

});

});

});

We use $state.current.name to check whether the transition happened as we had expected it to. Since we have access to the current state, we could do a more elaborate expectation here if necessary.

The tests themselves are arguably clear, concise and easy to read. I use the beforeEach statements liberally because they make it easy to add new tiny micro-tests for any other behavior I want to verify.

As a bonus, we implicitly check whether the correct view is being loaded. If this is unwanted, you could mock out $templateCache to always return whatever the argument was. I could see that being useful if your views have a ton of views that make it cumbersome to always mock them out explicitly.

You can use this pattern to test pretty much anything to do with URL mapping, including any state parameters, .rule() blocks and such. As they say, sky’s the limit here.

**Resolve**

People usually mock collaborators out in their controller tests. Ui-router makes this especially easy with the resolve mechanic. However, this often leaves a gaping hole in test coverage, where you verify whether the controller acts correctly given the correct dependencies BUT never check whether the correct dependencies are in fact given.

Here’s an example of how to retrieve and test the resolve blocks for a state:

$stateProvider

.state('stateWithoutViews', {

resolve: {

someModel: ['someRepository', function (someRepository) {

return someRepository.getModel();

}],

...

},

...

})

.state('stateWithViews', {

views: { 'main@layout': {

resolve: {

otherModel:['otherRepository', function (otherRepository) {

return otherRepository.getModel();

}]

},

...

}},

...

});

describe('state', function () {

describe('stateWithoutViews', function () {

it('should resolve someModel', function () {

var onResolved = sinon.spy(); // just a spy of any sort

mockSomeRepository.getModel.returns($q.when('something')); // just a mock for the service

resolve('someModel').forStateAndView('stateWithoutViews')().then(onResolved);

$rootScope.$digest();

expect(onResolved).toHaveBeenCalledWith('something');

});

});

describe('stateWithViews', function () {

it('should resolve otherModel', function () {

var onResolved = sinon.spy(); // just a spy of any sort

mockOtherRepository.getModel.returns($q.when('other')); // just a mock for the service

resolve('otherModel').forStateAndView('stateWithViews', 'main@layout')().then(onResolved);

$rootScope.$digest();

expect(onResolved).toHaveBeenCalledWith('other');

});

});

});

resolve(value).forStateAndView(state, [view]) is used to retrieve the resolved value, which in these cases were functions that returned promises.

As you can see, you can mock out any collaborators and test any scenarios as you wish. You can even add new $stateParams and check any permutations there.

**onEnter and onExit**

These are pretty much straightforward - just force a transition to the state you wish to test (and then out of it, for onExit). So, something like this:

describe('onEnter', function () {

it('should open a modal', function () {

goFrom('/modalState').toState('modal');

expect(mockModal.open).toHaveBeenCalled();

});

});

describe('onExit', function () {

it('should close the modal', function () {

var modal = {close: sinon.spy()};

mockModal.open.returns(modal);

goFrom('/modalState').toState('modal');

goFrom('/otherState').toState('other');

expect(modal.close).toHaveBeenCalled();

});

});

**$stateChange\* events**

Admittedly, these are a bit tricky to test. On the one hand, you could simply$emit the desired event, but then you’re bypassing ui-router completely, so there may be situations where that hides a bug somewhere.

On the other hand, you could just do a state transition (successful or failed) and then simply verify any side effects. The problem here is that you’re being a bit too indirect and the state transition itself might interfere with the properties you want to test. So, for example, to test a $stateChangeErrorhandler you might mock one of your state resolves to throw an error.

The exact situation depends on your use case, but either way, there are some drawbacks to either approach. At this point, I’m leaning towards the first one if you create a simple utility to hide the details of the event itself.

**Other notes**

* You can test state transition chains by listening in on the$stateChangeSuccess event:
* it('should visit multiple states', function(){
* var statesVisited = [];
* $rootScope.$on('$stateChangeSuccess', function (event, toState) {
* statesVisited.push(toState.name);
* });
* goTo('/someUrl');
* expect(statesVisited).toEqual(['state1', 'state2']);
* });
* Utilize beforeEach(). Your state configuration is usually a fairly complex tree structure. Correctly using these methods lets you defer common details closer to the describe() statement, so the preconditions are the same for any child tests. This keeps the test code itself clean and succinct.
* There are many more things you can test, such as whether the correct controllers are specified, state data fields, $stateParams parsing and such. Really, everything depends on your use case and what you deem most important. Pretty much everything about the ui-router configuration is accessible in one way or another, which makes testing it just a case of letting your imagination run wild.

**Summary**

You can and should unit test your routing configuration if you’re using ui-router. I’ve personally found a couple of hidden bugs in my apps while researching how to best approach this. The tests not only verify functionality is correct, but also serve as a high-level overview of your application routing.

By suggestion from [Fernando Araujo](http://nikas.praninskas.com/angular/2014/09/27/unit-testing-ui-router-configuration/#comment-2273936546) I’ve created an [example repository](https://github.com/nikaspran/example-ui-router-testing) that showcases everything in this post.