



Testing AngularJS



Jasmine and Karma (Part 1)

Our Goal

In this tutorial we will be building and testing an employee directory for a fictional company. This directory will have a view to show all of our users along with another view to serve as a profile page for individual users. Within this part of the tutorial we'll focus on building the service and its tests that will be used for these two views.

MeetIrl

Joe

 Developer
 Atlanta
 joedirt

Jim

 Designer
 Los Angeles
 jimbo

Jane

 Designer
 New York
 gijane

What You Should Know

The primary focus for this tutorial is testing so my assumption is that you're comfortable working with JavaScript and AngularJS applications. As a result of this I won't be taking the time to explain what a factory is and how it's used. Instead, I'll provide you with code as we work through our tests.

For now, we'll start with hard-coded data so we can get to writing tests ASAP. In future tutorials within this series we'll add an actual API to get as "real world" as possible.

Why Test?

From personal experience, tests are the best way to prevent software defects. I've been on many teams in the past where a small piece of code is updated and the developer manually opens their browser or Postman to verify that it still works. This approach (manual QA) is begging for a disaster.

| Tests are the best way to prevent software defects.

As features and codebases grow, manual QA becomes more expensive, time consuming, and error prone. If a feature or function is removed does every developer remember all of its potential side-effects? Are all developers manually testing in the same way? Probably not.

The reason we test our code is to verify that it behaves as we expect it to. As a result of this process you'll find you have better feature documentation for yourself and other developers as well as a design aid for your APIs.

Why Karma?

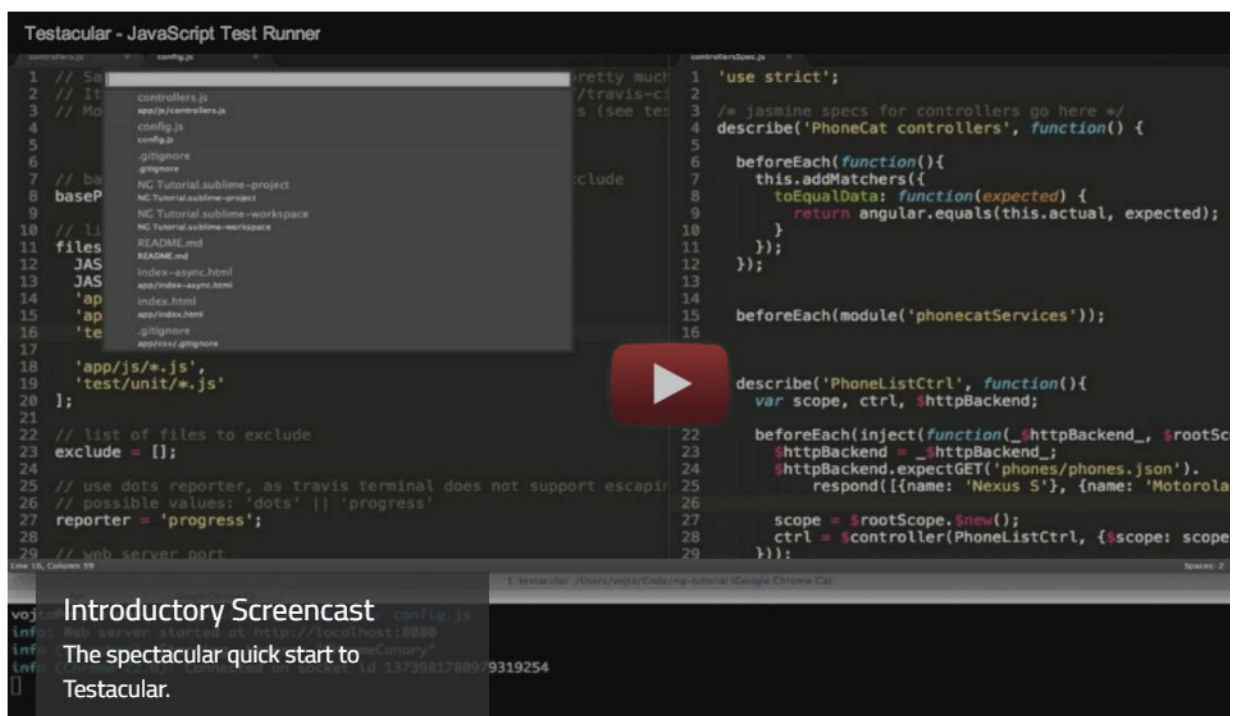


intro config plus dev about

On the [AngularJS](#) team, we rely on testing and we always seek better tools to make our life easier. That's why we created Karma - a test runner that fits all our needs.

[View project on GitHub](#)

[npm install karma](#)



Karma is a direct product of the [AngularJS](#) team from struggling to test their own framework features with existing tools. As a result of this, they made Karma and rightly suggest it as their preferred test runner within the AngularJS [documentation](#).

In addition to playing nicely with Angular, it also provides flexibility for you to tailor Karma to your workflow. This includes the option to test your code on various browsers and devices such as phones, tablets, and even a [PS3](#) like the YouTube team.

Karma also provides you options to replace Jasmine with other testing frameworks such as [Mocha](#) and [QUnit](#) or integrate with various continuous integration services like [Jenkins](#), [TravisCI](#), or [CircleCI](#).

Aside from the initial setup and configuration your typical interaction with Karma will be to run `karma start` in a terminal window.

Why Jasmine



Jasmine

Behavior-Driven JavaScript

Documentation:

[1.3](#)

[2.0](#)

[2.1](#)

[2.2](#)

[2.3](#)

[2.4](#)

[Edge](#)

Jasmine is a behavior-driven development framework for testing JavaScript code that plays very well with Karma. Similar to Karma, it's also the recommended testing framework within the AngularJS [documentation](#). Jasmine is also dependency free and doesn't require a DOM.

As far as features go, I love that Jasmine has almost everything I need for testing built into it. The most notable example would be [spies](#). A spy allows us to "spy" on a function and track attributes about it such as whether or not it was called, how many times it was called, and with which arguments it was called. With a framework like Mocha, spies are not built-in and would require pairing it with a separate library like [Sinon.js](#).

The good news is that the switching costs between testing frameworks is relatively low with differences in syntax as small as Jasmine's `toEqual()` and Mocha's `to.equal()`.

A Simple Test Example

Imagine you had an alien servant named Adder who follows you everywhere you go. Other than being a cute alien companion Adder can really only do one thing, add two numbers together.

To verify Adder's ability to add two numbers we could generate a set of test cases to see if Adder provides us the correct answer. So we could provide Adder with two positive numbers (2, 4), a positive number and a zero (3, 0), a positive number and a negative number (5, -2), and so on.

Our tests aren't concerned with *how* Adder arrives at the answer. We only care about the answer Adder provides us. In other words, we only care that Adder *behaves as expected* - we have no concern for Adder's implementation.

AngularJS Application Setup

Before we get started verify you have [npm](#) installed with version 5 of node. If you already have [node](#) and [nodemon](#) installed, you can skip this step.

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.1/install.sh |  
bash  
nvm install 5.0  
nvm use 5.0  
npm install -g nodemon
```

From here we can create the project directory and initial files we'll need to start testing our code. If you'd prefer to follow along the code within this tutorial can be found on [Github](#).

```
mkdir meet-irl && cd meet-irl  
touch server.js  
mkdir app && cd app  
touch index.html app.js  
app.css
```

At this point your project structure should look like this:

```
|--meet-irl
  |--app
    |--app.css
    |--app.js
    |--
  index.html
  |--server.js
```

We're going to use Express as our server which will, within this tutorial, serve up our Angular application. Back in the root directory of this project let's create a `package.json` for our project.

```
npm init
```

I left default values for everything *except* the entry point which I set to the file we just created, `server.js`.

```
⇒ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (test)
version: (1.0.0)
description:
entry point: (index.js) server.js
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/adammorgan/Desktop/test/package.json:

{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes)
```

From here we can install the packages we'll need.

```
npm install express body-parser morgan path --
save
```

Now that our dependencies are installed we can add this code to our `server.js` file.

```

var express = require('express'),
    app = express(),
    bodyParser = require('body-parser'),
    morgan = require('morgan'),
    path = require('path');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

app.use(function(req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST');
  res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With, content-type, Authorization');
  next();
});

app.use(morgan('dev'));

app.use(express.static(__dirname + '/app'));

app.get('*', function(req, res) {
  res.sendFile(path.join(__dirname + '/index.html'));
});

app.listen(8080);
console.log('meet-irl is running on 8080');

```

This sends our app to `index.html` within our `app` directory so let's add some content to that file as well.

```

<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="description" content="">
  <meta name="author" content="">

  <base href="/">

  <title>MeetIrl</title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7" crossorigin="anonymous">
  <link rel="stylesheet" href="app.css">

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.5/angular.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.2.18/angular-ui-router.js"></script>

  <script src="app.js"></script>
</head>

<body ng-app="meetIrl">
  <nav class="navbar navbar-default">
    <div class="container">
      <div class="navbar-header">
        <a class="navbar-brand">MeetIrl</a>
      </div>
    </div>
  </nav>

  <div ui-view>Hello, world!</div>
</body>
</html>

```

Within this file I've included CDN references to Bootstrap for styling, Angular, ui-router, and the `app.js` and `app.css` files we made earlier. Within the body we declare our `ng-app` so let's add that to `app.js`.

```

(function() {
  'use strict';

  angular.module('meetIrl', [
    'ui.router'
  ])
  .config(function($urlRouterProvider) {
    $urlRouterProvider.otherwise("/");
  });
})();

```

With that change to `app.js` running `nodemon server.js` should serve a barebones Angular application to `localhost:8080` with a navbar and the usual message: "Hello, world!".

Karma Setup

Before we start building out the Angular components of our application, we first need to install and setup [Karma](#) which is what we're going to use to run our tests.

Back in the root directory of the app we'll need to install a few more packages. First we'll grab Karma and all of its associated plugins.

```
npm install karma karma-jasmine jasmine-core karma-chrome-launcher --save-dev
```

While we're at it let's grab the Karma CLI as well.

```
npm install -g karma-cli
```

Finally we'll install Angular, ui-router, and angular-mocks since we'll have to provide these as dependencies to Karma. This will be explained in more detail shortly.

```
npm install angular angular-ui-router angular-mocks --save-dev
```

At this point we're ready to create our Karma configuration file so that we can start writing tests. If you've installed the Karma CLI you can run `karma init`.

Similar to `npm init` you'll see a series of questions that help you setup your configuration file. The questions are listed below along with the options you should provide. If there's nothing to the right of an `*` just hit your `return` key.

```
⇒ karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*.Spec.js".
Enter empty string to move to the next question.
>

Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> yes

Config file generated at "/Users/adammorgan/Desktop/meet-irl/karma.conf.js".
```

Once you're done you should see a message alerting you of your newly created file. Open the file and you should see something like this. (Note: Comments removed for brevity)

```

module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
    ],
    exclude: [
    ],
    preprocessors: {
    },
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false,
    concurrency: Infinity
  })
}

```

Our First Test

Our first test is going to test the service we're going to use for managing our users. For now, we're going to have one service method that returns a hard-coded list of users to mock an actual API.

Eventually we will integrate an actual API into our service but for now we're going to keep things as simple as possible. This style of development also mimicks the real world if you're working with backend developers who may be building their part of an application at the same time as the frontend.

First, let's create a new directory for our services along with another directory specifically for the `Users` service and its associated test file.

```

cd app
mkdir services && cd
services
mkdir users && cd users
touch users.js users.spec.js

```

With these additions your project structure should now look like this:

```

|-meet-irl
  |-app
    |-services
      |-users
        |-users.js
        |-
  users.spec.js
  |-app.css
  |-app.js
  |-index.html
  |-server.js

```

The `users.js` file will be used for our service and the `users.spec.js` file will be used to test the service. Open `users.spec.js` and we can start writing our first test.

When we're using Jasmine to test our code we group our tests together with what Jasmine calls a **"test suite"**. We begin our test suite by calling Jasmine's global `describe` function. This function takes two parameters: a string and a function. The string serves as a title and the function is the code that implements our tests.

We can setup our first test suite by adding the following code to `users.spec.js`.

```

describe('Users factory', function()
{
});

```

Within this `describe` block we'll add specs. Specs look similar to suites since they take a string and a function as arguments but rather than call `describe` we're going to call `it` instead. Within our `it` block is where we put our expectations that tests our code.

An example spec added to our `describe` block would look like this:

```

describe('Users factory', function() {
  it('has a dummy spec to test 2 + 2', function()
  {
    expect().toEqual(4);
  });
});

```

Similar to our `describe` block, we've provided a brief summary of our test in the form of a string. In this case we're doing a sample test, basic addition, to illustrate the structure of a test.

Within the function we call `expect` and provide it what is referred to as an **"actual"**. We've left it empty for now but we'll update it momentarily. After the `expect` is the chained **"Matcher"** function, `toEqual()`, which the testing developer provides with the *expected output of the code being tested*.

This is enough to do the "Hello, world" equivalent of testing so let's do that. But before we run Karma, we'll have to add this file to our Karma configuration file. So open `karma.conf.js` and update the `files` property with our new test file.

```

module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      './app/services/users/users.spec.js'
    ],
    exclude: [
    ],
    ...
  })
}

```

Save the file, run `karma start`, and you should see Karma display an error message for your test.

```

⇒ karma start
28 05 2016 16:56:23.310:WARN [karma]: No captured browser, open http://localhost:9876/
28 05 2016 16:56:23.320:INFO [karma]: Karma v0.13.22 server started at http://localhost:9876/
28 05 2016 16:56:23.324:INFO [launcher]: Starting browser Chrome
28 05 2016 16:56:24.572:INFO [Chrome 50.0.2661 (Mac OS X 10.11.4)]: Connected on socket /#96z
Chrome 50.0.2661 (Mac OS X 10.11.4) Users factory has a dummy spec to test 2 + 2 FAILED
    Expected undefined to equal 4.
    at Object.<anonymous> (/Users/adammorgan/Desktop/meet-irl/test.spec.js:3:14)
Chrome 50.0.2661 (Mac OS X 10.11.4): Executed 1 of 1 (1 FAILED) ERROR (0.013 secs / 0.006 sec)

```

In our `karma.conf.js` file we have `singleRun` set to `false` so Karma should automatically restart any time you save changes to your tests or code. Leave this running in a separate terminal window and you won't have to run `karma start` everytime we write or update a test.

Since we didn't pass anything into `expect()` it evaluated to `undefined`. Since `undefined` isn't equal to `4` our test failed. So far so good. Now let's make it pass with this small bit of code.

```
expect(2 + 2).toEqual(4);
```

Run `karma start` again if it's not already running and you should now see a success message. Well done Javascript addition operator.

```

⇒ karma start
28 05 2016 16:58:41.883:WARN [karma]: No captured browser, open http://localhost:9876/
28 05 2016 16:58:41.892:INFO [karma]: Karma v0.13.22 server started at http://localhost:9876/
28 05 2016 16:58:41.897:INFO [launcher]: Starting browser Chrome
28 05 2016 16:58:43.145:INFO [Chrome 50.0.2661 (Mac OS X 10.11.4)]: Connected on socket /#9tf
Chrome 50.0.2661 (Mac OS X 10.11.4): Executed 1 of 1 SUCCESS (0.009 secs / 0.002 secs)

```

Testing an Angular Factory

We wrote a failing test and made it pass but that was basic JavaScript. We're here to test Angular code so let's do it. But first, let's think about the smallest bit of a working factory we could test.

A factory is typically a JavaScript object with properties that evaluate to function expressions. The simplest factory we could possibly write would be a factory that returns an empty object. In other words, it's a defined factory with no available methods.

First we'll write the test for this behavior and then we'll write the factory to make the test pass.

As of right now your `users.spec.js` file should look like this:

```

describe('Users factory', function() {
  it('has a dummy spec to test 2 + 2', function() {
    {
      expect(2 + 2).toEqual(4);
    }
  });
});

```

Remove the 3-line `it` block and update it to this:


```
describe('Users factory', function() {
  var Users;

  beforeEach(angular.mock.module('api.users'));

  beforeEach(inject(function(_Users_) {
    Users = _Users_;
  }));

  it('should exist', function() {
    expect(Users).toBeDefined();
  });
});
```

There's some new, weird looking code here so let's break this down.

First I've created a variable `Users` which will be set to the factory we're eventually going to write.

Then there's `beforeEach(angular.mock.module('api.users'))`; `beforeEach()` is a function provided by Jasmine that allows us to run code before each test we've written is executed. `angular.mock.module()` is a function provided to us by `angular-mocks` which we installed earlier. We've specified our module here, `api.users`, so that it's available for us to use in our tests.

After that is another `beforeEach` block with `inject` being used to access the service we want to test - `Users`.

So in plain English we're saying, "Before each of my tests 1) load the module `api.users` and 2) inject the `Users` service (wrapped with underscores) and set it to the `Users` variable I defined locally."

The syntax here with our service name wrapped in underscores, `_Users_`, is a convenient yet optional trick that's commonly used so we can assign our service to the local `Users` variable we created earlier. You could just write `inject(function(Users))` and assign it to the local `Users` variable but I think the surrounding underscores help to distinguish between the two and it's more in line with conventions within the Angular community.

Finally, there's a test spec that expects our `Users` service to be defined. We'll write the actual service methods once this initial test is passing!

If you save this file you should see an error message stating `angular is not defined`.

The reason this is happening is because Karma only has access to one file - `users.spec.js`. That was fine earlier when we were testing basic JavaScript like `2 + 2` but now we're testing Angular and Karma needs Angular to test it. So open up `karma.conf.js` and update the `files` property from this:

```
frameworks: ['jasmine'],
files: [
  './app/services/users/users.spec.js'
],
exclude: [],
```

To this:

```
frameworks: ['jasmine'],
files: [
  './node_modules/angular/angular.js',

  './node_modules/angular-ui-router/release/angular-ui-router.js',
  './node_modules/angular-mocks/angular-mocks.js',
  './app/services/users/users.js',
  './app/app.js',
  './app/services/users/users.spec.js'
],
exclude: [],
```

In addition to our test file we're now including `angular.js` and its other dependency in our application `angular-ui-router`. We've also included `angular-mocks`, the module that provides us the ability to load in our Angular modules to test. Finally, we've added our `app.js` which initializes our angular app.

Save this, run `karma start` again, and now you should see an error stating module `api.users` cannot be found. All that's left now is to create that module and its respective service.

The order of our files within the `files` property is important since it determines the order in which they're loaded into the browser when Karma is run. That's why Angular and all of its related code are placed up top, then the application files, and finally the test files.

Back in `app/services/users` open `users.js` and add the following code to create our `Users` factory which returns an empty object.

```

(function() {
  'use strict';

  angular.module('api.users', [])
    .factory('Users', function() {
      var Users = {};

      return Users;
    });
})();

```

Save that file and you should now see a passing test!

Adding Users to the Users Service

Now that we know our test is successfully testing the existence of our `Users` service let's add an `all` method that will return a collection of users. This method will eventually be used to display the list of users shown in the image at the start of this tutorial.

Continuing with the progressive additions to our code, jump back into `users.spec.js` and add a new suite and spec for our new service method.

```

describe('Users factory', function() {
  var Users;

  beforeEach(angular.mock.module('api.users'));

  beforeEach(inject(function(_Users_) {
    Users = _Users_;
  }));

  it('should exist', function() {
    expect(Users).toBeDefined();
  });

  describe('.all()', function() {

    it('should exist', function() {
      expect(Users.all).toBeDefined();
    });
  });
});

```

Save this and you should see a failing test for the `Users.all` method because it doesn't exist yet. Jump back into `users.js` and add the new method.

```

(function() {
  'use strict';

  angular.module('api.users', [])
    .factory('Users', function() {
      var Users = {};

      Users.all = function() {

      };

      return Users;
    });
})();

```

Save that and our previously failing test should now be passing. The test says nothing about what `all` does, it only expects that the method is defined. We want it to return a list of users we can use to populate a view in our app, so let's add another spec to handle that.

First, we'll need to add a local variable for our hard-coded set of users.

```
describe('Users factory', function()
{
  var Users;

  var userList = [
    {
      id: '1',
      name: 'Jane',
      role: 'Designer',
      location: 'New York',
      twitter: 'gijane'
    },
    {
      id: '2',
      name: 'Bob',
      role: 'Developer',
      location: 'New York',
      twitter: 'billybob'
    },
    {
      id: '3',
      name: 'Jim',
      role: 'Developer',
      location: 'Chicago',
      twitter: 'jimbo'
    },
    {
      id: '4',
      name: 'Bill',
      role: 'Designer',
      location: 'LA',
      twitter: 'dabill'
    }
  ];
  ...
});
```

Then we'll add another spec to test the expected behavior of our method right below our previous test.

```
...

describe('.all()', function() {

  it('should exist', function() {
    expect(Users.all).toBeDefined();
  });

  it('should return a hard-coded list of users', function()
  {
    expect(Users.all()).toEqual(userList);
  });
});
```

Now that this test has been written we should have one failing test. The test for our `Users` factory is expecting a list of users when we call `.all()` but our service doesn't return anything. So update `users.js` to return the same array of users ...

```

...
var Users = {};
var userList = [
  {
    id: '1',
    name: 'Jane',
    role: 'Designer',
    location: 'New
York',
    twitter: 'gijane'
  },
  {
    id: '2',
    name: 'Bob',
    role: 'Developer',
    location: 'New
York',
    twitter: 'billybob'
  },
  {
    id: '3',
    name: 'Jim',
    role: 'Developer',
    location: 'Chicago',
    twitter: 'jimbo'
  },
  {
    id: '4',
    name: 'Bill',
    role: 'Designer',
    location: 'LA',
    twitter: 'dabill'
  }
];

Users.all = function() {
  return userList;
};

return Users;
...

```

Save the file and now the test should pass!

It may seem a little weird that our test was failing and we simply copied the user array from the test into our service but that's a reflection of our test's *expectations*. It doesn't expect "real" data or data which must be served from an API. It simply states that when `User.all()` is called we expect its return value to equal the hard-coded value we've supplied.

Assume our four users above are actually the founding members of a company. The requirement is that all four members should be displayed on the "Founders" section of their website.

If a developer added another user to `userList` in `users.js` the test would no longer pass. Our test expects the four founding members we provided via hard-coding in `users.spec.js`. This addition in our service alters the *expected behavior* we specified in our test cases so the test fails as it should.

At this point the expectations specified in the tests would signal one of two things to the developer:

1. Their code change was in the wrong section of the app. This is only for founding members so perhaps there's an employee section with an employee array somewhere else in the codebase.
2. A new founding member was added to the team. This would essentially be a change in requirements. At this point the user service and test would both change to reflect the new requirement and our new *expected behavior* of the service.

└─ This is why developers often refer to test cases and their expectations for how our code should behave as *documentation*, or a *design aid*, for their actual code.

Finding a Single User

Now that we have a service method to return a list of *all* our users let's add one extra service method to find a specific employee by their `id`.

First, we'll add a new test for our new service method. Like the previous service method we wrote, we'll progressively build towards a completed feature. In `users.spec.js` add a new `describe` block below the one we created for `Users.all`.

```

describe('.all()', function() {
  ...
});

describe('.findById()', function() {
  it('should exist', function() {
    expect(Users.findById()).toBeDefined();
  });
});

```

We can make this pass rather easily as we did before. In `users.js` add the new service method.

```

    Users.all = function() {
      return userList;
    };

    Users.findById = function(id)
  {

    };

    return Users;
  }

```

Then we'll add another spec where we actually call our method and provide an expectation.

```

describe('.findById(id)', function() {

  it('should exist', function() {
    expect(Users.findById).toBeDefined();
  });

  it('should return one user object if it exists', function()
  {
    expect(Users.findById('2')).toEqual(singleUser);
  });
});

```

Notice we've referenced `singleUser` in the matcher function. In this case that would be the user with `id: '2'`. So just below the previous variable we hard-coded, `userList`, add another variable for our single user object.

```

describe('Users factory', function()
{
  var Users;

  var userList = [
    ...
  ];

  var singleUser = {
    id: '2',
    name: 'Bob',
    role: 'Developer',
    location: 'New York',
    twitter: 'billybob'
  };
}

```

We have yet to implement `findById` so our test is currently failing. Add this piece of code, or your preferred implementation that returns a single object, and the test should now pass.

```

    Users.findById = function(id) {

      return userList.find(function(user)
  {
    return user.id === id;
  });

    };

    return Users;
  }

```

There's only one thing left to do now to wrap up the service and tests we'll need to implement the rest of our employee directory. Eventually when we create the profile page for each employee we're going to find a user based on the `id` within the url. We've already verified our method can return a user if it exists but in the case of a user that cannot be found, we'll redirect them to a 404 page with a link back to our user directory.

So for the purpose of our `findById` method, we expect a user object to be returned if it exists. Otherwise, it'll evaluate to `undefined` and our controller, which we'll build in the next part of this tutorial, will redirect the user to our to-be-created 404 page.

With that in mind open up `users.spec.js` one more time and add this test to the `describe` block for `findById()`.

```

describe('.findById(id)', function() {

  it('should exist', function() {
    expect(Users.findById).toBeDefined();
  });

  it('should return one user object if it exists', function() {
    expect(Users.findById('2')).toEqual(singleUser);
  });

  it('should return undefined if the user cannot be found', function()
  {
    expect(Users.findById('ABC')).not.toBeDefined();
  });
});

```

This test should automatically pass as a side effect of using `Array.find()` which you can read about [here](#).

Conclusion

At this point I hope testing, both its benefits and the reason for writing them, is starting to become clear. Personally, I pushed off testing for the longest time and my reasons were primarily because I didn't understand the why behind them and resources for setting it up were limited.

What we've created in this tutorial isn't visually impressive but it's a step in the right direction. Now that a service is working and tested, all we have to do now is add our controllers and their associated tests and then we can populate our views to mimic the screenshot I posted at the start.

If you'd like your tests to display in your terminal window in a more readable fashion install this npm package.

```
npm install karma-spec-reporter --save-dev
```

Then in your `karma.conf.js` change the value for `reporters` from `progress` to `spec`.

```
module.exports = function(config) {
  config.set({
    ...
    exclude: [
    ],
    preprocessors: [
    ],
    reporters: ['spec'],
    ...
  })
}
```

With that change the output from Karma should be much easier to digest.

```
⇒ karma start
29 05 2016 11:35:04.585:WARN [karma]: No captured browser, open http://localhost:9876/
29 05 2016 11:35:04.595:INFO [karma]: Karma v0.13.22 server started at http://localhost:9876/
29 05 2016 11:35:04.600:INFO [launcher]: Starting browser Chrome
29 05 2016 11:35:05.803:INFO [Chrome 50.0.2661 (Mac OS X 10.11.4)]: Connected on socket /#rf1

Users factory
  ✓ should exist
  .all()
    ✓ should exist
    ✓ should return a hard-coded list of users
  .findById()
    ✓ should exist
    ✓ should return one user object if it exists
    ✓ should return undefined if the user cannot be found

Chrome 50.0.2661 (Mac OS X 10.11.4): Executed 6 of 6 SUCCESS (0.049 secs / 0.037 secs)
TOTAL: 6 SUCCESS
```