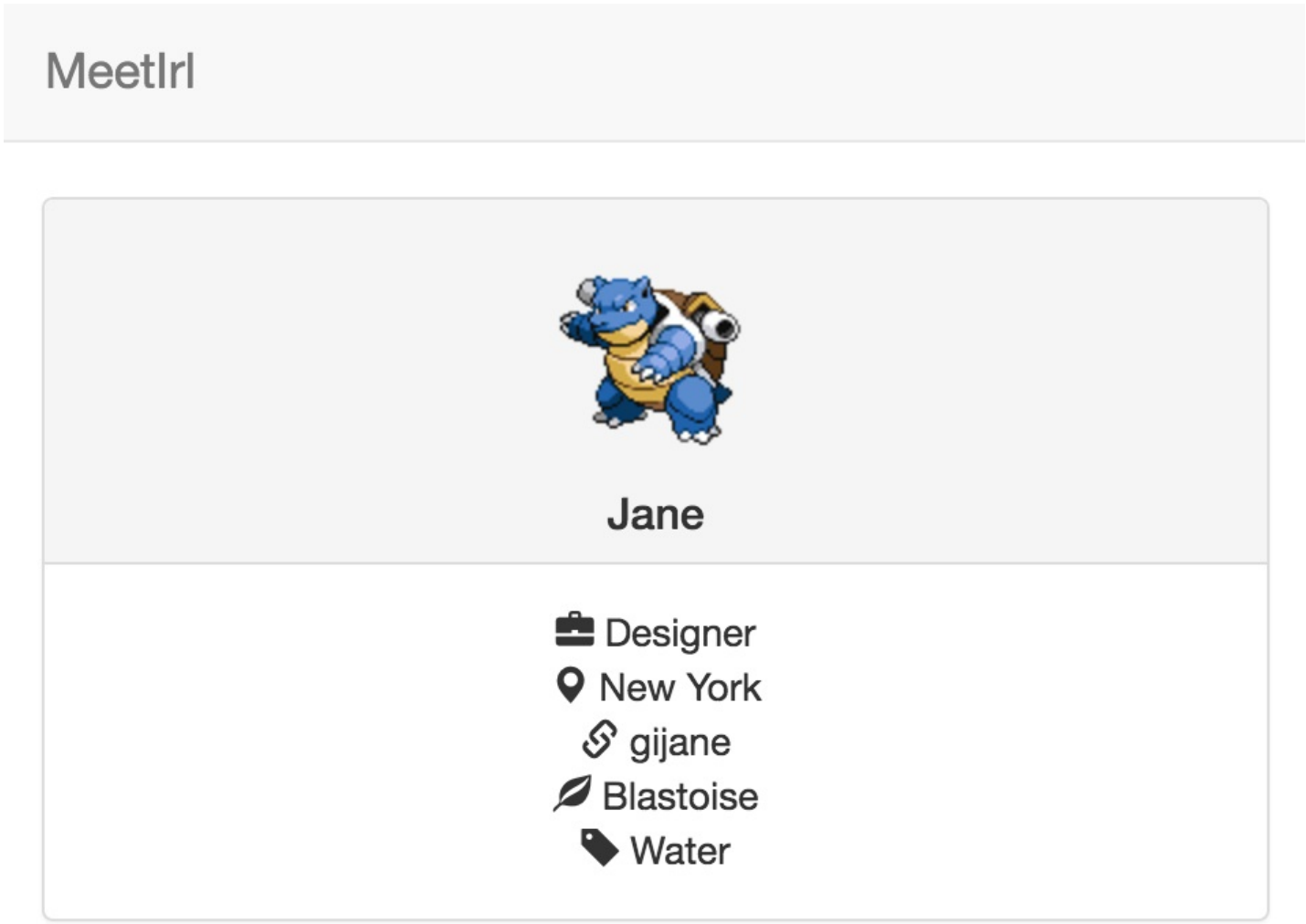
Testing AngularJS with Jasmine and Karma (Part 2)

scotch.io/tutorials/testing-angularjs-with-jasmine-and-karma-part-2

Our Goal

In this tutorial we will be creating and testing the user profile page for the employee directory we started building in Part 1 of this tutorial. The user profile page will show the details of each employee. Due to the recent comeback of Pokémon, thanks to Pokémon Go, our employee's have requested that their profile pages display an image of their favorite Pokémon as well. Thankfully for us, this provides us the opportunity to write tests for hitting a real API. We'll also write our own custom filter for our user profile page and test the filter as well. By the end of this tutorial you will have the ability to view user profiles that make HTTP requests to Pokéapi.



Related Course: Getting Started with Angular v2+

What You Should Know

Like the previous tutorial, this one will be focused on testing our controllers, factories, and filters that will be used for our user profile page so my assumption is that you're comfortable working with JavaScript and AngularJS applications. We'll be continuing with the application that was created in the first part of this tutorial so if you haven't worked your way through that yet, I'd recommend completing that first or cloning this repository which is the end result of the tutorial.

Testing an Angular Controller

In Part 1 of this tutorial , we created and tested a `Users` service but it isn't being used in our application just yet. Let's create a controller and view to display all of our users and write a test for this controller as well. Within the `app` directory of our application let's create a new `components` directory. Within this, create a `users` directory which will contain our view template, controller, and test file for our controller.

```
cd app
mkdir components && cd components
mkdir users && cd users
touch users.js users.spec.js users.html
```

At this point your project structure should now look like this:

```
|-meet-irl
  |-app
    |-components
      |-users
        |-users.js
        |-
users.spec.js
        |-users.html
    |-services
      |-users
        |-users.js
        |-
users.spec.js
    |-app.css
    |-app.js
    |-index.html
  |-karma.conf.js
  |-server.js
```

Our expectation for this view is that it will display all of our users that we defined in our `Users` service. So within the controller we're going to make a call to `Users.all` and set that to our controller's view-model object. From there we can use the `ng-repeat` directive to iterate over that list of users and display it in our view.

Before we test that functionality, let's first write a basic test for the existence of this controller in `components/users/users.spec.js`.

```
describe('UsersController', function() {
  var $controller, UsersController;


  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('components.users'));


  beforeEach(inject(function(_$controller_) {
    $controller = _$controller_;
    UsersController = $controller('UsersController',
{});
  }));


  it('should be defined', function() {
    expect(UsersController).toBeDefined();
  });
});
```

First we create two variables: `$controller` and `UsersController`. `$controller` will be set to Angular's built-in controller service and `UsersController` will be set to the actual instance of our controller we will write.

After that, we use `angular-mocks` to specify which modules we'll need within this test file. In this case we'll need `ui.router` and `components.users` which we'll create to make this test pass. The need for `ui.router` will be seen shortly when we create our controller since we specify all of its state options within the same file.

Then we create another `beforeEach` block with `inject` which is used to inject the AngularJS `$controller` service. We set `_$controller_` to the `$controller` variable we created and then create an instance of our controller by calling `$controller('UsersController', {})`. The first argument is the name of the controller we want to test and the second argument is an object of the dependencies for our controller. We'll leave it empty for

now since we're trying to keep this test as simple as possible.

Finally, we end this file with a basic test for the existence of our controller with the expectation that it should be defined.

> *The one line of code* `$controller = _$controller_;` *may seem unnecessary here when we could simply write* `UsersController = _$controller_('UsersController', {});`*. That would be completely valid in this specific case but in some of our later tests we'll need to instantiate controllers with different dependencies and that* `$controller` *variable will be needed. This will make more sense once we get to those tests.*

With that test file written update your `karma.conf.js` file to include our new test file within the `files` property along with the file for our controller which we're about to define.

```
  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/components/users/users.js',
    './app/app.js',
    './app/services/users/users.spec.js',
    './app/components/users/users.spec.js'
  ],
```

Restart Karma and you should now see a failing test stating our module `components.users` cannot be found. Let's create that and get this test to pass.

```
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 7 of 7 (1 FAILED) (0.069 secs / 0.053 secs)
TOTAL: 1 FAILED, 6 SUCCESS


1) should be defined
     UsersController
     Error: [$injector:modulerr] Failed to instantiate module components.users due to:
Error: [$injector:nomod] Module 'components.users' is not available! You either misspelled the module name or forgot t
o load it. If registering a module ensure that you specify the dependencies as the second argument.
```

Open up `components/users/users.js` and add the following code.

```
(function() {
  'use strict';


  angular.module('components.users', [])
  .controller('UsersController', function() {
    var vm = this;
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('users', {
        url: '/users',
        templateUrl:
'components/users/users.html',
        controller: 'UsersController as uc'
      });
  });
})();
```

Here we've declared our component `components.users` and the controller itself `UsersController` as we specified in our test file. In addition to this, we've also added a configuration for this file including its state, url, template, and controller. This configuration is why we included `ui.router` in our test file.

Save that file, restart Karma if it isn't already running, and you should now see a passing test for `UsersController should be defined` .

Now that we know our test is working at the most basic level we need to test the call to our service to get a list of users so we can populate our view. Open up `/components/users/users.spec.js` again and update it with another test.

```
describe('UsersController', function() {
  var $controller, UsersController, UsersFactory;


  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' }
  ];

  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('components.users'));

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

  beforeEach(inject(function(_$controller_, _Users_) {
    $controller = _$controller_;
    UsersFactory = _Users_;


    spyOn(UsersFactory, 'all').and.callFake(function() {
      return userList;
    });


    UsersController = $controller('UsersController', { Users: UsersFactory });
  }));

  it('should be defined', function() {
    expect(UsersController).toBeDefined();
  });


  it('should initialize with a call to Users.all()', function() {
    expect(UsersFactory.all).toHaveBeenCalled();
    expect(UsersController.users).toEqual(userList);
  });
});
```

Starting at the top we've added another variable `UsersFactory` which we'll set to our injected `Users` service. After that, we've added an array of users which we borrowed from the `Users` service from Part 1 of this tutorial which can be found in `/services/users/users.js`. Then, we load the module `api.users` using `angular-mocks`. In the following `beforeEach` block we inject our service `Users` using the underscore wrapping convention and set it to our local `UsersFactory` variable.

After that we add a spy to the `all` method of our factory and chain it with another one of Jasmine's built-in functions `callFake`. The `callFake` method allows us to intercept a call to that method and supply it our own return value. In this case, we're returning `userList` which we defined at the top of this file. Finally, we add our service as a dependency to `UsersController` when we call `$controller`. The property value `Users` refers to the service we'll inject into our actual controller and the value `UsersFactory` is a reference to the service we injected just two lines above it.

> *It's important to remember that our tests are testing expectations and not the actual implementation of our code. In this test file, we use Jasmine's `callFake` function to intercept the actual call and return a*

> *hardcoded list of users (our expectation). Our tests for that service don't belong here. It was already handled in Part 1 of this tutorial and the test for that method is located within* `/services/users/users.spec.js`.

Finally, we add a test spec for our controller with a new expectation: `should initialize with a call to Users.all()`. The test has two expectations. The first expectation uses the spy we declared above and simply expects that a call to the `all` method will be made. The second expectation expects that the controller's view-model property `users` will be set to the list of users we defined above. Save the file so Karma shows a failing test and let's add the real code to our controller which should help clarify our test.

Open up our controller file `/components/users/users.js` and update it to make our failing tests pass.

```
(function() {
  'use strict';

  angular.module('components.users', [])
  .controller('UsersController', function(Users) {
    var vm = this;


    vm.users = Users.all();
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('users', {
        url: '/users',
        templateUrl:
'components/users/users.html',
        controller: 'UsersController as uc'
      });
  });
})();
```

We've added the `Users` service as a dependency to our controller and also initialize a call to `Users.all` and set the return value to `vm.users`. Save that file and our previously failing test should now pass.

We've now created and tested the controller for our users and our users are waiting to be displayed in the browser. Open up our empty template `/components/users/users.html` and add the following code to iterate over our users in the `UsersController`.

```
<div class="container">
  <div class="row">
    <div class="col-md-4" ng-repeat="user in uc.users">
      <div class="panel panel-default">
        <div class="panel-heading">
          <h3 class="panel-title text-center">{{user.name}}</h3>
        </div>
        <div class="panel-body">
          <div><span class="glyphicon glyphicon-briefcase" aria-hidden="true"></span>
{{user.role}}</div>
          <div><span class="glyphicon glyphicon-map-marker" aria-hidden="true"></span>
{{user.location}}</div>
          <div><span class="glyphicon glyphicon-link" aria-hidden="true"></span>
{{user.twitter}}</div>
        </div>
      </div>
    </div>
  </div>
</div>
```

There's one last step to get this working in the browser. Until now we've only been adding our files to `karma.conf.js`. First, add the controller and service to `index.html`.

```
<head>
  ...
  ...

  <script src="services/users/users.js"></script>

  <script src="components/users/users.js"></script>

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

Then add the modules to our application dependencies in `app/app.js`. While we're here, update `$urlRouterProvider.otherwise('/')` to default to the `users` state we just created.

```
(function() {
  'use strict';

  angular.module('meetIrl', [
    'ui.router',
    'api.users',
    'components.users'
  ])
  .config(function($urlRouterProvider) {

$urlRouterProvider.otherwise('/users');
  });
})();
```

Run `nodemon server.js`, navigate to `http://localhost:8080/#/users`, and you should see our four users.

Testing an Angular Factory and a Real API Endpoint

In Part 1 we covered how to test an Angular factory and now we've tested our first controller which consumes that same factory. But that was a simple factory that returned a hard-coded list of users. How do we test a factory that makes an actual HTTP request to a real API? As I mentioned earlier we're going to display an avatar of each user's favorite Pokémon on their individual profile page using Pokéapi.

First, let's create a new directory in our services folder for our `Pokemon` service.

```
cd app/services
mkdir pokemon && cd pokemon
touch pokemon.js
pokemon.spec.js
```

Within our factory we'll have one method, `findByName`, which makes a GET request to the `/pokemon/` endpoint which you can see here. This one request will provide us everything we need to populate our user profiles with all the necessary Pokémon data.

Like we've done previously we'll first set up a basic test and a basic factory to ensure everything is working correctly. Open up `/services/pokemon/pokemon.spec.js` and add the following code.

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


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


  beforeEach(inject(function(_Pokemon_) {
    Pokemon = _Pokemon_;
  }));


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

Then update `karma.conf.js` accordingly with our two new Pokémon files.

```
  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/services/pokemon/pokemon.js',
    './app/components/users/users.js',
    './app/app.js',
    './app/services/users/users.spec.js',
    './app/services/pokemon/pokemon.spec.js',
    './app/components/users/users.spec.js'
  ],
```

Restarting Karma should show a failing test. Add the following code to `/services/pokemon/pokemon.js` to make our test pass.

```
(function() {
  'use strict';


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

    return Pokemon;
  });
})();
```

Within our test file for our `Pokemon` service we are going to handle two cases, or response types, from Pokéapi. The first is a GET request with a valid Pokémon name. In this scenario, we'll use the successful response to populate our user profile image with an image of their favorite Pokémon. The second will be for a request with an invalid Pokémon name. In this case, we'll set the user profile image to a placeholder image to preserve the look of the profile page. Let's handle the valid API request first. Jump back into `/services/pokemon/pokemon.spec.js` and update it with the following code.

```
describe('Pokemon factory', function() {
  var Pokemon, $q, $httpBackend;


  var API = 'http://pokeapi.co/api/v2/pokemon/';


  var RESPONSE_SUCCESS = {
    'id': 25,
    'name': 'pikachu',
    'sprites': {
      'front_default': 'http://pokeapi.co/media/sprites/pokemon/25.png'
    },
    'types': [{
      'type': { 'name': 'electric' }
    }]
  };

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


  beforeEach(inject(function(_Pokemon_, _$q_, _$httpBackend_) {
```

```
    Pokemon = _Pokemon_;
    $q = _$q_;
    $httpBackend = _$httpBackend_;
  }));

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

  describe('findByName()', function() {
    var result;

    beforeEach(function() {

      result = {};


      spyOn(Pokemon, "findByName").and.callThrough();
    });

    it('should return a Pokemon when called with a valid name', function() {
      var search = 'pikachu';


      $httpBackend.whenGET(API + search).respond(200,
$q.when(RESPONSE_SUCCESS));

      expect(Pokemon.findByName).not.toHaveBeenCalled();
      expect(result).toEqual({});

      Pokemon.findByName(search)
      .then(function(res) {
        result = res;
      });


      $httpBackend.flush();

      expect(Pokemon.findByName).toHaveBeenCalledWith(search);
      expect(result.id).toEqual(25);
      expect(result.name).toEqual('pikachu');
      expect(result.sprites.front_default).toContain('.png');
      expect(result.types[0].type.name).toEqual('electric');
    });
  })
});
```

At the top of this file we've added a few more variables: `$httpBackend`, `$q`, `API`, and `RESPONSE_SUCCESS`. `API` simply serves as a variable for the Pokéapi endpoint we're hitting and `RESPONSE_SUCCESS` is one example of a successful response from Pokéapi for the resource "pikachu". If you look at the example response in the documentation or hit the endpoint yourself with Postman you'll see there is a lot of data that's returned. We'll only be using a small set of that data so we've removed everything else while maintaining the data structure of the response for these four fields.

We then set `$q` and `$httpBackend` to their respective injected services in our second `beforeEach` call. The `$q` service allows us to simulate resolving or rejecting a promise which is important when testing asynchronous calls. The `$httpBackend` service allows us to verify whether or not our `Pokemon` factory makes an HTTP request to Pokéapi without actually hitting the endpoint itself. The two of these services combined provide us the ability to verify a request was made to the API while also giving us the option to resolve or reject the response depending on which response we are testing.

> *Remember that an API and it's various responses are expectations of our application. We're merely testing that our application will be able to consume those various responses. As mentioned earlier, we'll want to set the profile image to a Pokémon if it's valid or default to a placeholder image if the request is invalid.*

Below our previous test we've added another `describe` block for the `findByName` method which will make an HTTP request to the Pokeapi. We declare a variable `result` which will be set to the result of our service call and set it to an empty object before each test is run in our `beforeEach` block. We also create a spy on the `findByName` method and chain it with another one of Jasmine's built-in functions `callThrough`. By chaining the spy with `callThrough` we have the ability to track any calls made to this function but the implementation will continue to the HTTP request that will be made within the function itself.

Finally, we have our test spec for an API call to Pokéapi service with a valid Pokemon name. After declaring our `search` value as "pikachu" we make our first use of the `$httpBackend` service we injected earlier. Here we've called the `whenGET` method and supplied it with the `API` variable we defined earlier along with our search term "pikachu". We then chain it with `respond` and provide it two arguments: 200 as the status code and `RESPONSE_SUCCESS` as its return value wrapped with `$q.when`. When `$q.when` wraps a value it converts it into a simulated resolved "then-able" promise which is the behavior we'd expect when calling an Angular service that returns a promise. So in plain English this says, "When a GET request is made to `http://pokeapi.co/api/v2/pokemon/pikachu`, respond with a 200 status code and the resolved response object we created earlier."

After this we create two expectations: one for the initial state of our result variable and another for the Pokemon service call. We're expecting our spy on `findByName` not to have been called and the result variable to be an empty object. Then we call `Pokemon.findByName`, pass in our search term and chain it with `.then` where we set the returned result to our local `result` variable. After that we call `$httpBackend.flush`.

> *If we were to call `Pokemon.findByName` in a controller, the service's `$http` request would respond asynchronously. Within our unit tests this asynchronous behavior would be difficult to test. Thankfully, Angular's `$httpBackend` service provides us the ability to "flush" pending requests so we can write our tests in a synchronous manner. Because of this, it is important that any expectations we have that would come after an asynchronous call is finished are placed after our `$httpBackend.flush()` call in our test.*

Finally, we create our final set of expectations from the result of our service call. Our first expectation utilizes the spy we created earlier to verify our service was called with the correct search term and the remaining four expectations verify that our result object contains all of the data related to Pikachu. Save that file and you should now see Karma showing a failing test.

We can get this test to pass in our service with just a few small additions. Open up `/services/pokemon/pokemon.js` and add the `findByName` method.

```
(function() {
  'use strict';

  angular.module('api.pokemon', [])
  .factory('Pokemon', function($http) {
    var API =
'http://pokeapi.co/api/v2/pokemon/';
    var Pokemon = {};


    Pokemon.findByName = function(name) {
      return $http.get(API + name)
      .then(function(res) {
        return res.data;
      });
    };

    return Pokemon;
  });
})();
```

Before adding the `findByName` method itself, we've injected the `$http` service into our factory and also created an `API` variable set to the Pokéapi endpoint we want to hit similar to the way we did in our test. After that we declare the `findByName` method and make a GET request to the endpoint with the name provided to us when the service is called. When the promise is fulfilled, we return the response's `data` property. Save that change and your first test for an Angular factory hitting a real API should now be passing!

That test handles our first case where a request is made to Pokéapi with a valid Pokemon. But we still need to handle the case where we make a request to the API with an invalid Pokémon. Within the context of our factory, we're going to test that we're able to `catch` a promise rejection from the API. Go back into `/services/pokemon/pokemon.spec.js` and add another test spec for our `findByName` method.

```
describe('Pokemon factory', function() {
  var Pokemon, $q, $httpBackend;
  var API = 'http://pokeapi.co/api/v2/pokemon/';
  var RESPONSE_SUCCESS = {
    'id': 25,
    'name': 'pikachu',
    'sprites': {
      'front_default': 'http://pokeapi.co/media/sprites/pokemon/25.png'
    },
    'types': [{
      'type': { 'name': 'electric' }
    }]
  };


  var RESPONSE_ERROR = {
    'detail': 'Not found.'
  };

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

  beforeEach(inject(function(_Pokemon_, _$q_, _$httpBackend_) {
    Pokemon = _Pokemon_;
    $q = _$q_;
    $httpBackend = _$httpBackend_;
  }));

  it('should exist', function() {
```

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

  describe('findByName()', function() {
    var result;

    beforeEach(function() {
      result = {};
      spyOn(Pokemon, "findByName").and.callThrough();
    });

    it('should return a Pokemon when called with a valid name', function() {
      var search = 'pikachu';
      $httpBackend.whenGET(API + search).respond(200,
$q.when(RESPONSE_SUCCESS));

      expect(Pokemon.findByName).not.toHaveBeenCalled();
      expect(result).toEqual({});

      Pokemon.findByName(search)
      .then(function(res) {
        result = res;
      });
      $httpBackend.flush();

      expect(Pokemon.findByName).toHaveBeenCalledWith(search);
      expect(result.id).toEqual(25);
      expect(result.name).toEqual('pikachu');
      expect(result.sprites.front_default).toContain('.png');
      expect(result.types[0].type.name).toEqual('electric');
    });

    it('should return a 404 when called with an invalid name', function() {

      var search = 'godzilla';


      $httpBackend.whenGET(API + search).respond(404,
$q.reject(RESPONSE_ERROR));

      expect(Pokemon.findByName).not.toHaveBeenCalled();
      expect(result).toEqual({});


      Pokemon.findByName(search)
      .catch(function(res) {
        result = res;
      });
      $httpBackend.flush();

      expect(Pokemon.findByName).toHaveBeenCalledWith(search);
      expect(result.detail).toEqual('Not found.');
    });
  });
});
```

This new test is nearly identical to our previous test. At the top of our file we added another variable `RESPONSE_ERROR` which is the response we expect to receive if we pass it an invalid name. In our second test, we declare that we expect to receive a 404 when hitting the API with an invalid name. From there we change our search term from "pikachu" to "godzilla" and update our `whenGET` to respond with a 404 status code and our new

`RESPONSE_ERROR` variable wrapped with `q.reject` so that we can `catch` our rejected promise when we call `Pokemon.findByName`. Finally, we update our expectations for our `result` to test for the `detail` property of our response.

> *The Pokéapi documentation isn't explicit about this response error object but I hit the API with multiple, incorrect search terms and received the same response every time. I also looked into the project on Github and the else statement for this call raises a 404 if it can't find a match for our given search term. The 404 is more important here anyway since we'll be defaulting to a placeholder image instead of using the returned response text in our view.*

Save that file and Karma should now show our new test as a failing test. Go back into `/services/pokemon/pokemon.js` and add a `catch` to our HTTP request.

```
(function() {
  'use strict';

  angular.module('api.pokemon', [])
  .factory('Pokemon', function($http) {
    var API =
'http://pokeapi.co/api/v2/pokemon/';
    var Pokemon = {};

    Pokemon.findByName = function(name) {
      return $http.get(API + name)
      .then(function(res) {
        return res.data;
      })
      .catch(function(res) {
        return res.data;
      });
    };

    return Pokemon;
  });
})();
```

Save that file and our failing test should now be passing. We have now created an Angular factory that hits a real API and have the associated tests for both a valid and invalid response from Pokéapi. This fully tested service gives us the confidence to move on to the next and final part of our application where we create a new component for our user profile which will make the actual request to Pokéapi using our `Pokemon` factory.

A Quick Update to Our Users

Before we get started creating the profile page for our users, we'll need to update the users in our `Users` service so they each have a favorite Pokémon we can use to call our `Pokemon` service. What can I say? I didn't expect Pokémon to make a comeback when I was writing Part 1 of this tutorial.

Open `services/users/users.js` and update each user in the `userList` with a `pokemon` object and a `name` property within that object.

```
(function() {
  'use strict';

  angular.module('api.users', [])
  .factory('Users', function() {
    var Users = {};
    var userList = [
      {
        id: '1',
        name: 'Jane',
        role: 'Designer',
        location: 'New York',
        twitter: 'gijane',
        pokemon: { name: 'blastoise' }
      },
      {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      },
      {
        id: '3',
        name: 'Jim',
        role: 'Developer',
        location: 'Chicago',
        twitter: 'jimbo',
        pokemon: { name: 'hitmonchan'
}
      },
      {
        id: '4',
        name: 'Bill',
        role: 'Designer',
        location: 'LA',
        twitter: 'dabill',
        pokemon: { name: 'barney' }
      }
    ];

    ...

  });
})();
```

You're free to use any Pokémon here you'd like but remember to update your test files accordingly once you're done. You should make updates to `/services/users/users.spec.js` and `/components/users/users.spec.js` to reflect our new list of users and their favorite Pokémon. I've also added one invalid Pokémon for the sake of having at least one user default to our placeholder image we'll add shortly.

Creating and Testing a Controller for User Profiles

Before we create our user profile controller and its associated test, let's take a minute to recap the expected behavior of our profile page so we can incrementally build our way to the finished feature one test case at a time. When a user navigates to a profile page for a given user, we're going to provide the user object to our controller using the `resolve` property provided to us by `ui-router`.

Our first test (not including our very basic `toBeDefined` test) will test that our controller is instantiated with a valid resolved user. Our second test will test that a valid resolved user with a valid Pokémon will make a request to the Pokéapi using our `Pokemon` service. This value will eventually be set to a view-model object to be used within our view. Our third test will test that a valid resolved user with an invalid Pokémon will make a request to the Pokéapi using our `Pokemon` service, catch the rejection, and default our view-model object to a placeholder image. Finally, we'll add one extra test for a user that doesn't exist which will redirect to a 404 page without ever making a request to the Pokéapi. Let's get started.

We'll start by creating the directory for our user profile controller, view, and test file as usual.

```
cd app/components
mkdir profile && cd profile
touch profile.js profile.spec.js
profile.html
```

Open `/components/profile/profile.spec.js` and we can add our basic test for the existence of our controller.

```
describe('components.profile', function() {
  var $controller;


  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('components.profile'));


  beforeEach(inject(function(_$controller_) {
    $controller = _$controller_;
  }));

  describe('ProfileController', function() {
    var ProfileController;

    beforeEach(function() {

      ProfileController = $controller('ProfileController', {
});
    });


    it('should be defined', function() {
      expect(ProfileController).toBeDefined();
    });
  });
});
```

Then add our two new files to `karma.conf.js`.

```
  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/services/pokemon/pokemon.js',
    './app/components/users/users.js',
    './app/components/profile/profile.js',
    './app/app.js',
    './app/services/users/users.spec.js',
    './app/services/pokemon/pokemon.spec.js',
    './app/components/users/users.spec.js',
    './app/components/profile/profile.spec.js'
  ],
```

Restarting Karma should show a failing test. Add the following to `/components/profile/profile.js` to make it pass.

```
(function() {
  'use strict';


  angular.module('components.profile', [])
  .controller('ProfileController', function() {
    var vm = this;
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('profile', {
        url: '/user/:id',
        templateUrl:
'components/profile/profile.html',
        controller: 'ProfileController as pc'
      });
  });
})();
```

Now we're ready for our first real test to verify that our controller is instantiated with a resolved user object. Go back into `/components/profile/profile.spec.js` and add our new test.

```
describe('components.profile', function() {
  var $controller;

  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('components.profile'));

  beforeEach(inject(function(_$controller_) {
    $controller = _$controller_;
  }));

  describe('ProfileController', function() {
    var ProfileController, singleUser;

    beforeEach(function() {

      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };
      ProfileController = $controller('ProfileController', { resolvedUser: singleUser
});
    });

    it('should be defined', function() {
      expect(ProfileController).toBeDefined();
    });
  });

  describe('Profile Controller with a valid resolved user', function() {
    var ProfileController, singleUser;

    beforeEach(function() {

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


      ProfileController = $controller('ProfileController', { resolvedUser: singleUser
});
    });

    it('should set the view model user object to the resolvedUser', function() {
      expect(ProfileController.user).toEqual(singleUser);
    });
  });
});
```

Here we've added a new describe block for our tests related to a valid resolved user. This test is similar to our previous test except we've added another `beforeEach` call to mock our single resolved user. We then pass it in as a dependency to our controller and add an expectation that the resolved user will be set to the `user` view-model

property in our controller.

> *Since our tests serve as a form of documentation for our actual code, I've also added a `singleUser` to our previous test and passed it in as a dependency to that controller instance as well. We didn't specify any expectations about the resolved user within that test but it keeps our controller declarations consistent with our actual controller and the other controller declarations within this file. As we continue to add more dependencies in this test file, we'll go back and update our other tests to reflect this.*

To get this test to pass, go back into `/components/profile/profile.js` and update it with our new resolved property.

```
(function() {
  'use strict';

  angular.module('components.profile', [])
  .controller('ProfileController', function(resolvedUser) {
    var vm = this;
    vm.user = resolvedUser;
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('profile', {
        url: '/user/:id',
        templateUrl: 'components/profile/profile.html',
        controller: 'ProfileController as pc',
        resolve: {

          resolvedUser: function(Users, $stateParams) {
            return Users.findById($stateParams.id);
          }
        }
      });
  });
})();
```

Here we add the new `resolve` property to our controller configuration with `resolvedUser` being set to the user returned by our `Users.findById` method. Within the controller, we then set this to our view-model `user` property as we stated within our test. Once again, we're not concerned with testing expectations related to our `Users` service here. That's delegated to the test file for our service in `/services/users/users.spec.js`.

Now that we've finished our first test for a valid resolved user, let's move on to testing a call to the `Pokemon` service using the resolved user's Pokemon. Go back into `/components/profile/profile.spec.js` and add the following updates.

```
describe('components.profile', function() {
  var $controller, PokemonFactory, $q, $httpBackend;
  var API = 'http://pokeapi.co/api/v2/pokemon/';
  var RESPONSE_SUCCESS = {
    'id': 58,
    'name': 'growlithe',
    'sprites': {
      'front_default': 'http://pokeapi.co/media/sprites/pokemon/58.png'
    },
    'types': [{
      'type': { 'name': 'fire' }
    }]
```

```
  };


  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('api.pokemon'));
  beforeEach(angular.mock.module('components.profile'));


  beforeEach(inject(function(_$controller_, _Pokemon_, _$q_, _$httpBackend_) {
    $controller = _$controller_;
    PokemonFactory = _Pokemon_;
    $q = _$q_;
    $httpBackend = _$httpBackend_;
  }));

  describe('ProfileController', function() {
    var ProfileController, singleUser;

    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };

      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory });
    });

    it('should be defined', function() {
      expect(ProfileController).toBeDefined();
    });
  });


  describe('Profile Controller with a valid resolved user and a valid Pokémon',
function() {
    var singleUser, ProfileController;

    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };


      spyOn(PokemonFactory, "findByName").and.callThrough();


      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory });
    });

    it('should set the view model user object to the resolvedUser', function() {
      expect(ProfileController.user).toEqual(singleUser);
```

```
    });

    it('should call Pokemon.findByName and return a Pokemon object', function() {

      expect(ProfileController.user.pokemon.id).toBeUndefined();
      expect(ProfileController.user.pokemon.name).toEqual('growlithe');
      expect(ProfileController.user.pokemon.image).toBeUndefined();
      expect(ProfileController.user.pokemon.type).toBeUndefined();


      $httpBackend.whenGET(API + singleUser.pokemon.name).respond(200,
$q.when(RESPONSE_SUCCESS));
      $httpBackend.flush();


      expect(PokemonFactory.findByName).toHaveBeenCalledWith('growlithe');
      expect(ProfileController.user.pokemon.id).toEqual(58);
      expect(ProfileController.user.pokemon.name).toEqual('growlithe');
      expect(ProfileController.user.pokemon.image).toContain('.png');
      expect(ProfileController.user.pokemon.type).toEqual('fire');
    });
  });
});
```

Starting at the top we've added a few more variables. `PokemonFactory`, `$q`, and `$httpBackend` will be set to their respective injected services. `API` and `RESPONSE_SUCCESS` will be used when we test our controller's call to Pokéapi using our `Pokemon` service. After that we load our `api.pokemon` module and set all of our variables to their injected services.

Then, we updated our second `describe` title to include our expectation that this test will be working with a valid resolved user with a valid Pokémon. In the `beforeEach` within this `describe` we add a spy to our `PokemonFactory`'s `findByName` method and chain it with `callThrough` so that our call to the service continues on to the actual HTTP request within the service. We also add the `PokemonFactory` as a dependency to both of our controller instances.

Below our previous test we then add another expectation for our controller that it will make a request using our `Pokemon` service. Similar to our service test, we use `$httpBackend`'s `whenGET` to state the API endpoint we expect to hit and supply it with a 200 status code and our `RESPONSE_SUCCESS` variable we defined at the top of this file. We then `flush` our asynchronous request to Pokéapi and list all of our expectations for the result and the view-model properties they will be set to.

> *Unlike our service test we don't actually call `Pokemon.findByName` here directly. Instead, that call will occur within our controller after we set our view-model's `users` attribute to the resolved user object as we did earlier. The expectations before that call occurs within our controller are placed above `$httpBackend.flush` and the expectations after that asynchronous call finishes are placed after `$httpBackend.flush`. This goes back to Angular's `$httpBackend` service providing us the ability to test asynchronous calls in a synchronous manner within our tests. As far as `$httpBackend.whenGET` is concerned, that can be placed anywhere within this `it` block and even above within our `beforeEach` block for this test suite. That line simply waits for a request to be made to the endpoint and responds accordingly. `flush()` is the magic line which triggers our service call to resolve or reject within our test case.*

This may be a little confusing so let's add the code to make the test pass in our controller. Go back into `/components/profile/profile.js` and add our call to the Pokémon service.

```
(function() {
  'use strict';

  angular.module('components.profile', [])
  .controller('ProfileController', function(resolvedUser, Pokemon) {
    var vm = this;
    vm.user = resolvedUser;


    Pokemon.findByName(vm.user.pokemon.name)
    .then(function(result) {
      vm.user.pokemon.id = result.id;
      vm.user.pokemon.image = result.sprites.front_default;
      vm.user.pokemon.type = result.types[0].type.name;
    });
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('profile', {
        url: '/user/:id',
        templateUrl: 'components/profile/profile.html',
        controller: 'ProfileController as pc',
        resolve: {
          resolvedUser: function(Users, $stateParams) {
            return Users.findById($stateParams.id);
          }
        }
      });
  });
})();
```

First, we add the `Pokemon` service as a dependency to our controller. Then we call the `findByName` method with the resolved user's Pokémon, `vm.user.pokemon.name`. We then chain it with `then` and set all of the properties we stated earlier in our test to their respective properties in our returned `result` object. Before that call is made the values for `id`, `image`, and `type` would be `undefined` as we stated in our test above our call to `$httpBackend.flush`.

Now that we've tested a call to the `Pokemon` service with a valid Pokémon, let's add the test for an invalid Pokémon. The good news is that these tests are very similar with only a few small changes.

```
describe('components.profile', function() {
  var $controller, PokemonFactory, $q, $httpBackend;
  var API = 'http://pokeapi.co/api/v2/pokemon/';
  var RESPONSE_SUCCESS = {
    'id': 58,
    'name': 'growlithe',
    'sprites': {
      'front_default': 'http://pokeapi.co/media/sprites/pokemon/58.png'
    },
    'types': [{
      'type': { 'name': 'fire' }
    }]
  };


  var RESPONSE_ERROR = {
    'detail': 'Not found.'
  };

  beforeEach(angular.mock.module('ui.router'));
```

```
  beforeEach(angular.mock.module('api.pokemon'));
  beforeEach(angular.mock.module('components.profile'));

  beforeEach(inject(function(_$controller_, _Pokemon_, _$q_, _$httpBackend_) {
    $controller = _$controller_;
    PokemonFactory = _Pokemon_;
    $q = _$q_;
    $httpBackend = _$httpBackend_;
  }));

  describe('ProfileController', function() {
    var ProfileController, singleUser;

    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };
      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory });
    });

    it('should be defined', function() {
      expect(ProfileController).toBeDefined();
    });
  });

  describe('Profile Controller with a valid resolved user and a valid Pokemon',
function() {
    var singleUser, ProfileController;

    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };

      spyOn(PokemonFactory, "findByName").and.callThrough();

      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory });
    });

    it('should set the view model user object to the resolvedUser', function() {
      expect(ProfileController.user).toEqual(singleUser);
    });

    it('should call Pokemon.findByName and return a Pokemon object', function() {
      expect(ProfileController.user.pokemon.id).toBeUndefined();
      expect(ProfileController.user.pokemon.name).toEqual('growlithe');
      expect(ProfileController.user.pokemon.image).toBeUndefined();
      expect(ProfileController.user.pokemon.type).toBeUndefined();

      $httpBackend.whenGET(API + singleUser.pokemon.name).respond(200,
```

```
$q.when(RESPONSE_SUCCESS));
      $httpBackend.flush();

      expect(PokemonFactory.findByName).toHaveBeenCalledWith('growlithe');
      expect(ProfileController.user.pokemon.id).toEqual(58);
      expect(ProfileController.user.pokemon.name).toEqual('growlithe');
      expect(ProfileController.user.pokemon.image).toContain('.png');
      expect(ProfileController.user.pokemon.type).toEqual('fire');
    });
  });


  describe('Profile Controller with a valid resolved user and an invalid Pokemon',
function () {
    var singleUser, ProfileController;

    beforeEach(function() {

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

      spyOn(PokemonFactory, "findByName").and.callThrough();

      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory });
    });

    it('should call Pokemon.findByName and default to a placeholder image', function()
{
      expect(ProfileController.user.pokemon.image).toBeUndefined();


      $httpBackend.whenGET(API + singleUser.pokemon.name).respond(404,
$q.reject(RESPONSE_ERROR));
      $httpBackend.flush();


      expect(PokemonFactory.findByName).toHaveBeenCalledWith('godzilla');

expect(ProfileController.user.pokemon.image).toEqual('http://i.imgur.com/HddtBOT.png');
    });
  });
});
```

First, we add another `describe` block for a valid resolved user with an invalid Pokémon. Then we change our pokemon value from `growlithe` to `godzilla`. From there we change our `whenGET` to respond with a 404 and to reject our `RESPONSE_ERROR` object so that we can `catch` it in our controller. Finally, we update our expectations for the `image` property. Before the promise is rejected we expect the property to be undefined. Once the promise is actually rejected, we'll set the image property to our placeholder image.

> *Earlier in the tutorial I mentioned a seemingly unnecessary line of code:* `$controller = _$controller_;` *. This is where that's paying off. When we only had one* `userList` *to test in our* `UsersController` *we could have avoided that variable declaration. But as*

> *we see here, we're now testing a controller with a slightly modified `resolvedUser`. While the dependency is the same the object itself is different. In this case it's the Pokémon's name so the ability to have separate controller instances within each test block is needed to successfully test our controller.*

To make this test pass open `/components/profile/profile.js` and complete our call to `Pokemon.findByName`.

```
(function() {
  'use strict';

  angular.module('components.profile', [])
  .controller('ProfileController', function(resolvedUser, Pokemon) {
    var vm = this;
    vm.user = resolvedUser;

    Pokemon.findByName(vm.user.pokemon.name)
    .then(function(result) {
      vm.user.pokemon.id = result.id;
      vm.user.pokemon.image = result.sprites.front_default;
      vm.user.pokemon.type = result.types[0].type.name;
    })
    .catch(function(result) {

      vm.user.pokemon.image = 'http://i.imgur.com/HddtBOT.png';
    });
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('profile', {
        url: '/user/:id',
        templateUrl: 'components/profile/profile.html',
        controller: 'ProfileController as pc',
        resolve: {
          resolvedUser: function(Users, $stateParams) {
            return Users.findById($stateParams.id);
          }
        }
      });
  });
})();
```

As our test stated, when the promise is rejected we set the `image` property on our view-model to a placeholder image. We're almost done. Before we finish with our last test for redirecting our users to a 404 page, let's create that component first.

Creating 404 component

Our 404 page is going to be extremely basic. We'll test it for the sake of extra practice but it won't do much since the page will largely be an HTML page with a hardcoded image within it. For that reason, we'll work through this without all the details since a lot of this is the boilerplate we've seen across all of our previous controller and factory tests.

We'll start off as usual creating a directory for our 404 component.

```
cd app/components
mkdir missingno && cd missingno
touch missingno.js missingno.spec.js
missingno.html
```

In `/components/missingno/missingno.spec.js` we'll add our basic test for our controller.

```
describe('components.missingno', function() {
  var $controller, MissingnoController;


  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('components.missingno'));


  beforeEach(inject(function(_$controller_) {
    $controller = _$controller_;
    MissingnoController = $controller('MissingnoController',
{});
  }));


  it('should be defined', function() {
    expect(MissingnoController).toBeDefined();
  });
});
```

As usual we bring in our required modules, inject the `$controller` service, create an instance of our controller, and create an expectation for it to be defined.

Once again, let's add our files to `karma.conf.js`.

```
  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/services/pokemon/pokemon.js',
    './app/components/users/users.js',
    './app/components/profile/profile.js',
    './app/components/missingno/missingno.js',
    './app/app.js',
    './app/services/users/users.spec.js',
    './app/services/pokemon/pokemon.spec.js',
    './app/components/users/users.spec.js',
    './app/components/profile/profile.spec.js',
    './app/components/missingno/missingno.spec.js'
  ],
```

Then we create our controller in `/components/missingno/missingno.js`.

```
(function() {
  'use strict';


  angular.module('components.missingno', [])
  .controller('MissingnoController', function() {
    var vm = this;
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('404', {
        url: '/404',
        templateUrl:
'components/missingno/missingno.html',
        controller: 'MissingnoController as mn'
      });
  });
})();
```

And populate our view in `/components/missingno/missingno.html`.

```
<div class="container">
  <div class="row">
    <div class="col-md-4 col-md-offset-4 text-center">
      <div><img src="http://i.imgur.com/5pG5t.jpg"
class="missingno"></div>
      <a ui-sref="users">RUN</a>
    </div>
  </div>
</div>
```

Before we get this working in a browser we'll also need to add our file to `index.html` and our module to `app.js`.

```
<head>
  ...
  ...

  <script src="services/users/users.js"></script>

  <script src="components/users/users.js"></script>

  <script src="components/missingno/missingno.js"></script>

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

```
(function() {
  'use strict';

  angular.module('meetIrl', [
    'ui.router',
    'api.users',
    'components.users',
    'components.missingno'
  ])
  .config(function($urlRouterProvider) {

$urlRouterProvider.otherwise('/users');
  });
})();
```

Open your browser to `http://localhost:8080/#/404` and you should see our newly created 404 page!

Now we can update our `ProfileController` and its test to redirect us to this page in the case of a missing user.

Testing a State Change to a 404 Page for Missing Users

In the case that a user navigates to a url such as `http://localhost:8080/user/scotch` or `http://localhost:8080/user/999`, assuming a user with an id "999" doesn't exist, we'll want to trigger a state change to our new 404 page.

Let's add our test for this new expected behavior. Open `/components/profile/profile.spec.js` and add our new test.

```
describe('components.profile', function() {
  var $controller, PokemonFactory, $q, $httpBackend, $state;
  var API = 'http://pokeapi.co/api/v2/pokemon/';
  var RESPONSE_SUCCESS = {
    'id': 58,
    'name': 'growlithe',
    'sprites': {
      'front_default': 'http://pokeapi.co/media/sprites/pokemon/58.png'
    },
    'types': [{
      'type': { 'name': 'fire' }
    }]
  };


  var RESPONSE_ERROR = {
    'detail': 'Not found.'
  };

  beforeEach(angular.mock.module('ui.router'));
  beforeEach(angular.mock.module('api.pokemon'));
  beforeEach(angular.mock.module('components.profile'));


  beforeEach(inject(function(_$controller_, _Pokemon_, _$q_, _$httpBackend_, _$state_)
{
    $controller = _$controller_;
    PokemonFactory = _Pokemon_;
    $q = _$q_;
    $httpBackend = _$httpBackend_;
    $state = _$state_;
  }));

  describe('ProfileController', function() {
    var ProfileController;

    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };

      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory, $state: $state });
    });

    it('should be defined', function() {
      expect(ProfileController).toBeDefined();
    });
  });

  describe('Profile Controller with a valid resolved user and a valid Pokemon',
function() {
    var singleUser, ProfileController;
```

```
    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'growlithe' }
      };

      spyOn(PokemonFactory, "findByName").and.callThrough();

      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory, $state: $state });
    });

    it('should set the view model user object to the resolvedUser', function() {
      expect(ProfileController.user).toEqual(singleUser);
    });

    it('should call Pokemon.findByName and return a Pokemon object', function() {
      expect(ProfileController.user.pokemon.id).toBeUndefined();
      expect(ProfileController.user.pokemon.name).toEqual('growlithe');
      expect(ProfileController.user.pokemon.image).toBeUndefined();
      expect(ProfileController.user.pokemon.type).toBeUndefined();

      $httpBackend.whenGET(API + singleUser.pokemon.name).respond(200,
$q.when(RESPONSE_SUCCESS));
      $httpBackend.flush();

      expect(PokemonFactory.findByName).toHaveBeenCalledWith('growlithe');
      expect(ProfileController.user.pokemon.id).toEqual(58);
      expect(ProfileController.user.pokemon.name).toEqual('growlithe');
      expect(ProfileController.user.pokemon.image).toContain('.png');
      expect(ProfileController.user.pokemon.type).toEqual('fire');
    });
  });

  describe('Profile Controller with a valid resolved user and an invalid Pokemon',
function () {
    var singleUser, ProfileController;

    beforeEach(function() {
      singleUser = {
        id: '2',
        name: 'Bob',
        role: 'Developer',
        location: 'New York',
        twitter: 'billybob',
        pokemon: { name: 'godzilla' }
      };

      spyOn(PokemonFactory, "findByName").and.callThrough();

      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory, $state: $state });
    });

    it('should call Pokemon.findByName and default to a placeholder image', function()
{
      expect(ProfileController.user.pokemon.image).toBeUndefined();
```

```
      $httpBackend.whenGET(API + singleUser.pokemon.name).respond(404,
$q.reject(RESPONSE_ERROR));
      $httpBackend.flush();

      expect(PokemonFactory.findByName).toHaveBeenCalledWith('godzilla');

expect(ProfileController.user.pokemon.image).toEqual('http://i.imgur.com/HddtBOT.png');
    });
  });

  describe('Profile Controller with an invalid resolved user', function() {
    var singleUser, ProfileController;

    beforeEach(function() {

      spyOn($state, "go");
      spyOn(PokemonFactory, "findByName");


      ProfileController = $controller('ProfileController', { resolvedUser: singleUser,
Pokemon: PokemonFactory, $state: $state });
    });

    it('should redirect to the 404 page', function() {
      expect(ProfileController.user).toBeUndefined();
      expect(PokemonFactory.findByName).not.toHaveBeenCalled();
      expect($state.go).toHaveBeenCalledWith('404');
    });
  });
});
```

At the top of the file we declare a new variable `$state`. We then inject the `$state` service and set that to our `$state` variable. We then add another `describe` block for our controller test with an invalid resolved user. Within the block we declare `singleUser` but leave it as `undefined`. If you'll recall from Part 1 of this tutorial, that's exactly the return value we would expect from our `Users.findById` service call and we even wrote a test for that behavior in `/services/users/users.spec.js`.

We then create two spies: one for the `go` method of the `$state` service and another for the `findByName` method of our `PokemonFactory`. We then pass in both of these as dependencies to our controllers. Finally, we create our test expectation to redirect to our 404 page. First we specify our expectation that the `resolvedUser` is `undefined` and then we utilize our spies to ensure `PokemonFactory.findByName` isn't called and that `$state.go` is called to redirect us to our 404 page.

We can make this failing test pass with a small update to our profile controller. Go back into `/components/profile/profile.js` and add the following conditional for our `resolvedUser` along with our new `$state` dependency.

```
(function() {
  'use strict';

  angular.module('components.profile', [])
  .controller('ProfileController', function(resolvedUser, Pokemon, $state)
{
    var vm = this;


    if (resolvedUser) {
      vm.user = resolvedUser;
    } else {
      return $state.go('404');
    }

    Pokemon.findByName(vm.user.pokemon.name)
    .then(function(result) {
      vm.user.pokemon.id = result.id;
      vm.user.pokemon.image = result.sprites.front_default;
      vm.user.pokemon.type = result.types[0].type.name;
    })
    .catch(function(result) {
      vm.user.pokemon.image = 'http://i.imgur.com/HddtBOT.png';
    });
  })
  .config(function($stateProvider) {
    $stateProvider
      .state('profile', {
        url: '/user/:id',
        templateUrl: 'components/profile/profile.html',
        controller: 'ProfileController as pc',
        resolve: {
          resolvedUser: function(Users, $stateParams) {
            return Users.findById($stateParams.id);
          }
        }
      });
  });
})();
```

Save that file and all of our tests should now be passing.

> *It's worth noting that we added a conditional statement here for `resolvedUser` which made our new test for the 404 page pass without breaking our previous test for `should set the view model user object to the resolvedUser`. This test says nothing about how this will be done, it only cares that it actually happens. Within the `if` statement we could `if(true)` nest ten more `{}` statements and our test would still pass. That wouldn't make much sense logically speaking but once again our tests only care that our `ProfileController` behaves as expected with all of our various test cases. The implementation to make them pass is up to you.*

Now that our `ProfileController` is completed and fully tested, let's update our template so we can see this code in action. Open up `/components/profile/profile.html` and add the following code.

```
<div class="container">
  <div class="row">
    <div class="col-md-4 col-md-offset-4">
      <div class="panel panel-default">
        <div class="panel-heading">
          <div class="text-center">
            <img ng-src="{{pc.user.pokemon.image}}" class="img-circle pokemon">
          </div>
          <h3 class="panel-title text-center">{{pc.user.name}}</h3>
        </div>
        <div class="panel-body text-center">
          <div><span class="glyphicon glyphicon-briefcase" aria-hidden="true"></span>
{{pc.user.role}}</div>
          <div><span class="glyphicon glyphicon-map-marker" aria-hidden="true"></span>
{{pc.user.location}}</div>
          <div><span class="glyphicon glyphicon-link" aria-hidden="true"></span>
{{pc.user.twitter}}</div>
          <div><span class="glyphicon glyphicon-leaf" aria-hidden="true"></span>
{{pc.user.pokemon.name}}</div>
          <div><span class="glyphicon glyphicon-tag" aria-hidden="true"></span>
{{pc.user.pokemon.type}}</div>
        </div>
      </div>
    </div>
  </div>
</div>
```

And add some styling for the profile image to `app.css`.

```
.pokemon {
  max-width: 75px;
  height: 75px;
  border: 1px solid
white;
}
```

While we're at it, let's also add the ability to navigate to this page from our `/components/users/users.html` page by adding a `ui-sref` to each user's name.

```
<div class="container">
  <div class="row">
    <div class="col-md-4" ng-repeat="user in uc.users">
      <div class="panel panel-default">
        <div class="panel-heading">
          <h3 class="panel-title text-center"><a ui-sref="profile({id:
user.id})">{{user.name}}</a></h3>
        </div>
        <div class="panel-body">
          <div><span class="glyphicon glyphicon-briefcase" aria-hidden="true"></span>
{{user.role}}</div>
          <div><span class="glyphicon glyphicon-map-marker" aria-hidden="true"></span>
{{user.location}}</div>
          <div><span class="glyphicon glyphicon-link" aria-hidden="true"></span>
{{user.twitter}}</div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Finally, we'll once again need to update our `index.html` and `app.js` to include our `api.pokemon` and `components.profile` modules.

```
<head>
  ...
  ...

  <script src="services/users/users.js"></script>

  <script src="services/pokemon/pokemon.js"></script>

  <script src="components/users/users.js"></script>

  <script src="components/profile/profile.js"></script>
  <script src="components/missingno/missingno.js"></script>

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

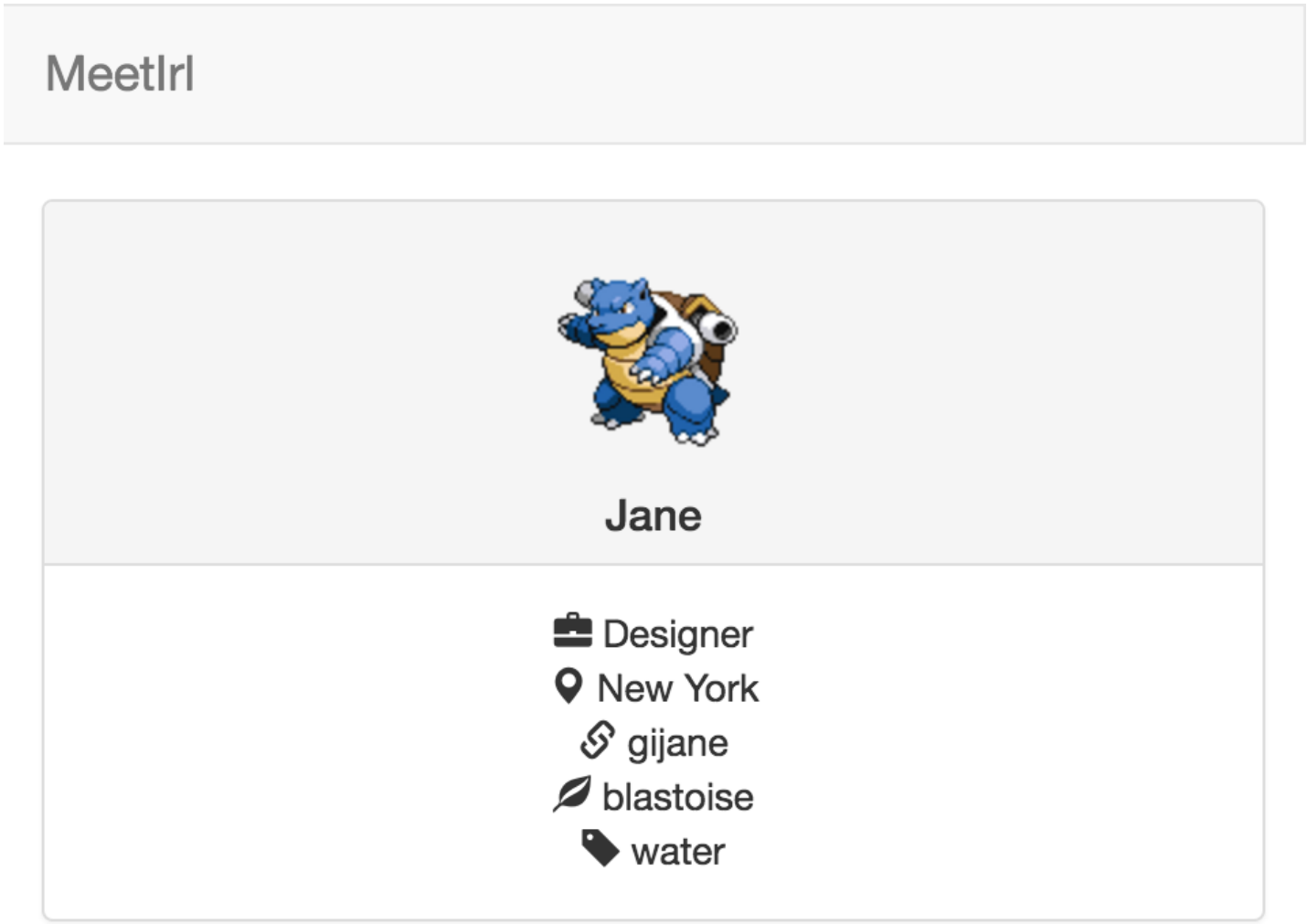
```
(function() {
  'use strict';

  angular.module('meetIrl', [
    'ui.router',
    'api.users',
    'api.pokemon',
    'components.users',
    'components.profile',
    'components.missingno'
  ])
  .config(function($urlRouterProvider) {

$urlRouterProvider.otherwise('/users');
  });
})();
```

With those final changes, you should now be able to click on each user's name in our `/users` page and see an

image for their favorite Pokémon within their profile page.



Conclusion

At this point we've now learned how to test a controller and a service that hits a real API. In our service tests, we utilized `$httpBackend` to listen to HTTP endpoints and `$q` to resolve or reject our expected responses from the API. We then learned how to test our controllers with all of its dependencies including our tested services. Given that our users can have valid or invalid favorite Pokémon names, we finally learned how to test the logic within our controller. We did this using multiple controller instances within our tests each with their own `resolvedUser`.

Bonus - Testing an Angular Filter

The Pokéapi is very specific about the search term it expects. The value we provide must be entirely lowercase. Send a GET request with "Pikachu" and it won't work. That's fine for our service call but when we display the user's Pokémon in their profile page we'd like it to be proper case. Let's create a simple filter to capitalize the first letter of a given string so we can use it in our view template. First, let's create a directory for our filter.

```
cd app/ && mkdir filters && cd
filters
mkdir capitalize && cd capitalize
touch capitalize.js
capitalize.spec.js
```

Open up `/filters/capitalize/capitalize.spec.js` and add the following test for our filter.

```
describe('Capitalize filter', function() {
  var capitalizeFilter;


  beforeEach(angular.mock.module('filters.capitalize'));


  beforeEach(inject(function(_$filter_) {
    capitalizeFilter = _$filter_('capitalize');
  }));

  it('should capitalize the first letter of a string', function()
{
    expect(capitalizeFilter('blastoise')).toEqual('Blastoise');
  });
});
```

Similar to our other tests, we load our module `filters.capitalize`, inject the `$filter` service, and create an instance of the filter by calling it with our service name `capitalize` and setting it to our `capitalizeFilter` variable. We then create a test for our filter providing it a lowercase Pokémon name "blastoise" with the expectation that the return value will be "Blastoise".

Once again, add these two files to `karma.conf.js` to reveal our failing test.

```
  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/services/pokemon/pokemon.js',
    './app/components/users/users.js',
    './app/components/profile/profile.js',
    './app/components/missingno/missingno.js',
    './app/filters/capitalize/capitalize.js',
    './app/app.js',
    './app/services/users/users.spec.js',
    './app/services/pokemon/pokemon.spec.js',
    './app/components/users/users.spec.js',
    './app/components/profile/profile.spec.js',
    './app/components/missingno/missingno.spec.js',
    './app/filters/capitalize/capitalize.spec.js'
  ],
```

Then we can go into `/filters/capitalize/capitalize.js` and create our filter.

```
(function() {
  'use strict';


  angular.module('filters.capitalize', [])
  .filter('capitalize', function() {
    return function(word) {
      return (word) ? word.charAt(0).toUpperCase() + word.substring(1) :
'';
    };
  });
})();
```

Save that and our test should be passing. To use this in our app let's add it to our `index.html` file and add it as a dependency to our `app.js` file.

```
<head>
  ...
  ...
  <script src="filters/capitalize/capitalize.js"></script>

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


(function() {
  'use strict';

  angular.module('meetIrl', [
    'ui.router',
    'api.users',
    'api.pokemon',
    'components.users',
    'components.profile',
    'components.missingno',
    'filters.capitalize'
  ])
  .config(function($urlRouterProvider) {

$urlRouterProvider.otherwise('/users');
  });
})();
```

Now we can update our profile page at `/components/profile/profile.html` to use our new `capitalize` filter.

```
...
    <div class="panel-body text-center">
      <div><span class="glyphicon glyphicon-briefcase" aria-hidden="true"></span>
{{pc.user.role}}</div>
      <div><span class="glyphicon glyphicon-map-marker" aria-hidden="true"></span>
{{pc.user.location}}</div>
      <div><span class="glyphicon glyphicon-link" aria-hidden="true"></span>
{{pc.user.twitter}}</div>
      <div><span class="glyphicon glyphicon-leaf" aria-hidden="true"></span>
{{pc.user.pokemon.name | capitalize}}</div>
      <div><span class="glyphicon glyphicon-tag" aria-hidden="true"></span>
{{pc.user.pokemon.type | capitalize}}</div>
    </div>
...
```

MeetIrl

Jane

Designer
New York
gijane
Blastoise
Water