In this step, you will change the way our app fetches data.

- We define a custom service that represents a RESTful client. Using this client we can make requests to the server for data in an easier way, without having to deal with the lower-level $http API, HTTP methods and URLs.

Workspace Reset Instructions ➤

The most important changes are listed below. You can see the full diff on GitHub

---

Dependencies

The RESTful functionality is provided by Angular in the `ngResource` module, which is distributed separately from the core Angular framework.

We are using Bower to install client side dependencies. This step updates the `bower.json` configuration file to include the new dependency:

```
{
"name": "angular-seed",
"description": "A starter project for AngularJS",
"version": "0.0.0",
"homepage": "https://github.com/angular/angular-seed",
"license": "MIT",
"private": true,
"dependencies": {
  "angular": "~1.3.0",
  "angular-mocks": "~1.3.0",
  "bootstrap": "~3.1.1",
  "angular-route": "~1.3.0",
  "angular-resource": "~1.3.0"
}
}
```

The new dependency `"angular-resource": "~1.3.0"` tells bower to install a version of the angular-resource component that is compatible with version 1.3.x. We must ask bower to download and install this dependency. We can do this by running:

```
npm install
```

Warning: If a new version of Angular has been released since you last ran `npm install`, then you may have a problem with the `bower install` due to a conflict between the versions of angular.js that need to be installed. If you get this then simply delete your `app/bower_components` folder before running `npm install`.

Note: If you have bower installed globally then you can run `bower install` but for this project we have preconfigured `npm install` to run bower for us.

Template

Our custom resource service will be defined in `app/js/services.js` so we need to include this file in our layout template. Additionally, we also need to load the `angular-resource.js` file, which contains the ngResource module:

`app/index.html`.

```
...
<script src="bower_components/angular-resource/angular-resource.js"></script>
<script src="js/services.js"></script>
...
```

Service

We create our own service to provide access to the phone data on the server:

app/js/services.js .

```
var phonecatServices = angular.module('phonecatServices', ['ngResource']);

phonecatServices.factory('Phone', ['$resource',
  function($resource){
    return $resource('phones/:phoneId.json', {}, {
      query: {method:'GET', params:{phoneId:'phones'}, isArray:true}
    });
  }]);
```

We used the module API to register a custom service using a factory function. We passed in the name of the service - 'Phone' - and the factory function. The factory function is similar to a controller's constructor in that both can declare dependencies to be injected via function arguments. The Phone service declared a dependency on the `$resource` service.

The `$resource` service makes it easy to create a RESTful client with just a few lines of code. This client can then be used in our application, instead of the lower-level $http service.

app/js/app.js .

```
...
angular.module('phonecatApp', ['ngRoute', 'phonecatControllers','phonecatFilters', 'phonecatServices']).
...
```

We need to add the 'phonecatServices' module dependency to 'phonecatApp' module's requires array.

Controller

We simplified our sub-controllers (`PhoneListCtrl` and `PhoneDetailCtrl`) by factoring out the lower-level $http service, replacing it with a new service called `Phone` . Angular's `$resource` service is easier to use than `$http` for interacting with data sources exposed as RESTful resources. It is also easier now to understand what the code in our controllers is doing.

app/js/controllers.js .

```
var phonecatControllers = angular.module('phonecatControllers', []);

...

phonecatControllers.controller('PhoneListCtrl', ['$scope', 'Phone', function($scope, Phone) {
  $scope.phones = Phone.query();
  $scope.orderProp = 'age';
}]);

phonecatControllers.controller('PhoneDetailCtrl', ['$scope', '$routeParams', 'Phone', function($scope,
$routeParams, Phone) {
  $scope.phone = Phone.get({phoneId: $routeParams.phoneId}, function(phone) {
    $scope.mainImageUrl = phone.images[0];
  });

  $scope.setImage = function(imageUrl) {
    $scope.mainImageUrl = imageUrl;
  }
}]);
```

Notice how in `PhoneListCtrl` we replaced:

```
$http.get('phones/phones.json').success(function(data) {
$scope.phones = data;
});
```

with:

```
$scope.phones = Phone.query();
```

This is a simple statement that we want to query for all phones.

An important thing to notice in the code above is that we don't pass any callback functions when invoking methods of our Phone service. Although it looks as if the result were returned synchronously, that is not the case at all. What is returned synchronously is a "future" — an object, which will be filled with data when the XHR response returns. Because of the data-binding in Angular, we can use this future and bind it to our template. Then, when the data arrives, the view will automatically update.

Sometimes, relying on the future object and data-binding alone is not sufficient to do everything we require, so in these cases, we can add a callback to process the server response. The `PhoneDetailCtrl` controller illustrates this by setting the `mainImageUrl` in a callback.

Test

Because we're now using the ngResource module, it's necessary to also need to update the Karma config file with angular-resource so the new tests will pass.

test/karma.conf.js :

```
files : [
  'app/bower_components/angular/angular.js',
  'app/bower_components/angular-route/angular-route.js',
  'app/bower_components/angular-resource/angular-resource.js',
  'app/bower_components/angular-mocks/angular-mocks.js',
  'app/js/**/*.js',
  'test/unit/**/*.js'
],
```

We have modified our unit tests to verify that our new service is issuing HTTP requests and processing them as expected. The tests also check that our controllers are interacting with the service correctly.

The $resource service augments the response object with methods for updating and deleting the resource. If we were to use the standard `toEqual` matcher, our tests would fail because the test values would not match the responses exactly. To solve the problem, we use a newly-defined `toEqualData` Jasmine matcher. When the `toEqualData` matcher compares two objects, it takes only object properties into account and ignores methods.

`test/unit/controllersSpec.js` :

```
describe('PhoneCat controllers', function() {

beforeEach(function(){
  this.addMatchers({
    toEqualData: function(expected) {
      return angular.equals(this.actual, expected);
    }
  });
});

beforeEach(module('phonecatApp'));
beforeEach(module('phonecatServices'));


describe('PhoneListCtrl', function(){
  var scope, ctrl, $httpBackend;

  beforeEach(inject(function(_$httpBackend_, $rootScope, $controller) {
    $httpBackend = _$httpBackend_;
    $httpBackend.expectGET('phones/phones.json').
        respond([{name: 'Nexus S'}, {name: 'Motorola DROID'}]);

    scope = $rootScope.$new();
    ctrl = $controller('PhoneListCtrl', {$scope: scope});
  }));


  it('should create "phones" model with 2 phones fetched from xhr', function() {
    expect(scope.phones).toEqualData([]);
    $httpBackend.flush();

    expect(scope.phones).toEqualData(
        [{name: 'Nexus S'}, {name: 'Motorola DROID'}]);
  });
```

```
  it('should set the default value of orderProp model', function() {
    expect(scope.orderProp).toBe('age');
  });
});


describe('PhoneDetailCtrl', function(){
  var scope, $httpBackend, ctrl,
      xyzPhoneData = function() {
        return {
          name: 'phone xyz',
          images: ['image/url1.png', 'image/url2.png']
        }
      };


  beforeEach(inject(function(_$httpBackend_, $rootScope, $routeParams, $controller) {
    $httpBackend = _$httpBackend_;
    $httpBackend.expectGET('phones/xyz.json').respond(xyzPhoneData());

    $routeParams.phoneId = 'xyz';
    scope = $rootScope.$new();
    ctrl = $controller('PhoneDetailCtrl', {$scope: scope});
  }));


  it('should fetch phone detail', function() {
    expect(scope.phone).toEqualData({});
    $httpBackend.flush();

    expect(scope.phone).toEqualData(xyzPhoneData());
  });
});
});
```

You should now see the following output in the Karma tab:

```
Chrome 22.0: Executed 4 of 4 SUCCESS (0.038 secs / 0.01 secs)
```

Summary

Now that we've seen how to build a custom service as a RESTful client, we're ready for step 12 (the last step!) to learn how to improve this application with animations.

Previous | Live Demo | Code Diff | Next