Enough of building an app with three phones in a hard-coded dataset! Let's fetch a larger dataset from our server using one of Angular's built-in services called $http. We will use Angular's dependency injection (DI) to provide the service to the `PhoneListCtrl` controller.

- There is now a list of 20 phones, loaded from the server.

Workspace Reset Instructions ➤

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

---

Data

The `app/phones/phones.json` file in your project is a dataset that contains a larger list of phones stored in the JSON format.

Following is a sample of the file:

```
[
{
 "age": 13,
 "id": "motorola-defy-with-motoblur",
 "name": "Motorola DEFY\u2122 with MOTOBLUR\u2122",
 "snippet": "Are you ready for everything life throws your way?"
 ...
},
...
]
```

Controller

We'll use Angular's $http service in our controller to make an HTTP request to your web server to fetch the data in the `app/phones/phones.json` file. `$http` is just one of several built-in Angular services that handle common operations in web apps. Angular injects these services for you where you need them.

Services are managed by Angular's DI subsystem. Dependency injection helps to make your web apps both well-structured (e.g., separate components for presentation, data, and control) and loosely coupled (dependencies between components are not resolved by the components themselves, but by the DI subsystem).

app/js/controllers.js:

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

phonecatApp.controller('PhoneListCtrl', function ($scope, $http) {
  $http.get('phones/phones.json').success(function(data) {
    $scope.phones = data;
  });

  $scope.orderProp = 'age';
});
```

`$http` makes an HTTP GET request to our web server, asking for `phones/phones.json` (the url is relative to our `index.html` file). The server responds by providing the data in the json file. (The response might just as well have been dynamically generated by a backend server. To the browser and our app they both look the same. For the sake of simplicity we used a json file in this tutorial.)

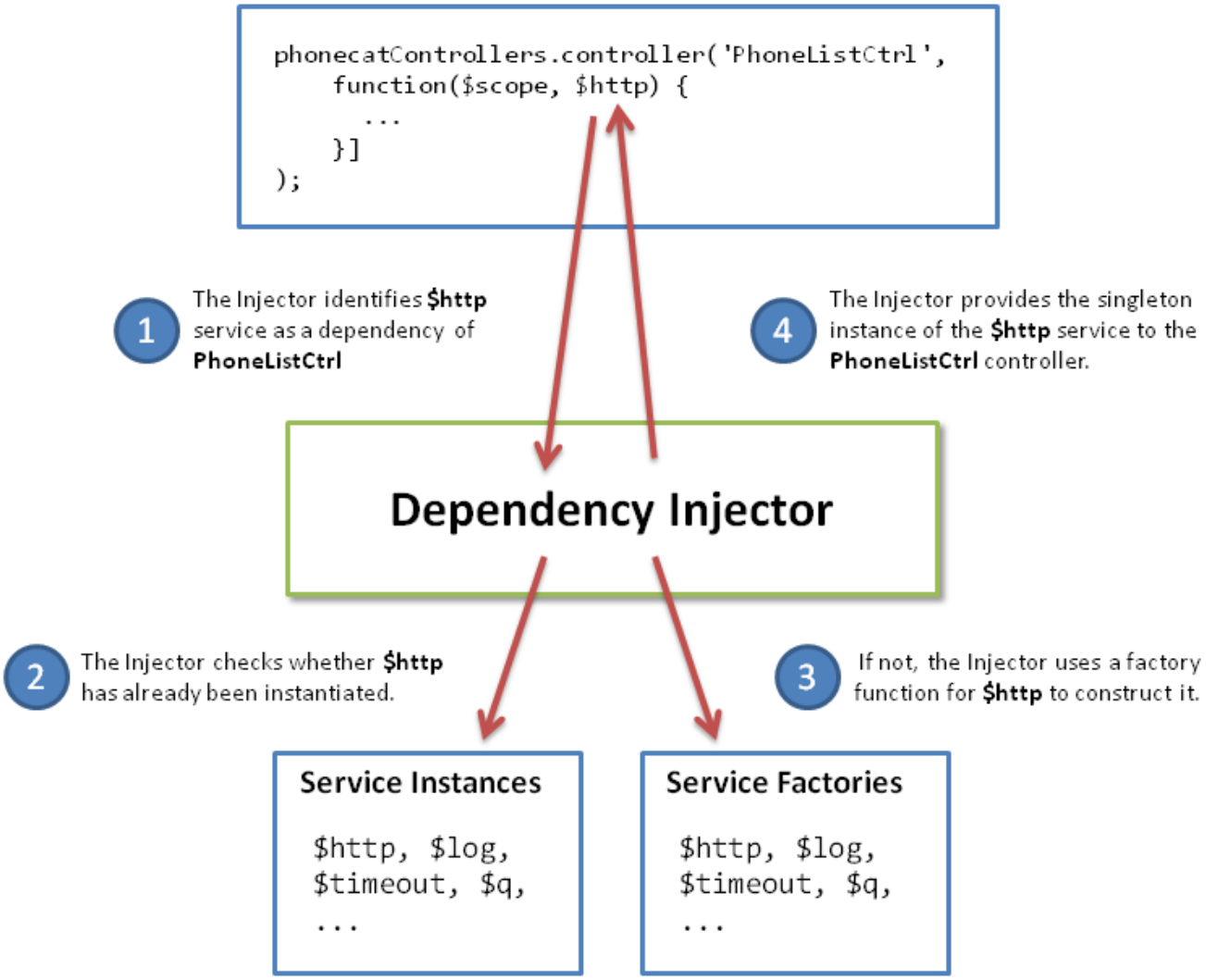
The `$http` service returns a promise object with a `success` method. We call this method to handle the asynchronous response and assign the phone data to the scope controlled by this controller, as a model called `phones`. Notice that Angular detected the json response and parsed it for us!

To use a service in Angular, you simply declare the names of the dependencies you need as arguments to the controller's constructor function, as follows:

```
phonecatApp.controller('PhoneListCtrl', function ($scope, $http) {...}
```

Angular's dependency injector provides services to your controller when the controller is being constructed. The dependency injector also takes care of creating any transitive dependencies the service may have (services often depend upon other services).

Note that the names of arguments are significant, because the injector uses these to look up the dependencies.



`$` Prefix Naming Convention

You can create your own services, and in fact we will do exactly that in step 11. As a naming convention, Angular's built-in services, Scope methods and a few other Angular APIs have a `$` prefix in front of the name.

The `$` prefix is there to namespace Angular-provided services. To prevent collisions it's best to avoid naming your services and models anything that begins with a `$`.

If you inspect a Scope, you may also notice some properties that begin with `$$`. These properties are considered private, and should not be accessed or modified.

A Note on Minification

Since Angular infers the controller's dependencies from the names of arguments to the controller's constructor function, if you were to minify the JavaScript code for `PhoneListCtrl` controller, all of its function arguments would be minified as well, and the dependency injector would not be able to identify services correctly.

We can overcome this problem by annotating the function with the names of the dependencies, provided as strings, which will not get minified. There are two ways to provide these injection annotations:

- Create a `$inject` property on the controller function which holds an array of strings. Each string in the array is the name of the service to inject for the corresponding parameter. In our example we would write:

  ```
  function PhoneListCtrl($scope, $http) {...}
  PhoneListCtrl.$inject = ['$scope', '$http'];
  phonecatApp.controller('PhoneListCtrl', PhoneListCtrl);
  ```

- Use an inline annotation where, instead of just providing the function, you provide an array. This array contains a list of

the service names, followed by the function itself.

```
function PhoneListCtrl($scope, $http) {...}
phonecatApp.controller('PhoneListCtrl', ['$scope', '$http', PhoneListCtrl]);
```

Both of these methods work with any function that can be injected by Angular, so it's up to your project's style guide to decide which one you use.

When using the second method, it is common to provide the constructor function inline as an anonymous function when registering the controller:

```
phonecatApp.controller('PhoneListCtrl', ['$scope', '$http', function($scope, $http) {...}]);
```

From this point onward, we're going to use the inline method in the tutorial. With that in mind, let's add the annotations to our `PhoneListCtrl`:

app/js/controllers.js:

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

phonecatApp.controller('PhoneListCtrl', ['$scope', '$http',
  function ($scope, $http) {
    $http.get('phones/phones.json').success(function(data) {
      $scope.phones = data;
    });

    $scope.orderProp = 'age';
  }]);
```

Test

test/unit/controllersSpec.js :

Because we started using dependency injection and our controller has dependencies, constructing the controller in our tests is a bit more complicated. We could use the `new` operator and provide the constructor with some kind of fake `$http` implementation. However, Angular provides a mock `$http` service that we can use in unit tests. We configure "fake" responses to server requests by calling methods on a service called `$httpBackend`:

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

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

  // Load our app module definition before each test.
  beforeEach(module('phonecatApp'));

  // The injector ignores leading and trailing underscores here (i.e. _$httpBackend_).
  // This allows us to inject a service but then attach it to a variable
  // with the same name as the service in order to avoid a name conflict.
  beforeEach(inject(function(_$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});
  }));
```

Note: Because we loaded Jasmine and `angular-mocks.js` in our test environment, we got two helper methods `module` and `inject` that we'll use to access and configure the injector.

We created the controller in the test environment, as follows:

- We used the `inject` helper method to inject instances of `$rootScope`, `$controller` and `$httpBackend` services into the Jasmine's `beforeEach` function. These instances come from an injector which is recreated from scratch for every single test. This guarantees that each test starts from a well known starting point and each test is isolated from the work done in other tests.
- We created a new scope for our controller by calling `$rootScope.$new()`
- We called the injected `$controller` function passing the name of the `PhoneListCtrl` controller and the created scope as parameters.

Because our code now uses the `$http` service to fetch the phone list data in our controller, before we create the `PhoneListCtrl` child scope, we need to tell the testing harness to expect an incoming request from the controller. To do this we:

- Request `$httpBackend` service to be injected into our `beforeEach` function. This is a mock version of the service that in a production environment facilitates all XHR and JSONP requests. The mock version of this service allows you to write tests without having to deal with native APIs and the global state associated with them — both of which make testing a nightmare.
- Use the `$httpBackend.expectGET` method to train the `$httpBackend` service to expect an incoming HTTP request and tell it what to respond with. Note that the responses are not returned until we call the `$httpBackend.flush` method.

Now we will make assertions to verify that the `phones` model doesn't exist on `scope` before the response is received:

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

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

- We flush the request queue in the browser by calling `$httpBackend.flush()` . This causes the promise returned by the `$http` service to be resolved with the trained response.
- We make the assertions, verifying that the phone model now exists on the scope.

Finally, we verify that the default value of `orderProp` is set correctly:

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

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

```
Chrome 22.0: Executed 2 of 2 SUCCESS (0.028 secs / 0.007 secs)
```

Experiments

At the bottom of `index.html` , add a `<pre>{{phones | json}}</pre>` binding to see the list of phones displayed in json format.

In the `PhoneListCtrl` controller, pre-process the http response by limiting the number of phones to the first 5 in the list. Use the following code in the `$http` callback:

```
$scope.phones = data.splice(0, 5);
```

Summary

Now that you have learned how easy it is to use Angular services (thanks to Angular's dependency injection), go to step 6, where you will add some thumbnail images of phones and some links.

Previous | Live Demo | Code Diff | Next

Super-powered by Google ©2010-2014 (v1.3.0 superluminal-nudge)

Back to top

Code licensed under the The MIT License. Documentation licensed under CC BY 3.0.