v1.5.6-build.4831 (snapshot

/ Tutorial (tutorial)
/ 7 - XHR & Dependency Injection (tutorial/step_07)

tent/tutorial/step_07.ngdoc?message=docs(tutorial%2F7 - XHR & Dependency Injection)%3A%20describe%20your%20change...)

Previous (tutorial/step_06) Live Demo (http://angular.github.io/angular-phonecat/step-7/app) Code Diff (https://github.com/angular/angular-phonecat/compare/step-6...step-7) Next (tutorial/step_08)

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 (guide/services) called $http (api/ng/service/$http). We will use Angular's dependency injection (DI) (guide/di) to provide the service to the `phoneList` component's 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 (https://github.com/angular/angular-phonecat/compare/step-6...step-7).

Data

The `app/phones/phones.json` file in our project is a dataset that contains a larger list of phones, stored in 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?"
    ...
  },
  ...
]
```

Component Controller

We will use Angular's $http (api/ng/service/$http) service in our controller for making an HTTP request to our web server to fetch the data in the `app/phones/phones.json` file. `$http` is just one of several built-in Angular services (guide/services) that handle common operations in web applications. Angular injects these services for you, right where you need them.

Services are managed by Angular's DI subsystem (guide/di). Dependency injection helps to make your web applications both well-structured (e.g. separate entities for presentation, data, and control) and loosely coupled (dependencies between entities are not resolved by the entities themselves, but by the DI subsystem). As a result, applications are easier to test as well.

app/phone-list/phone-list.component.js:

```
angular.
  module('phoneList').
  component('phoneList', {
    templateUrl: 'phone-list/phone-list.template.html',
    controller: function PhoneListController($http) {
      var self = this;
      self.orderProp = 'age';


$http.get('phones/phones.json').then(function(response) {
        self.phones = response.data;
      });
    }
  });
```

`$http` makes an HTTP GET request to our web server, asking for `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 will use JSON files in this tutorial.)

The `$http` service returns a promise object (api/ng/service/$q), which has a **`then()`** method. We call this method to handle the asynchronous response and assign the phone data to the controller, as a property called `phones`. Notice that Angular detected the JSON response and parsed it for us into the `data` property of the `response` object passed to our callback!

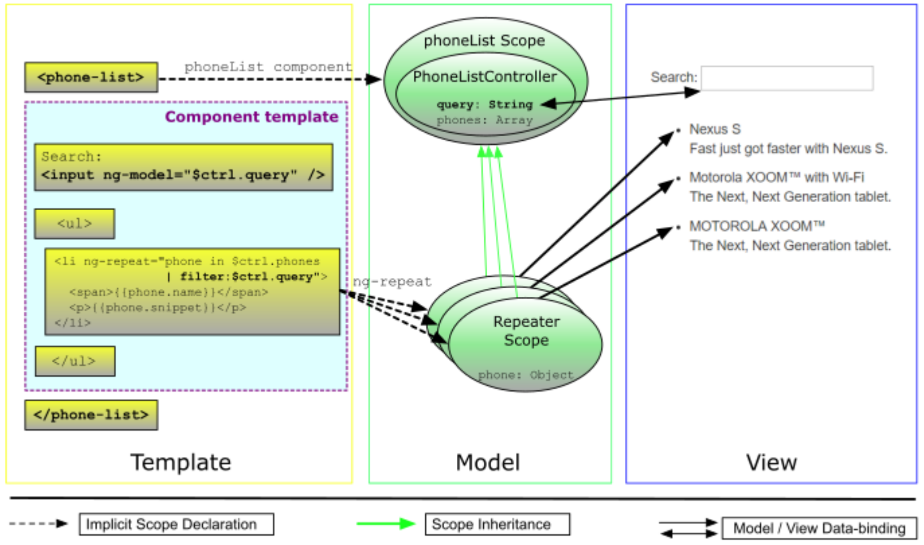
Since we are making the assignment of the `phones` property in a callback function, where the **`this`** value is not defined, we also introduce a local variable called **`self`** that points back to the controller instance.

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:

```
function PhoneListController($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.


<phone-list>
phoneList component
Component template
Search:
<input ng-model="$ctrl.query" />
<ul>
<li ng-repeat="phone in $ctrl.phones | filter:$ctrl.query">
<span>{{phone.name}}</span>
<p>{{phone.snippet}}</p>
</li>
</ul>
</phone-list>
ng-repeat
phoneList Scope
PhoneListController
query: String
phones: Array
Repeater Scope
phone: Object
Search:
Nexus S
Fast just got faster with Nexus S.
Motorola XOOM™ with Wi-Fi
The Next, Next Generation tablet.
MOTOROLA XOOM™
The Next, Next Generation tablet.
Template
Model
View
Implicit Scope Declaration
Scope Inheritance
Model / View Data-binding

$-prefix Naming Convention

You can create your own services, and in fact we will do exactly that a few steps down the road. 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 (https://en.wikipedia.org/wiki/Minification_(programming)) the JavaScript code for the **`PhoneListController`** 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 an `$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 PhoneListController($http) {...}
  PhoneListController.$inject = ['$http'];
  ...
  .component('phoneList', {..., controller:
  PhoneListController});
  ```

- 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 as the last item of the array.

```
function PhoneListController($http) {...}
...
.component('phoneList', {..., controller: ['$http',
PhoneListController]});
```

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, when registering the controller:

```
.component('phoneList', {..., controller: ['$http', function
PhoneListController($http) {...}]});
```

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

`app/phone-list/phone-list.component.js`

```
angular.
  module('phoneList').
  component('phoneList', {
    templateUrl: 'phone-list/phone-list.template.html',
    controller: ['$http',
      function PhoneListController($http) {
        var self = this;
        self.orderProp = 'age';


$http.get('phones/phones.json').then(function(response) {
          self.phones = response.data;
        });
      }
    ]
  });
```

Testing

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`:

`app/phone-list/phone-list.component.spec.js`:

```
describe('phoneList', function() {

  beforeEach(module('phoneList'));

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

    // The injector ignores leading and trailing underscores
here (i.e. _$httpBackend_).
    // This allows us to inject a service and assign it to a
variable with the same name
    // as the service while avoiding a name conflict.
    beforeEach(inject(function($componentController,
_$httpBackend_) {
      $httpBackend = _$httpBackend_;
      $httpBackend.expectGET('phones/phones.json')
                  .respond([{name: 'Nexus S'}, {name:
'Motorola DROID'}]);

      ctrl = $componentController('phoneList');
    }));

    ...

  });

});
```

Note: Because we loaded Jasmine and `angular-mocks.js` in our test environment, we got two helper methods module (api/ngMock/function/angular.mock.module) and inject (api/ngMock/function/angular.mock.inject) that we can 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 $componentController (api/ngMock/service/$componentController) and $httpBackend (api/ng/service/$httpBackend) services into 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 called the injected `$componentController` function passing the name of the `phoneList` component (whose controller we wanted to instantiate) as a parameter.

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

- Inject the `$httpBackend` service into the `beforeEach()` function. This is a mock version (api/ngMock/service/$httpBackend) of the service that in a production environment facilitates all XHR and JSONP requests. The mock version of this service allows us to write tests without having to deal with native APIs and the global state associated with them — both of which make testing a nightmare. It also overcomes the asynchronous nature of these calls, which would slow down unit tests.
- 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` property doesn't exist on the controller before the response is received:

```
it('should create a `phones` property with 2 phones fetched
with `$http`', function() {
  expect(ctrl.phones).toBeUndefined();

  $httpBackend.flush();
  expect(ctrl.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. See Flushing HTTP requests (api/ngMock/service/$httpBackend#flushing-http-requests) in the mock `$httpBackend` documentation for a full explanation of why this is necessary.
- We make the assertions, verifying that the `phones` property now exists on the controller.

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

```
it('should set a default value for the `orderProp`
property', function() {
  expect(ctrl.orderProp).toBe('age');
});
```

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

```
Chrome 49.0: Executed 2 of 2 SUCCESS (0.133 secs / 0.097
secs)
```

Experiments

- At the bottom of `phone-list.template.html` , add a `<pre>{{$ctrl.phones | filter:$ctrl.query | orderBy:$ctrl.orderProp | json}}</pre` binding to see the list of phones displayed in JSON format.

- In the **`PhoneListController`** 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:

```
self.phones = response.data.slice(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 8 (tutorial/step_08), where you will add some thumbnail images of phones and some links.

Previous (tutorial/step_06) **Live Demo** (http://angular.github.io/angular-phonecat/step-7/app) **Code Diff** (https://github.com/angular/angular-phonecat/compare/step-6...step-7)