Now it's time to make the web page dynamic — with AngularJS. We'll also add a test that verifies the code for the controller we are going to add.

There are many ways to structure the code for an application. For Angular apps, we encourage the use of the Model-View-Controller (MVC) design pattern to decouple the code and to separate concerns. With that in mind, let's use a little Angular and JavaScript to add model, view, and controller components to our app.

- The list of three phones is now generated dynamically from data

Workspace Reset Instructions ➤

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

View and Template

In Angular, the **view** is a projection of the model through the HTML **template**. This means that whenever the model changes, Angular refreshes the appropriate binding points, which updates the view.

The view component is constructed by Angular from this template:

app/index.html :

```
<html ng-app="phonecatApp">
<head>
  ...
  <script src="bower_components/angular/angular.js"></script>
  <script src="js/controllers.js"></script>
</head>
<body ng-controller="PhoneListCtrl">

  <ul>
    <li ng-repeat="phone in phones">
      {{phone.name}}
      <p>{{phone.snippet}}</p>
    </li>
  </ul>

</body>
</html>
```

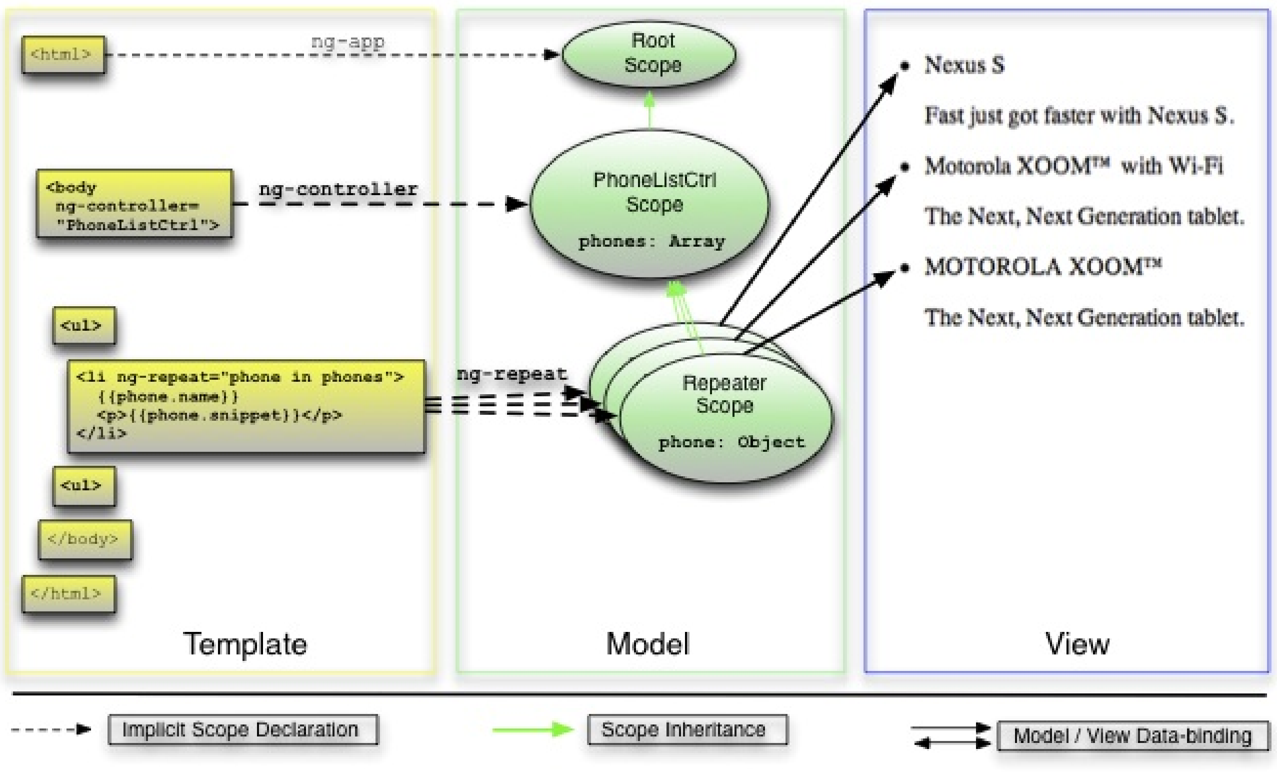
We replaced the hard-coded phone list with the ngRepeat directive and two Angular expressions:

- The `ng-repeat="phone in phones"` attribute in the `<li>` tag is an Angular repeater directive. The repeater tells Angular to create a `<li>` element for each phone in the list using the `<li>` tag as the template.
- The expressions wrapped in curly braces (`{{phone.name}}` and `{{phone.snippet}}`) will be replaced by the value of the expressions.

We have added a new directive, called `ng-controller`, which attaches a `PhoneListCtrl` **controller** to the <body> tag. At this point:

- The expressions in curly braces (`{{phone.name}}` and `{{phone.snippet}}` denote bindings, which are referring to our application model, which is set up in our `PhoneListCtrl` controller.

Note: We have specified an Angular Module to load using `ng-app="phonecatApp"`, where `phonecatApp` is the name of our module. This module will contain the `PhoneListCtrl`.



<html>
ng-app
Root Scope
<body ng-controller="PhoneListCtrl">
ng-controller
PhoneListCtrl Scope
phones: Array
<ul>
<li ng-repeat="phone in phones"> {{phone.name}} <p>{{phone.snippet}}</p> </li>
ng-repeat
Repeater Scope
phone: Object
<ul>
</body>
</html>
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

Model and Controller

The data **model** (a simple array of phones in object literal notation) is now instantiated within the `PhoneListCtrl` **controller**. The **controller** is simply a constructor function that takes a `$scope` parameter:

app/js/controllers.js :

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

phonecatApp.controller('PhoneListCtrl', function ($scope) {
  $scope.phones = [
    {'name': 'Nexus S',
     'snippet': 'Fast just got faster with Nexus S.'},
    {'name': 'Motorola XOOM™ with Wi-Fi',
     'snippet': 'The Next, Next Generation tablet.'},
    {'name': 'MOTOROLA XOOM™',
     'snippet': 'The Next, Next Generation tablet.'}
  ];
});
```

Here we declared a controller called `PhoneListCtrl` and registered it in an AngularJS module, `phonecatApp`. Notice that our `ng-app` directive (on the `<html>` tag) now specifies the `phonecatApp` module name as the module to load when bootstrapping the Angular application.

Although the controller is not yet doing very much, it plays a crucial role. By providing context for our data model, the controller allows us to establish data-binding between the model and the view. We connected the dots between the presentation, data, and logic components as follows:

- The ngController directive, located on the `<body>` tag, references the name of our controller, `PhoneListCtrl` (located in the JavaScript file `controllers.js`).

- The `PhoneListCtrl` controller attaches the phone data to the `$scope` that was injected into our controller function. This *scope* is a prototypical descendant of the *root scope* that was created when the application was defined. This controller scope is available to all bindings located within the `<body ng-controller="PhoneListCtrl">` tag.

Scope

The concept of a scope in Angular is crucial. A scope can be seen as the glue which allows the template, model and controller to work together. Angular uses scopes, along with the information contained in the template, data model, and controller, to keep models and views separate, but in sync. Any changes made to the model are reflected in the view; any changes that occur in the view are reflected in the model.

To learn more about Angular scopes, see the angular scope documentation.

Tests

The "Angular way" of separating controller from the view, makes it easy to test code as it is being developed. If our controller is available on the global namespace then we could simply instantiate it with a mock `scope` object:

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

it('should create "phones" model with 3 phones', function() {
  var scope = {},
      ctrl = new PhoneListCtrl(scope);

  expect(scope.phones.length).toBe(3);
});

});
```

The test instantiates `PhoneListCtrl` and verifies that the phones array property on the scope contains three records. This example demonstrates how easy it is to create a unit test for code in Angular. Since testing is such a critical part of software development, we make it easy to create tests in Angular so that developers are encouraged to write them.

Testing non-Global Controllers

In practice, you will not want to have your controller functions in the global namespace. Instead, you can see that we have registered it via an anonymous constructor function on the `phonecatApp` module.

In this case Angular provides a service, `$controller`, which will retrieve your controller by name. Here is the same test using `$controller`:

test/unit/controllersSpec.js :

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

beforeEach(module('phonecatApp'));

it('should create "phones" model with 3 phones', inject(function($controller) {
  var scope = {},
      ctrl = $controller('PhoneListCtrl', {$scope:scope});

  expect(scope.phones.length).toBe(3);
}));

});
```

- Before each test we tell Angular to load the `phonecatApp` module.
- We ask Angular to `inject` the `$controller` service into our test function
- We use `$controller` to create an instance of the `PhoneListCtrl`
- With this instance, we verify that the phones array property on the scope contains three records.

Writing and Running Tests

Angular developers prefer the syntax of Jasmine's Behavior-driven Development (BDD) framework when writing tests. Although Angular does not require you to use Jasmine, we wrote all of the tests in this tutorial in Jasmine v1.3. You can learn about Jasmine on the Jasmine home page and at the Jasmine docs.

The angular-seed project is pre-configured to run unit tests using Karma but you will need to ensure that Karma and its necessary plugins are installed. You can do this by running `npm install`.

To run the tests, and then watch the files for changes: `npm test`.

- Karma will start a new instance of Chrome browser automatically. Just ignore it and let it run in the background. Karma will use this browser for test execution.
- You should see the following or similar output in the terminal:

  ```
  info: Karma server started at http://localhost:9876/
  info (launcher): Starting  browser "Chrome"
  info (Chrome 22.0): Connected on socket id tPUm9DXcLHtZTKbAEO-n
  Chrome 22.0: Executed 1 of 1 SUCCESS (0.093 secs / 0.004 secs)
  ```

  Yay! The test passed! Or not...

- To rerun the tests, just change any of the source or test .js files. Karma will notice the change and will rerun the tests for you. Now isn't that sweet?

Make sure you don't minimize the browser that Karma opened. On some OS, memory assigned to a minimized browser is limited, which results in your karma tests running extremely slow.

Experiments

Add another binding to `index.html`. For example:

```
<p>Total number of phones: {{phones.length}}</p>
```

Create a new model property in the controller and bind to it from the template. For example:

```
$scope.name = "World";
```

Then add a new binding to `index.html`:

```
<p>Hello, {{name}}!</p>
```

Refresh your browser and verify that it says "Hello, World!".

Update the unit test for the controller in `./test/unit/controllersSpec.js` to reflect the previous change. For example by adding:

```
expect(scope.name).toBe('World');
```

Create a repeater in `index.html` that constructs a simple table:

```
<table>
  <tr><th>row number</th></tr>
  <tr ng-repeat="i in [0, 1, 2, 3, 4, 5, 6, 7]"><td>{{i}}</td></tr>
</table>
```

Now, make the list 1-based by incrementing `i` by one in the binding:

```
<table>
  <tr><th>row number</th></tr>
  <tr ng-repeat="i in [0, 1, 2, 3, 4, 5, 6, 7]"><td>{{i+1}}</td></tr>
</table>
```

Extra points: try and make an 8x8 table using an additional `ng-repeat`.

Make the unit test fail by changing `expect(scope.phones.length).toBe(3)` to instead use `toBe(4)`.

Summary

You now have a dynamic app that features separate model, view, and controller components, and you are testing as you go. Now, let's go to step 3 to learn how to add full text search to the app.

Previous   Live Demo   Code Diff   Next