Recipes with Angular.js

Practical concepts and techniques for rapid web application development

```
<body ng-app>
  <div ng-controller="MyCtrl">
    <button class="btn" ng-click="toggle()">Toggle</button>
    <p ng-show="isVisible()">Hello World!</p>
    <p>Debug Scope: visible = {{visible}}</p>
  </div>
</body>
```

by Frederik Dietz

version 0.1

Recipes with Angular.js

Practical concepts and techniques for rapid web application development

Frederik Dietz

This book is for sale at http://leanpub.com/recipes-with-angular-js

This version was published on 2013-01-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

Introduction

Angular.js 1.0 has been released only half a year ago but is already changing the development status quo of client-side web apps. With its focus on CRUD based applications it achieves a very high productivity unmatched by other frameworks. If you are using Angular.js, or considering it, this cookbook provides easy to follow recipes for issues you are likely to face.

Each recipe solves a specific problem and provides a solution and in-depth discussion of it.

Code Examples

All code examples in this book can be found on http://github.com/fdietz/recipes-with-angular.js

How to contact us

If you have questions or comments please get in touch with:

Frederik Dietz

fdietz@gmail.com

Acknowledgements

Thanks go to bla for reviewing the book!

An Introduction to Angular.js

Including angular.js in a web page

Problem

You want to include Angular.js in a web page.

Solution

In order to get your first Angular.js app up and running you need to include the angular javascript file via script tag and make use of the `ng-app` directive.

```
<html>
  <head>
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.0.4/angul\
ar.js"></script>
  </head>
  <body ng-app>
    <p>This is your first angular expression: {{ 1 + 2 }}</p>
  </body>
</html>
```

Tip: You can checkout a complete example on github[a].

[a] http://github.com/fdietz/recipes-with-angular.js/chapter1/recipe1

Discussion

Adding the `ng-app` directive tells Angular to kick in its magic. The expression `{{ 1 + 2 }}` is evaluated by Angular and the result `3` is rendered. Note, that removing `ng-app` will result in the browser to render the expression as is instead of evaluating it. Play around with the expression! You can use numbers as in the example or concatenate Strings, etc.

For brevity reasons we skip the boilerplate code in the following recipes.

Binding a text input to an expression

Problem

You want user input to be used in another part of your html page.

Solution

Use Angulars `ng-model` directive to bind the text input to the expression

```
Enter your name: <input type="text" ng-model="name"></input>
<p>Hello {{name}}!</p>
```

Discussion

Assigning "name" to the `ng-model` attribute and using the name variable in an expression will keep both in sync automatically. Typing in the text input will automatically reflect these changes in the paragraph below.

Consider how you would implement this traditionally using jQuery:

```
<html>
  <head>
    <script src="http://code.jquery.com/jquery.min.js"></script>
  </head>
  <body>
    Enter your name: <input type="text"></input>
    <p id="name"></p>

    <script>
      $(document).ready(function() {
        $("input").keypress(function() {
          $("#name").text($(this).val());
        });
      });
    </script>

  </body>
</html>
```

On document ready we bind to the keypress event in the text input and replace the text in the paragraph in the callback function. Using jQuery you need to deal with document ready callbacks, element selection, event binding and the context of this. Quite a lot of concepts to swallow and lines of code to maintain!

Convert expression output with Filters

Problem

When presenting data to the user, you might need to convert the data to a more user-friendly format. In our case we want to uppercase the "name" value from the previous recipe in the expression.

Solution

Use the `uppercase` Angular filter.

```
1 Enter your name: <input type="text" ng-model="name"></input>
2 <p>Hello {{name | uppercase }}!</p>
```

Discussion

Angular uses the `|` (pipe) character to combine filters with variables in expressions. When evaluating the expression, the name variable is passed to the uppercase filter. This is similar to working with the Unix bash pipe symbol where an input can be transformed by another program. Also try the `lowercase` filter!

Use Controllers for your business logic

Problem

You want to hide an html element on button click.

Solution

Use the `ng-hide` directive in conjunction with a controller to change the visibility status on button click.

```
<html>
  <head>
    <script src="js/angular.js"></script>
    <script src="js/app.js"></script>
    <link rel="stylesheet" href="css/bootstrap.css">
  </head>
  <body ng-app>
    <div ng-controller="MyCtrl">
      <button ng-click="toggle()">Toggle</button>
      <p ng-show="isVisible()">Hello World!</p>
      <p>Debug Scope: visible = {{visible}}</p>
    </div>
  </body>
</html>
```

And the controller in `js/app.js`:

```
function MyCtrl($scope) {
  $scope.visible = true;

  $scope.toggle = function() {
    $scope.visible = !$scope.visible;
  };

  $scope.isVisible = function() {
    return $scope.visible === true;
  };
}
```

Discussion

Using the `ng-controller` directive we bind the div element including its children to the context of the `MyCtrl` Controller. The `ng-click` directive will call the `toggle()` function of the `MyCtrl` Controller on button click. The controller implementation defaults the `visible` attribute to true and toggles its boolean state in the toggle function. The `ng-show` directive calls the `isVisible()` function to retrieve the boolean state. Note, that you could use the `visible` attribute instead if `isVisible()`. Using a function encapsulates the logic and allows more complex logic.

Create your own directive

Problem

You want to render a Hello World snippet in several places.

Solution

Create a custom directive which renders your Hello World snippet.

```
<body ng-app="MyApp">
  <hello-world/>
</body>
```

The directive implementation:

```
var app = angular.module("MyApp", []);

app.directive("helloWorld", function() {
  return {
    restrict: "E",
    template: '<span>Hello World</span>'
  };
});
```

Discussion

We ignore the module creation for a later recipe for now. The browser will render the `span` element as defined in the directive. Note, that it did not replace the `hello-world` element, but instead inserted the `span`. If you want to replace the content completely you need to return an addditional attribute replace set to the true:

```
app.directive("helloWorld", function() {
  return {
    restrict: "E",
    replace: true,
    template: '<span>Hello World</span>'
  };
});
```

Now the `hello-world` element is not rendered at all and replaced with the `span` element.

Also note the `restrict` attribute is set to `E` which means the directive can be used only as an html element. A full discussion will follow in later chapters.

Controllers

Controllers in Angular handle view behaviour. The user clicking a button or entering some text in a form - what should happen next is implemented in a controller. As a general rule a controller should not reference the DOM directly. This dramatically simplifies unit testing controllers.

Assign default value to model

Problem

You want to assign a default value to the scope in the controllers context.

Solution

```
<div ng-controller="MyCtrl">
  <p>{{value}}</p>
</div>

var MyCtrl = function($scope) {
  $scope.value = "some value";
};
```

Discussion

Depending on where you use the ng-controller directive, you define its assigned scope. The scope is hierachical and follows the DOM nodes hierarchy. In our example the value expression is correctly evaluated to `some value`, since value is set in the `MyCtrl` controller. Note, that this would not work if the value expression is moved outside the controllers scope:

```
<p>{{value}}</p>

<div ng-controller="MyCtrl">
</div>
```

In this case `{{value}}` will simply be not rendered at all.

Change model value with a function

Problem

You want to increment the model value by 1.

Solution

Define a increment function which changes the scope.

```
<div ng-controller="MyCtrl">
  <p ng-init="incrementValue(5)">{{value}}</p>
</div>

function MyCtrl($scope) {
  $scope.value = 1;

  $scope.incrementValue = function(value) {
    $scope.value += 1;
  };
}
```

Discussion

The `ng-init` directive is executed on page load and calls the function defined in `MyCtrl`.

Expose model value with a function

Problem

You want to retrieve a model via function (instead of directly accessing the scope from the template) which further changes the model value.

Solution

Define a getter function which returns the model value.

```
<div ng-controller="MyCtrl">
  <p>{{getIncrementedValue()}}</p>
</div>

function MyCtrl($scope) {
  $scope.value = 1;

  $scope.getIncrementedValue = function() {
    return $scope.value + 1;
  };
}
```

Discussion

`MyCtrl` defines the `getIncrementedValue` function which uses the current value and returns it incremented by one. One could argue that depending on the use case it would make more sense to use a filter. But, there are use cases specific to the controllers behaviour which you might not want to use a generic directive.

Respond to scope changes

Problem

You want to react on a model change to trigger some further actions. In our example we simple want to set another model value depending on the value we are listing on.

Solution

Use the $watch function in your controller.

```
<div ng-controller="MyCtrl">
  <input type="text" ng-model="name" placeholder="Enter your name">
  <p>{{greeting}}</p>
</div>

function MyCtrl($scope) {
  $scope.name = "";

  $scope.$watch("name", function(newValue, oldValue) {
    if ($scope.name.length > 0) {
      $scope.greeting = "Greetings " + $scope.name;
    }
  });
}
```

The value `greeting` will be changed whenever there's a change on the `name` model and the value is not blank.

Discussion

The first argument `name` of the `$watch` function is actually an Angular expression, so you can use more complicated expressions (for example: `[value1, value2] | json`). Alternatively, you can also use a javascript function:

```
$scope.$watch(function() {
  return $scope.name;
}, function(newValue, oldValue) {
  console.log("change detected: " + newValue)
});
```

Note, that you need to return a String in the watcher function. The second function will only be called if the returned String changed compared to the previous execution. Internally this uses `angular.equals` to determine equality.

Testing Controllers

Problem

You want to unit test your business logic.

Solution

Implement a unit test using Jasmine[1] and the angular-seed[2] bootstrap. Following our previous `$watch` recipe this is how our spec would look like.

```
describe('MyCtrl', function(){
  var scope, ctrl;

  beforeEach(inject(function($injector, $controller, $rootScope) {
    scope = $rootScope.$new();
    ctrl = $controller(MyCtrl, { $scope: scope });
  }));

  it('should change greeting value if name value is changed', function() {
    scope.name = "Frederik";
    scope.$digest();
    expect(scope.greeting).toBe("Greetings Frederik");
  });
});
```

Discussion

Jasmine specs use `describe` and `it` functions to group specs and `beforeEach` and `afterEach` to setup and teardown code. The actual expectation compares the greeting from the scope with our expectation `Greetings Frederik`.

The scope and controller initialization is a bit more involved. We use `inject` to initialize the scope and controller as closely as possible to how our code would behave at runtime too. We can't just initialize the scope as an javascript object `{}` since then we would not be able to call `$watch` on it.

The `$digest` call is required in order for another watch execution. We need to call `$digest` manually in our spec whereas at runtime Angular will do this for us automatically.

[1] http://pivotal.github.com/jasmine/

[2] https://github.com/angular/angular-seed

Directives

Directives are one of the most powerful concepts in Angular since they let you invent new html syntax specific to your application. This allows you to create reusable components which encapsulate complex DOM structures, stylesheets and even behaviour.

Enable/Disable DOM element conditionally

Problem

You want to disable a button depending on a checkbox state.

Solution

Use the `ng-disabled` directive and bind its condition to the checkbox state.

```
<body ng-app>
  <label><input type="checkbox" ng-model="checked"/>Toggle Button</label>
  <button ng-disabled="checked">Press me</button>
</body>
```

Discussion

The `ng-disabled` directive is a direct translation from the disabled HTML attribute, without you needing to worry about browser incompatibilities. It is bound to the `checked` model using an attribute value as is the checkbox using the `ng-model` directive. In fact the `checked` attribute value is again an Angular expression. You could for example invert the logic and use `!checked` instead.

This is just one example of a directive shipped with Angular. There are many others as for example `ng-hide`, `ng-checked` or `ng-mouseenter` and I encourage you go through the API Reference[3] and explore all the directives Angular has to offer.

In the next recipes we will focus on implementing directives.

[3] http://docs.angularjs.org/api

Implement DOM changes in response to user behaviour

Problem

You want to change the css of an HTML element on a mouse click and encapsulate this behaviour in a reusable component.

Solution

Implement a directive with defines a link function.

```
<body ng-app="MyApp">
  <my-widget>
    <p>Hello World</p>
  </my-widget>
</body>


var app = angular.module("MyApp", []);

app.directive("myWidget", function() {
  var linkFunction = function(scope, element, attributes) {
    var paragraph = element.children()[0];
    $(paragraph).on("click", function() {
      $(this).css({ "background-color": "red" });
    });
  };

  return {
    restrict: "E",
    link: linkFunction
  };
});
```

When clicking on the paragraph the background color changes to red.

Discussion

In the HTML document we use the new directive as an HTML element `my-widget`, which can be found in the javascript code as `myWidget` again. The directive function returns a restriction and a

link function.

The restriction means that this directive can only be used as an HTML element and not for example an HTML attribute. If you want to use it as an HTML attribute, change the `restrict` to return `A` instead. The usage would then have to be adapted to:

```
<div my-widget>
  <p>Hello World</p>
</div>
```

Wether you use the attribute or element mechanism depends on your use case. Generally speaking one would use the element mechanism to define a custom reusable component. The attribute mechanism would be used whenever you want to "configure" some element or enhance it with more behaviour. Other options are using the directive as a class attribute or a comment.

The link function is much more interesting since it defines the actual behaviour. The scope, the actual HTML element `my-widget` and the html attributes are passed as params. Note, that this has nothing to do with Angulars dependency injection mechanism. Ordering of the parameters is important!

First we select the paragraph element, which is a child of the `my-widget` element using Angulars `children()` function as defined by element. In the second step we use jQuery to bind to the click event and modify the css property on click. This is of special interest since we have a mixture of Angular element functions and jQuery here. In fact under the hood Angular will use jQuery in the `children()` function if its defined and will fall back to jqLite (shipped with Angular) otherwise. You can find all supported methods in the API Reference of element[4].

Following a slightly changed version of the code using jQuery only:

```
element.on("click", function() {
  $(this).css({ "background-color": "red" });
});
```

In this case `element` is alreay an jQuery element and we can directly use the on function.

Render an html snippet

Problem

You want to render a html snippet in multiple places

[4] http://docs.angularjs.org/api/angular.element

Solution

Implement a directive and use the template attribute to define the HTML.

```
<body ng-app="MyApp">
  <my-widget/>
</body>

var app = angular.module("MyApp", []);

app.directive("myWidget", function() {
  return {
    restrict: "E",
    template: "<p>Hello World</p>"
  };
});
```

Discussion

This will render the Hello World paragraph as a child node of your `my-widget` element. In case you want to replace the element entirely with the paragraph you have to additionally return the replace attribute:

```
app.directive("myWidget", function() {
  return {
    restrict: "E",
    replace: true,
    template: "<p>Hello World</p>"
  };
});
```

Another option would be to use a file for the HTML snippet. In this case you need to use the templateUrl attribute, as for example:

```
app.directive("myWidget", function() {
  return {
    restrict: "E",
    replace: true,
    templateUrl: "widget.html"
  };
});
```

The `widget.html` should reside in the same directory as the `index.html` file. This will only work if you use a web server to host the file. The example on Github uses angular-seed as bootstrap again.

Use directives child nodes in directive output

Problem

Your widget uses the child nodes of the directive element to create a combined rendering.

Solution

Use the `transclude` attribute together with the `ng-transclude` directive.

```
<my-widget>
  <p>This is my paragraph text.</p>
</my-widget>

var app = angular.module("MyApp", []);

app.directive("myWidget", function() {
  return {
    restrict: "E",
    transclude: true,
    template: "<div ng-transclude><h3>Heading</h3></div>"
  };
});
```

This will render a `div` element containing a `h3` element and append the directives child node with the paragraph element below.

Discussion

In this context transclusion refers to the inclusion of a part of a document into another document by reference. The `ng-transclude` attribute should be placed depending on where you want your child nodes to be appended to.

Pass configuration params using HTML attributes

Problem

You want to pass a configuration param to change the rendered output.

Solution

Use the attribute based directive and pass an attribute value for the configuration. The attribute is passed as a parameter to the link function.

```
<body ng-app="MyApp">
  <div my-widget="Hello World"></div>
</body>

var app = angular.module("MyApp", []);

app.directive("myWidget", function() {
  var linkFunction = function(scope, element, attributes) {
    scope.text = attributes["myWidget"];
  };

  return {
    restrict: "A",
    template: "<p>{{text}}</div>",
    link: linkFunction
  };
});
```

Discussion

The link function has access to the element and its attributes. It is therefore straight forward to set the scope to the text passed as the attributes value and use this in the template evaluation.

The scope context is important though. The `text` scope we changed might have changed and existing model used in another place of your app. In order to isolate the context and thereby using it only locally inside of your directive we have to return an additional scope attribute.

```
return {
  restrict: "A",
  template: "<p>{{text}}</div>",
  link: linkFunction,
  scope: {}
};
```

Render an HTML snippet repeatedly

Problem

You want to render an html snippet repeatedly using the directives child nodes as the "stamp" content.

Solution

Implement a compile function in your directive.

```
<repeat-ntimes repeat="10">
  <h1>Header 1</h1>
  <p>This is the paragraph.</p>
</repeat-n-times>

var app = angular.module("MyApp", []);

app.directive("repeatNtimes", function() {
  return {
    restrict: "E",
    compile: function(tElement, attrs) {
      var content = tElement.children();
      for (var i=1; i<attrs.repeat; i++) {
        tElement.append(content.clone());
      }
    }
  };
});
```

This will render the header and paragraph 10 times.

Discussion

The directive repeats the child nodes as often as configured in the `repeat` attribute. It works similarly to the ng-repeat[5] directive. The implemention uses Angulars element methods to append the child nodes in a for loop.

Note, that the compile method only has access to the templates element (tElement) and template attributes. It has no access to the scope and you therefore also can't use `$watch` to add behaviour. This is in comparison to the link function which has access to the DOM "instance" (after the compile phase) and has access to the scope to add behaviour.

Use the compile function for template DOM manipulation only. Use the link function when you want to add behaviour.

Note, that you can use both compile and link function combined. In this case the compile function must return the link function. As an example you want to react to a click on the header:

```
1  compile: function(tElement, attrs) {
2    var content = tElement.children();
3    for (var i=1; i<attrs.repeat; i++) {
4      tElement.append(content.clone());
5    }
6
7    return function (scope, element, attrs) {
8      element.on("click", "h1", function() {
9        $(this).css({ "background-color": "red" });
10     });
11   };
12 }
```

Testing directives

[5] http://docs.angularjs.org/api/ng.directive:ngRepeat

Filters

Transform String to lowercase/uppercase before rendering

Implement filter to reverse string output

Combine multiple filters in a filter chain

Format string with a currency filter

Format numeric input with decimal marks

create format date/time filter using moment.js

filter with argument

test filter implementation

Testing Filters

Services

Reuse code between Controllers using Services

Use $http to do low-level http requests

Change http request headers

Use $resource for RESTful APIs calls (Using mongolab as example service)

Doing JSONP calls

Testing Services

Routing and Partials

Forms

Common User Interface Patterns

Debugging and Profiling

Backend Integration

Rails

Node.js