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

beta version

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 2013-02-15

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.

Tweet This Book!

Please help Frederik Dietz by spreading the word about this book on Twitter!

The suggested hashtag for this book is #recipeswithangularjs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#recipeswithangularjs

Contents

Preface

Introduction

Angular.js is an open-source Javascript MVC (Model-View-Controller) framework developed by Google. It gives Javascript developers a highly structured approach to developing rich browser-based applications which leads to very high productivity.

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 Github[1].

How to contact me

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

Frederik Dietz (fdietz@gmail.com)

Acknowledgements

Thanks go to John Lindquist for his excellent Angular.js screencasts[2] and his project Egghead IO[3], Lukas Ruebbelke for his awesome videos[4]

TODO: Thanks for reviewing the book!

[1] http://github.com/fdietz/recipes-with-angular-js-examples
[2] http://www.youtube.com/user/johnlindquist/videos?query=angular
[3] http://egghead.io/
[4] http://www.youtube.com/user/simpulton/videos?flow=grid&view=0

An Introduction to Angular.js

Including the Angular.js library Code in an HTML page

Problem

You want to use Angular.js on 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-examples/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 for example concatenate Strings and invert or combine Boolean values.

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 element 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!

Converting 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!

Responding to Click Events using Controllers

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 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 also use the `visible` attribute directly if you don't need to encapsulate your business logic.

Creating Custom HTML elements with Directives

Problem

You want to render an HTML snippet as a reusable custom HTML element.

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 does not replace the `hello-world` element, but instead inserts the `span` as a child. If you want to replace the content completely you need to return an additional attribute `replace` set to 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 a later chapter.

Controllers

Controllers in Angular handle view behaviour, for example 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.

Assigning a Default Value to a 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.

Changing a Model Value with a Controller Function

Problem

You want to increment a model value by 1.

Solution

Implement an 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`.

Encapsulating a Model Value with a Controller 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.

Responding 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) {
```

```
11        $scope.greeting = "Greetings " + $scope.name;
12      }
13    });
14  }
```

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.

Sharing Models Between Nested Controllers

Problem

You want to share a model between a nested hierarchy of controllers.

Solution

Use javascript objects instead of primitives or direct `$parent` references.

The example template uses a controller and a nested controller `MyNestedCtrl`:

```
<body ng-app="MyApp">
  <div ng-controller="MyCtrl">
    <label>Primitive</label>
    <input type="text" ng-model="name">

    <label>Object</label>
    <input type="text" ng-model="user.name">

    <div class="nested" ng-controller="MyNestedCtrl">
      <label>Primitive</label>
      <input type="text" ng-model="name">

      <label>Primitive with explicit $parent reference</label>
      <input type="text" ng-model="$parent.name">

      <label>Object</label>
      <input type="text" ng-model="user.name">
    </div>
  </div>
</body>
```

The `app.js` file contains the controller definition and initializes the scope with some defaults:

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

app.controller("MyCtrl", function($scope) {
  $scope.name = "Peter";
  $scope.user = {
    name: "Parker"
  };
});

app.controller("MyNestedCtrl", function($scope) {
});
```

Play around with the various input fields and check how changes affect each other.

Discussion

All the default values are defined in `MyCtrl` which is the parent of `MyNestedCtrl`. When doing changes in the first input field, the changes will be in sync with the other input fields bound to the

`name` variable. They all share the same scope variable as long as they only read from the variable. If you change the nested value, a copy in the scope of the `MyNestedCtrl` will be created. From now on, changing the first input field will only change the nested input field which explicitly references the parent scope via `$parent.name` expression.

The object based value behaves differently in this regard. Wether you change the nested or the `MyCtrl` scopes input fields, the changes will stay in sync. In Angular a scope prototypically inherits properties from a parent scope. Objects are therefore references and kept in sync. Whereas primitive types are only in sync as long they are not changed in the child scope.

Generally I tend to not use `$parent.name` and instead always use objects to share model properties. Additionally, the `MyNestedCtrl` not only requires certain model attributes but also a correct scope hierachy to work with.

Sharing Code Between Controllers using Services

Problem

You want to share business logic between controllers.

Solution

Use a Service[5] to implement your business logic and use dependency injection to use this service in your controllers.

The template shows access to a list of users from two controllers:

```
1 <div ng-controller="MyCtrl">
2   <ul ng-repeat="user in users">
3     <li>{{user}}</li>
4   </ul>
5   <div class="nested" ng-controller="AnotherCtrl">
6     First user: {{firstUser}}
7   </div>
8 </div>
```

The service and controller implementation in `app.js` implements a user service and the controllers set the scope initially:

[5] http://docs.angularjs.org/guide/dev_guide.services

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

app.factory("UserService", function() {
  var users = ["Peter", "Daniel", "Nina"];

  return {
    all: function() {
      return users;
    },
    first: function() {
      return users[0];
    }
  };
});

app.controller("MyCtrl", function($scope, UserService) {
  $scope.users = UserService.all();
});

app.controller("AnotherCtrl", function($scope, UserService) {
  $scope.firstUser = UserService.first();
});
```

Discussion

The `factory` method creates a singleton `UserService` which returns two functions for retrieving all users and the first user only. The controllers get the `UserService` injected by adding it to the function as params.

Testing Controllers

Problem

You want to unit test your business logic.

Solution

Implement a unit test using Jasmine[6] and the angular-seed[7] project. Following our previous `$watch` recipe this is how our spec would look like.

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

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

```
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.

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.

Enabling/Disabling DOM Elements 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.

```
1 <body ng-app>
2   <label><input type="checkbox" ng-model="checked"/>Toggle Button</label>
3   <button ng-disabled="checked">Press me</button>
4 </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[8] and explore all the directives Angular has to offer.

In the next recipes we will focus on implementing directives.

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

Changing the DOM in Response to User Actions

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 which 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[9].

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.

The `directive` method expects a function which can be used for initialization and injection of dependencies.

```
app.directive("myWidget", function factory(injectables) {
  // ...
}
```

Rendering an HTML Snippet in a Directive

Problem

You want to render an HTML snippet as a reusable component.

[9] 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.

Rendering a Directive's DOM Node Children

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.

Passing 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}}</p>",
    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` model we changed might be already defined in the parent scope and 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}}</p>",
  link: linkFunction,
  scope: {}
};
```

In Angular this is called an isolate scope. It does not prototypically inherit from the parent scope and is especially useful when creating reusable components.

Lets look into another way of passing params to the directive, this time we define an HTML element `my-widget`.

```
<my-widget2 text="Hello World"></my-widget2>

app.directive("myWidget2", function() {
  return {
    restrict: "E",
    template: "<p>{{text}}</p>",
    scope: {
      text: "@text"
    }
  };
});
```

The scope definition using `@text` is binding the text model to the directive's attribute. Note, that any changes on the parent scope `text` will change the local scope `text`, but not the other way around

If you want instead to have a bi-directional binding between the parent scope and the local scope, you should use the = equality character:

```
scope: {
  text: "=text"
}
```

Changes to the local scope will also change the parent scope.

Another possibility would be to pass an expression as a function to the directive using the `&` character.

```
<my-widget-expr fn="count = count + 1"></my-widget-expr>

app.directive("myWidgetExpr", function() {
  var linkFunction = function(scope, element, attributes) {
    scope.text = scope.fn({ count: 5 });
  };

  return {
    restrict: "E",
    template: "<p>{{text}}</p>",
    link: linkFunction,
    scope: {
      fn: "&fn"
    }
  };
});
```

We pass the attribute `fn` to the directive and since the local scope defines `fn` accordingly we can call the function in the linkFunction and pass in the expression arguments as a hash.

Repeatedly Rendering Directive's DOM Node Children

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[10] 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:

```
compile: function(tElement, attrs) {
  var content = tElement.children();
  for (var i=1; i<attrs.repeat; i++) {
    tElement.append(content.clone());
  }

  return function (scope, element, attrs) {
    element.on("click", "h1", function() {
      $(this).css({ "background-color": "red" });
    });
  };
}
```

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

Directive to Directive Communication

Problem

You want a directive to communicate with another directive and augment each other's behaviour using a well defined interface (API).

Solution

Implement a directive with a controller function and a second directive which "requires" this controller.

```
<body ng-app="MyApp">
  <basket apple orange>Roll over me and check the console!</basket>
</body>

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

app.directive("basket", function() {
  return {
    restrict: "E",
    controller: function($scope, $element, $attrs) {
      $scope.content = [];

      this.addApple = function() {
        $scope.content.push("apple");
      };

      this.addOrange = function() {
        $scope.content.push("orange");
      };
    },
    link: function(scope, element) {
      element.bind("mouseenter", function() {
        console.log(scope.content);
      });
    }
  };
});

app.directive("apple", function() {
```

```
30    return {
31      require: "basket",
32      link: function(scope, element, attrs, basketCtrl) {
33        basketCtrl.addApple();
34      }
35    };
36  });
37
38  app.directive("orange", function() {
39    return {
40      require: "basket",
41      link: function(scope, element, attrs, basketCtrl) {
42        basketCtrl.addOrange();
43      }
44    };
45  });
```

If you hover with the mouse over the rendered text the console should print and the basket's content.

Discussion

`Basket` is the example directive which demonstrates an API using the controller function, whereas the `apple` and `orange` directives augment the `Basket` directive. They both define a dependency to the `basket` controller with the `require` attribute. The link function then gets `basketCtrl` injected.

Note, how the `basket` directive is defined as an HTML element and the `apple` and `orange` directives are defined as HTML attributes (the default for directives). This demonstrates the typical use case of a reusable component augmented by other directives.

Now there might be other ways of passing data back and forth between directives - we have seen the different semantics of using the (isolated) context in directives in previous recipes - but what's especially great about the controller is the clear API contract it lets you define.

Testing Directives

Problem

You want to test your directive with a unit test. As an example we use a tab component directive implementation which can be easily used in your HTML document.

```
<tabs>
  <pane title="First Tab">First pane.</pane>
  <pane title="Second Tab">Second pane.</pane>
</tabs>
```

The directive implementation is split into the tabs and the pane directive. Let us start with the tabs directive.

```
app.directive("tabs", function() {
  return {
    restrict: "E",
    transclude: true,
    scope: {},
    controller: function($scope, $element) {
      var panes = $scope.panes = [];

      $scope.select = function(pane) {
        angular.forEach(panes, function(pane) {
          pane.selected = false;
        });
        pane.selected = true;
        console.log("selected pane: ", pane.title);
      };

      this.addPane = function(pane) {
        if (!panes.length) $scope.select(pane);
        panes.push(pane);
      };
    },
    template:
      '<div class="tabbable">' +
        '<ul class="nav nav-tabs">' +
          '<li ng-repeat="pane in panes" ng-class="{active:pane.selected}">\
'+
            '<a href="" ng-click="select(pane)">{{pane.title}}</a>' +
          '</li>' +
        '</ul>' +
        '<div class="tab-content" ng-transclude></div>' +
      '</div>',
    replace: true
  };
});
```

It manages a list of panes and the selected state of the panes. The template definition makes use of the selection to change the class and responds on the click event to change the selection.

The pane directive depends on the tabs directive to add itself to it.

```
app.directive("pane", function() {
  return {
    require: "^tabs",
    restrict: "E",
    transclude: true,
    scope: {
      title: "@"
    },
    link: function(scope, element, attrs, tabsCtrl) {
      tabsCtrl.addPane(scope);
    },
    template: '<div class="tab-pane" ng-class="{active: selected}" ng-trans\
clude></div>',
    replace: true
  };
});
```

Solution

Using the angular-seed in combination with jasmine and jasmine-jquery you can implement a unit test.

```
describe('MyApp Tabs', function() {
  var elm, scope;

  beforeEach(module('MyApp'));

  beforeEach(inject(function($rootScope, $compile) {
    elm = angular.element(
      '<div>' +
        '<tabs>' +
          '<pane title="First Tab">' +
            'First content is {{first}}' +
          '</pane>' +
          '<pane title="Second Tab">' +
            'Second content is {{second}}' +
          '</pane>' +
```

```
16          '</tabs>' +
17        '</div>');
18
19      scope = $rootScope;
20      $compile(elm)(scope);
21      scope.$digest();
22    }));
23
24    it('should create clickable titles', function() {
25      console.log(elm.find('ul.nav-tabs'));
26      var titles = elm.find('ul.nav-tabs li a');
27
28      expect(titles.length).toBe(2);
29      expect(titles.eq(0).text()).toBe('First Tab');
30      expect(titles.eq(1).text()).toBe('Second Tab');
31    });
32
33    it('should set active class on title', function() {
34      var titles = elm.find('ul.nav-tabs li');
35
36      expect(titles.eq(0)).toHaveClass('active');
37      expect(titles.eq(1)).not.toHaveClass('active');
38    });
39
40    it('should change active pane when title clicked', function() {
41      var titles = elm.find('ul.nav-tabs li');
42      var contents = elm.find('div.tab-content div.tab-pane');
43
44      titles.eq(1).find('a').click();
45
46      expect(titles.eq(0)).not.toHaveClass('active');
47      expect(titles.eq(1)).toHaveClass('active');
48
49      expect(contents.eq(0)).not.toHaveClass('active');
50      expect(contents.eq(1)).toHaveClass('active');
51    });
52  });
```

Discussion

Combining jasmine with jasmine-jquery gives you useful assertions like `toHaveClass` and actions like `click` which are used extensively in the example above.

To prepare the template we use `$compile` and `$digest` in the `beforeEach` function and then access the resulting Angular element in our tests.

The angular-seed project was slightly extended to add jquery and jasmine-jquery to the project.

The example code was extracted from Vojta Jina' Github example[11] - the author of the awesome Testacular[12].

[11] https://github.com/vojtajina/ng-directive-testing/tree/start

[12] https://github.com/testacular/testacular

Filters

Angular Filters are typically used to format expressions in bindings in your template. They transform the input data to a new formatted data type.

Formatting a String With a Currency Filter

Problem

You want to format the amount with a localized currency label.

Solution

Use the built-in `currency` filter and make sure you load the corresponding locale file for the users language.

```
<html>
  <head>
    <meta charset='utf-8'>
    <script src="js/angular.js"></script>
    <script src="js/angular-locale_de.js"></script>
  </head>
  <body ng-app>
    <input type="text" ng-model="amount" placeholder="Enter amount"/>
    <p>Default Currency: {{ amount | currency }}</p>
    <p>Custom Currency: {{ amount | currency: "Euro Ã¢â□□Â¬" }}</p>
  </body>
</html>
```

Enter an amount and it will be displayed using Angular's default locale.

Discussion

In our example we explicitly load the German locale settings and therefore the default formatting will be in German. The english locale is shipped by default, so there's no need to include the angular-locale_en.js file. If you remove the script tag, you will see the formatting change to English instead. This means in order for a localized application to work correctly you need to load the corresponding locale file. All available locale files can be seen on github[13].

[13] https://github.com/angular/angular.js/tree/master/src/ngLocale

Implementing a Custom Filter to Reverse an Input String

Problem

You want to reverse users text input.

Solution

Implement a custom filter which reverses the input.

```
<body ng-app="MyApp">
  <input type="text" ng-model="text" placeholder="Enter text"/>
  <p>Input: {{ text }}</p>
  <p>Filtered input: {{ text | reverse }}</p>
</body>

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

app.filter("reverse", function() {
  return function(input) {
    var result = "";
    input = input || "";
    for (var i=0; i<input.length; i++) {
      result = input.charAt(i) + result;
    }
    return result;
  };
});
```

Discussion

Angular's filter function expects a filter name and a function as params. The function must return the actual filter function, where you must implement your business logic. In this example it will only have an `input` param. The result will be returned after the for loop reversed the input completely.

Passing Configuration Params to Filters

Problem

You want to make your filter customizable by introducing config params.

Solution

Angular filters can be passed a hash of params which can be directly accessed in the filter function.

```
<body ng-app="MyApp">
  <input type="text" ng-model="text" placeholder="Enter text"/>
  <p>Input: {{ text }}</p>
  <p>Filtered input: {{ text | reverse: { suffix: "!"} }}</p>
</body>

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

app.filter("reverse", function() {
  return function(input, options) {
    input = input || "";
    var result = "";
    var suffix = options["suffix"] || "";

    for (var i=0; i<input.length; i++) {
      result = input.charAt(i) + result;
    }

    if (input.length > 0) result += suffix;

    return result;
  };
});
```

Discussion

The suffix `!` is passed as an option to the filter function and is appended to the output.

Filtering a List of DOM Nodes

Problem

You want to filter a list of names.

Solution

Angular filters not only work with Strings as input but also with Arrays.

```
<body ng-app="MyApp">
  <ul ng-init="names = ['Peter', 'Anton', 'John']">
    <li ng-repeat="name in names | exclude:'Peter' ">
      <span>{{name}}</span>
    </li>
  </ul>
</body>

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

app.filter("exclude", function() {
  return function(input, exclude) {
    var result = [];
    for (var i=0; i<input.length; i++) {
      if (input[i] !== exclude) {
        result.push(input[i]);
      }
    }

    return result;
  };
});
```

We pass `Peter` as the single param to the exclude filter.

Discussion

The filter implemention loops through all names and creates a result array excluding 'Peter'. Note, that the actual syntax of the filter function didn't change at all from our previous example with the String input.

Chaining Filters together

Problem

You want to combine several filters to a single result

Solution

Filters can be chained using the UNIX-like pipe syntax.

```
<body ng-app="MyApp">
  <ul ng-init="names = ['Peter', 'Anton', 'John']">
    <li ng-repeat="name in names | exclude:'Peter' | sortAscending ">
      <span>{{name}}</span>
    </li>
  </ul>
</body>
```

Discussion

The pipe symbol (`|`) is used to chain multiple filters together. I leave the implementation of the `sortAscending` filter as an exercise to the reader.

Testing Filters

Problem

You want to unit test your new filter. Let us start with an easy filter which renders a checkmark depending on a boolean value.

```
<body ng-init="data = true">
  <p>{{ data | checkmark}}</p>
  <p>{{ !data | checkmark}}</p>
  <script src="lib/angular/angular.js"></script>
  <script src="js/app.js"></script>
</body>

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

```

```
app.filter('checkmark', function() {
  return function(input) {
    return input ? '\u2713' : '\u2718';
  };
});
```

Solution

Use the angular-seed project as a bootstrap again.

```
describe('MyApp Tabs', function() {
  beforeEach(module('MyApp'));

  describe('checkmark', function() {
     it('should convert boolean values to unicode checkmark or cross', inje\
ct(function(checkmarkFilter) {
       expect(checkmarkFilter(true)).toBe('\u2713');
       expect(checkmarkFilter(false)).toBe('\u2718');
     }));
   });
});
```

Discussion

The `beforeEach` loads the module and the `it` methods injects the filter function for us. Note, that it has to be called `checkmarkFilter`, otherwise Angular can't inject our filter function correctly.

Consuming Externals Services

Angular has built-in support for doing AJAX requests with low- and high-level APIs.

Requesting JSON Data with AJAX

Problem

You want to do an AJAX request to fetch JSON data and render it.

Solution

Implement a controller using `$http` to fetch the data and store it in the scope.

```
<body ng-app="MyApp">
  <div ng-controller="PostsCtrl">
    <ul ng-repeat="post in posts">
      <li>{{post.title}}</li>
    </ul>
  </div>
</body>

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

app.controller("PostsCtrl", function($scope, $http) {
  $http.get('data/posts.json').
    success(function(data, status, headers, config) {
      $scope.posts = data;
    });
});
```

Discussion

The controller defines a dependency to the `$scope` and the `$http` module. An HTTP get call to the `data/posts.json` URL is done and on success the JSON data is put in `$scope.posts`.

You can set custom HTTP headers by using the `$http.defaults` function:

```
$http.defaults.headers.common["X-Custom-Header"] = "Angular.js"
```

The complete example uses the angular-seed project again.

Consuming RESTful APIs

Problem

You want to consume a RESTful API.

Solution

Use Angular's high-level `$resource` module to use RESTful APIs.

Let us start with defining the application module and our `Post` model:

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

app.factory("Post", function($resource) {
  return $resource("/api/posts/:id");
});
```

Now we can use the ‘$resource$ to retrieve a list of posts:

```
app.controller("PostIndexCtrl", function($scope, Post) {
  Post.query(function(data) {
    $scope.posts = data;
  });
});
```

Or a specific post by `id`:

```
app.controller("PostShowCtrl", function($scope, Post) {
  Post.get({ id: 1 }, function(data) {
    $scope.post = data;
  });
});
```

And delete a specific post:

```
app.controller("PostDestroyCtrl", function($scope, Post) {
  $scope.destroy = function(id) {
    Post.delete({ id: id });
  }
});
```

Note, that the Angular `ngResource` module needs to be separately loaded since it is not included in the base angular.js file:

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

Discussion

Following some conventions simplifies our code quite a bit. We define the `$resource` by passing the URL schema only. This gives us a handful of nice methods including query, get, save and remove to work with our resource. In the example above we implement several controllers to cover the various typical use cases.

It is generally a good practice to encapsulate your model and `$resource$` usage in an Angular `service` module and inject that in your controller.

What if your response of posts is not an array but a more complex json? This typically results in the following error:

```
TypeError: Object #<Resource> has no method 'push'
```

Have a look at the following JSON example:

```
{
  "posts": [
    {
      "id"    : 1,
      "title" : "title 1"
    },
    {
      "id": 2,
      "title" : "title 2"
    }
  ]
}
```

In this case you have to change the ‘$resource$ definition accordingly.

```
app.factory("Post", function($resource) {
  return $resource("/api/posts/:id", {}, {
    query: { method: "GET", isArray: false }
  });
});

app.controller("PostIndexCtrl", function($scope, Post) {
  Post.query(function(data) {
    $scope.posts = data.posts;
  });
});
```

Note, that we only change the configuration of the `query` action to not expecting an array. Then in our controller we can directly access the `data.posts`.

The complete example code is based on Brian Ford's angular-express-seed[14] and uses the Express[15] framework.

Consuming JSONP APIs

Problem

You want to call a JSONP API.

Solution

Use the `$resource` and configure it to use JSONP. As an example we will call the Twitter search API here.

```
<body ng-app="MyApp">
  <div ng-controller="MyCtrl">
    <input type="text" ng-model="searchTerm" placeholder="Search term">
    <button ng-click="search()">Search</button>
    <ul ng-repeat="tweet in searchResult.results">
      <li>{{tweet.text}}</li>
    </ul>
  </div>
</body>

```

[14] https://github.com/btford/angular-express-seed
[15] http://expressjs.com/

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

function MyCtrl($scope, $resource) {
  $scope.twitterAPI = $resource("http://search.twitter.com/search.json",
    { callback: "JSON_CALLBACK" },
    { get: { method: "JSONP" }});

  $scope.search = function() {
    $scope.searchResult = $scope.twitterAPI.get({ q: $scope.searchTerm });
  };
}
```

Discussion

The `$resource` definition sets the `callback` attribute to `JSON_CALLBACK`, a requirement from Angular. Additionally we overwrite the get method to use `JSONP`. Now, when calling the API we use the `q` param to pass the entered `searchTerm`.

Deferred and Promise

TODO: Write me!

Testing Services

Problem

You want to unit test your controller and service consuming a JSONP API.

Lets have a look again at our example we want to test:

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

app.factory("TwitterAPI", function($resource) {
  return $resource("http://search.twitter.com/search.json",
    { callback: "JSON_CALLBACK" },
    { get: { method: "JSONP" }});
});

app.controller("MyCtrl", function($scope, TwitterAPI) {
  $scope.search = function() {
```

```
11      $scope.searchResult = TwitterAPI.get({ q: $scope.searchTerm });
12    };
13  });
```

Note, that it slightly changed from the previous recipe as the `TwitterAPI` is pulled out of the controller and resides in its own service now.

Solution

Use the angular-seed project and the $http_backend mocking service.

```
1   describe('MyCtrl', function(){
2     var scope, ctrl, httpBackend;
3
4     beforeEach(module("MyApp"));
5
6     beforeEach(inject(function($controller, $rootScope, TwitterAPI, $httpBack\
7   end) {
8       httpBackend = $httpBackend;
9       scope = $rootScope.$new();
10      ctrl = $controller("MyCtrl", { $scope: scope, TwitterAPI: TwitterAPI })\
11  ;
12
13      var mockData = { key: "test" };
14      var url = "http://search.twitter.com/search.json?callback=JSON_CALLBACK\
15  &q=angularjs";
16      httpBackend.whenJSONP(url).respond(mockData);
17    }));
18
19    it('should set searchResult on successful search', function() {
20      scope.searchTerm = "angularjs";
21      scope.search();
22      httpBackend.flush();
23
24      expect(scope.searchResult.key).toBe("test");
25    });
26
27  });
```

Discussion

Since we now have a clear separation between the service and the controller, we can simply inject the `TwitterAPI` in our `beforeEach` function.

Mocking with the `$httpBackend` is done as a last step in `beforeEach`. When a JSONP request happens we respond with `mockData`. After the `search()` is triggered we `flush()` the `httpBackend` in order to return our `mockData`.

Have a look at the ngMock.$httpBackend[16] module for more details.

[16] http://docs.angularjs.org/api/ngMock.\protect\char"0024\relaxhttpBackend

URLs, Routing and Partials

The $location service[17] in Angular.js parses the current browser URL and makes it available to your application. Changes in either the browser address bar or the `$location` service will be kept in sync.

Depending on the configuration the `$location` service behaves differently and has different requirements for your application. We will first look into client-side routing with hashbang URLs since it is the default mode and later into the new HTML5 based routing.

Client-Side Routing with Hashbang URLs

Problem

You want the browser address bar reflect your apps page flow consistently.

Solution

Use the `$routeProvider` and `$locationProvider` services to define your routes and the `ng-view` Directive as the placeholder for the partials which should be shown for a particular route definition.

The basic HTML uses the `ng-view` directive:

```
<body>
  <h1>Routing Example</h1>
  <ng-view></ng-view>

  <script src="lib/angular/angular.js"></script>
  <script src="js/app.js"></script>
</body>
```

The route configuration is implemented in `app.js` using the `config` method:

[17] http://docs.angularjs.org/guide/dev_guide.services.\protect\char"0024\relaxlocation

```
var app = angular.module("MyApp", []).
  config(function($routeProvider, $locationProvider) {
    $locationProvider.hashPrefix('!');
    $routeProvider.
      when("/persons", { templateUrl: "partials/index.html" }).
      when("/persons/:id", { templateUrl: "partials/show.html", controller:\
 "ShowCtrl" }).
      otherwise( { redirectTo: "/persons" });
});
```

The partial `index.html`:

```
<h3>Person List</h3>
<div ng-controller="IndexCtrl">
  <table>
    <thead>
      <tr>
        <td>Name</td>
        <td>Actions</td>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="person in persons">
        <td>{{person.name}}</td>
        <td><a href="#!persons/{{person.id}}">Details</a></td>
      </tr>
    </tbody>
  </table>
</div>
```

And the partial `show.html`;

```
<h3>{{person.name}} Details</h3>
<p>Name: {{person.name}}</p>
<p>Age: {{person.age}}</p>

<a href="#!persons">Go back</a>
```

Our app is configured to render either the `index.html` or the `show.html` partial depending on the URL. The `index.html` shows a list of `persons` and the `show.html` shows more detailed information for a specific `person`.

This example is based on the Angular Seed Bootstrap again and will not work without starting the development server. The complete project is available on Github.

Discussion

Let's give our app a try and open the `index.html`. The `otherwise` defined route redirects us from `index.html` to `index.html#!/persons`. This is the fallback in case other `when` conditions don't apply. Note, how the hashbang (`#!`) separates the `index.html` from the dynamic client-side part `/persons`.

The `/persons` route loads the `index.html` partial via HTTP Request (that is also the reason why it won't work without a development server). It shows a list of persons and therefore defines a `ng-controller` directive inside the template. Let us assume for now that the Controller implementation defines a `$scope.persons` somewhere. Now for each person we also render a link to show the details via `#!persons/{{person.id}}`.

The route definition for the person's details uses a placeholder `/persons/:id` which in our case resolves to for example `/persons/1`. The `show.html` partial and additionally a Controller are defined for this URL. The Controller will be scoped to the partial, which basically resembles our `index.html` template where we defined our own `ng-controller` Directive to achieve the same effect.

The `show.html` has a back link to `#!persons` which leads back to the `index.html` page.

Let us come back to the `ng-view` directive. It is automatically bound to the router definition. Therefore you can currently use only a single `ng-view` on your page. For example, you cannot use nested `ng-views` to achieve user interaction patterns with a first- and second level navigation.

And lastly the HTTP request for the partials happens only once and is then cached via `$templateCache` service.

The hashbang based routing is client-side only and doesn't require server-side configuration. Let us look into the HTML5 based approach next.

Using Regular URLs with the HTML5 History API

Problem

You want nice looking URLs and can provide server-side support.

Solution

We use the same example but use the Express[18] framework to serve all content and do the URL rewriting. Let us start with the route configuration:

[18] http://expressjs.com/

```
app.config(function($routeProvider, $locationProvider) {
  $locationProvider.html5Mode(true);

  $routeProvider.
    when("/persons", { templateUrl: "/partials/index.jade", controller: "Pe\
rsonIndexCtrl" }).
    when("/persons/:id", { templateUrl: "/partials/show.jade", controller: \
"PersonShowCtrl" }).
    otherwise( { redirectTo: "/persons" });
});
```

There are no changes except the `html5Mode` method which enables our new routing mechanism. The Controller implementation does not change at all.

We have to take care of the partial loading. Our `Express` app will have to serve the partials for us:

```
app.get('/partials/:name', function (req, res) {
  var name = req.params.name;
  res.render('partials/' + name);
});
```

The `Express` route definition loads the partial with given name from the `partials` directory and renders its content.

When supporting HTML5 routing our server has to redirect all other URLs to the entry point of our application, the `index` page. First the rendering of the `index` page, which contains the `ng-view` Directive:

```
app.get('/', function(req, res) {
  res.render('index');
});
```

Then the catch all route which redirects to the same page:

```
app.get('*', function(req, res) {
  res.redirect('/');
});
```

The person details link `/persons/{{person.id}}` and the back link `/persons` are both now much cleaner.

Have a look at the complete example on Github and start the `Express` app with `node app.js`.

Discussion

If we wouldn't redirect all requests to the root, what would happen if you navigate to the persons list at `http://localhost:3000/persons`? The `Express` framework would show us an error because there is no route defined for `persons`, we only defined routes for our root `/` and the partials URL `/partials/:name`. The redirect ensures that we actually end up at our root URL which then kicks in our Angular app. When the client-side routing takes over we actually then redirect back to the `/persons` URL.

Also note how navigating to a Person's detail page will load only the `show.jade` partial and navigating back to the `persons` list won't do any server requests. Everything our app needs is loaded from the server and cached client-side.

If you have a hard time understanding the server implementation I suggest to read the excellent Express Guide[19]. Additionally, there is going to be an extra chapter which goes into more details on how to integrate Angular.js with server-side frameworks.

[19] http://expressjs.com/guide.html

Using Forms

Implementing a Basic Form

Problem

You want to create a form to enter user details and capture this information in an Angular.js scope.

Solution

Use the standard `form` tag and the `ng-model` Directive to implement a basic form:

```
<body ng-app="MyApp">
  <div ng-controller="User">
    <form ng-submit="submit()" class="form-horizontal">
      <label>Firstname</label>
      <input type="text" ng-model="user.firstname"/>
      <label>Lastname</label>
      <input type="text" ng-model="user.lastname"/>
      <label>Age</label>
      <input type="text" ng-model="user.age"/>
      <button class="btn">Submit</button>
    </form>
  </div>
</body>
```

And a controller to bind the form data to your user model:

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

app.controller("User", function($scope) {
  $scope.user = {};
  $scope.wasSubmitted = false;

  $scope.submit = function() {
    $scope.wasSubmitted = true;
  };
});
```

Discussion

The initial idea when using forms would be to implement them the traditional way by serializing the form data and submit it to the server. Instead we use `ng-model` to bind the form to our model, something we have been doing a lot already in previous recipes. The submit button state is reflected in our `wasSubmitted` scope variable, but no submit to the server was actually done. The default behaviour in Angular.js forms is to prevent the default action since we do not want to reload the whole page. We want to handle the submission in an application specific way. In fact there is even more going on in the background and we are going to look into the behaviour of the `form` or `ng-form` Directive in the next recipe.

Validating a Form Model client-side

Problem

You want to validate the form client-side using HTML5 form attributes.

Solution

Angular.js works in tandem with HTML5 form attributes. Let us start with the same form but let us add some HTML5 attributes to make the input required:

```
<form name="form" ng-submit="submit()" class="form-horizontal">
  <label>Firstname</label>
  <input name="firstname" type="text" ng-model="user.firstname" required/>
  <label>Lastname</label>
  <input type="text" ng-model="user.lastname" required/>
  <label>Age</label>
  <input type="text" ng-model="user.age"/>
  <br>
  <button class="btn">Submit</button>
</form>
```

It is still the same form but this time we defined the name attribute on the form and the input for the firstname.

Let us add some more debug output:

```
Firstname input valid: {{form.firstname.$valid}}
<br>
Firstname validation error: {{form.firstname.$error}}
<br>
Form valid: {{form.$valid}}
<br>
Form validation error: {{form.$error}}
```

Discussion

When starting with a fresh empty form, you notice that Angular adds the css class `ng-pristine` and `ng-valid` to the form tag and each input tag. When editing the form the `ng-pristine` class will be removed from the changed input and also from the form tag. Instead it will be replaced by the `ng-dirty` class. Very useful because you can easily add new features to your app depending on these states.

In addition to these two css classes there are two more to look into. The `ng-valid` class will be added whenever an input is valid, otherwise the css class `ng-invalid` is added. Note, that the form tag also gets either a valid or invalid class depending on the inputs. To demonstrate this I've added the `required` HTML5 attribute. Initially, the firstname and lastname input fields are empty and therefore have the `ng-invalid` css class, whereas the age input field has the `ng-valid` class. Additionally, there's `ng-invalid-required` class alongside the `ng-invalid` for even more specificity.

Since we defined the `name` attribute on the form HTML element we can now access Angular's form controller via scope variables. In the debug output we can check the validity and specific error for each named form input and the form itself. Note, that this only works on the level of the form name attributes and not on the model scope. If you output the following expression `{{user.firstname.$error}}` it will not work.

Angular's Form Controller exposes `$valid`, `$invalid`, `$error`, `$pristine` and `$dirty` variables.

Angular.js provides built-in Directives for validation including `required`, `pattern`, `minlength`, `maxlength`, `min` and `max`.

Let us use Angular's form integration to actually show validation errors.

Displaying Form Validation Errors

Problem

You want to show valiation errors to the user by marking the input field red and displaying an error message.

Solution

We can use the `ng-show` Directive to show an error message if a form input is invalid and css classes to change the input background color depending on its state.

Let us start with the styling changes:

```
<style type="text/css">
  input.ng-invalid.ng-dirty {
    background-color: red;
  }
  input.ng-valid.ng-dirty {
    background-color: green;
  }
</style>
```

And here is a small part of the form with a error message for the input field:

```
<label>Firstname</label>
<input name="firstname" type="text" ng-model="user.firstname" required/>
<p ng-show="form.firstname.$invalid && form.firstname.$dirty">Firstname is \
required</p>
```

Discussion

The CSS classes ensure that we initially show the fresh form without any colors. When the user starts typing in a input for the first time, we change it to either green or red. That is a good example usage of the `ng-dirty` and `ng-invalid` css classes.

We use the same logic in the ‘ng-show directive to only show the error message when the user started typing for the first time.

Only Enabling the Submit button if the Form is Valid

Problem

You want to disable the Submit button as long as the form contains invalid data.

Solution

Use the `$form.invalid` state in combination with a `ng-disabled` Directive.

Here the changed submit button:

```
<button ng-disabled="form.$invalid" class="btn">Submit</button>
```

Discussion

The Form Controller attributes `form.$invalid` and friends are very useful to cover all kinds of use cases which focus on the form as a whole instead of individual fields.

Implementing Custom Validations

Problem

You want to validate user input by comparing it to a blacklist of words.

Solution

The angular-ui[20] project offers a nice custom validation directive which lets you pass in options via expression.

Let us have a look at the template first with the usage of the `ui-validate` Directive:

```
<input name="firstname" type="text" ng-model="user.firstname" required ui-v\
alidate=" { blacklisted: 'notBlacklisted($value)' } "/>

<p ng-show='form.firstname.$error.blackListed'>This firstname is blackliste\
d.</p>
```

And the controller with the `notBlackListed` implementation:

```
$scope.blacklist = ['idiot','looser'];

$scope.notBlackListed = function(value) {
  return $scope.blacklist.indexOf(value) === -1;
};
```

Our blacklist contains the firstnames we do not want to accept as user input and the `notBlackListed` function checks if the user input matches.

[20] http://angular-ui.github.com/

Discussion

The `ui-validate` Directive is pretty powerful since it lets you define your custom validations easily by just implementing the business logic in a controller function.

If you want even more, have a look at how to implement custom directives yourself in Angular's excellent guide[21].

[21] http://docs.angularjs.org/guide/forms

Common User Interface Patterns

Filtering and Sorting a List

Paginating Through Client-Side Data

Paginating Through Server-Side Data

Pagination using Infinite Results

Selectable List Items

Displaying a Flash Notice/Failure Message

Editing Text In-Place using HTML5 ContentEditable

Displaying a Modal Dialog

Displaying a Loading Spinner

Debugging and Tooling

Inspecting the Scope Using the Chrome Console

Debugging and Profiling with the Batarang Chrome Extension

Enhancing Developer Workflow with Yeoman

Backend Integration

Rails

Node.js

The complete example code is based on Brian Ford's angular-express-seed[22] and uses the Express[23] framework.

[22] https://github.com/btford/angular-express-seed
[23] http://expressjs.com/