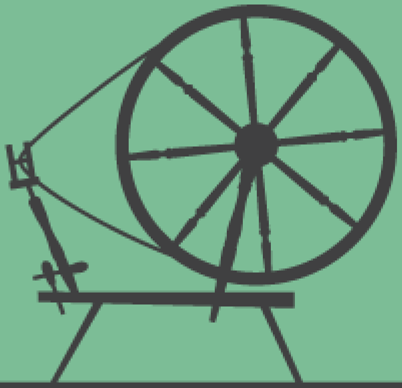

AngularJS Succinctly

by Frederik Dietz

Syncfusion | Technology Resource Portal

Angular.js Succinctly

By
Frederik Dietz

Foreword by Daniel Jebaraj

Syncfusion
Deliver innovation with ease

Technical Reviewer: Rui Machado
Copy Editor: Suzanne Kattau
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet, and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just like everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Frederik Dietz is a passionate software engineer with a focus on web-based and mobile technologies. He currently works at the startup Protonet, shipping orange boxes made with love in Hamburg.

Frederik has a blog at fdietz.de. You can reach him on Twitter via @fdietz or by email at fdietz@gmail.com.

Preface

Introduction

Angular.js is an open-source JavaScript 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.

How to Contact Me

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

Frederik Dietz (fdietz@gmail.com)

Acknowledgements

Special thanks go to my English editor and friend Robert William Smales!

Thanks go to John Lindquist for his excellent screencast project egghead.io, Lukas Ruebbelke for his awesome videos, and Matias Niemela for his great blog. And, of course, the whole development team behind Angular.js!

Chapter 1 An Introduction to Angular.js

Including the Angular.js Library Code in an HTML Page

Problem

You wish 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/angular.js">
    </script>
  </head>
  <body ng-app>
    <p>This is your first angular expression: {{ 1 + 2 }}</p>
  </body>
</html>
```

You can check out a complete example on GitHub.

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 rendering the expression as is instead of evaluating it. Play around with the expression! You can, for instance, concatenate Strings and invert or combine Boolean values.

For reasons of brevity, we will 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 Angular's `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>
```

You can find the complete example on GitHub.

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!

Responding to Click Events using Controllers

Problem

You wish 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="visible">Hello World!</p>
    </div>
  </body>
</html>
```

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

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

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

You can find the complete example on GitHub.

When toggling the button, the "Hello World" paragraph will change its visibility.

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 our controller on button click. Note that the `ng-show` directive is bound to the `visible` scope variable and will toggle the paragraph's visibility accordingly.

The controller implementation defaults the `visible` attribute to true and toggles its Boolean state in the `toggle` function. Both the `visible` variable and the `toggle` function are defined on the `$scope` service, which is passed to all controller functions automatically via dependency injection.

The next chapter will go into all the details of controllers in Angular. For now, let us quickly discuss the Model-View-ViewModel (MVVM) pattern as used by Angular. In the MVVM pattern, the model is plain JavaScript, the view is the HTML template, and the ViewModel is the glue between the template and the model. The ViewModel makes Angular's two-way binding possible where changes in the model or the template are in sync automatically.

In our example, the `visible` attribute is the model, but it could of course be much more complex when, for example, retrieving data from a backend service. The controller is used to define the scope which represents the ViewModel. It interacts with the HTML template by binding the scope variable `visible` and the function `toggle()` to it.

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:

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

You can find the complete example on GitHub.

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!

Creating Custom HTML Elements with Directives

Problem

You wish to render an HTML snippet but hide it conditionally.

Solution

Create a custom directive which renders your Hello World snippet:

```
<body ng-app="MyApp">
  <label for="checkbox">
    <input id="checkbox" type="checkbox" ng-model="visible">Toggle me
  </label>
  <div show="visible">
    <p>Hello World</p>
  </div>
</body>
```

The `show` attribute is our directive, with the following implementation:

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

app.directive("show", function() {
  return {
    link: function(scope, element, attributes) {
      scope.$watch(attributes.show, function(value){
        element.css('display', value ? '' : 'none');
      });
    }
  };
});
```

The browser will only show the "Hello World" paragraph if the checkbox is checked and will hide it otherwise.

You can find the complete example on GitHub.

Discussion

Let us ignore the `app` module creation for now and look into it in more depth in a later chapter. In our example, it is just the means to create a directive. Note that the `show` String is the name of the directive which corresponds to the `show` HTML attribute used in the template.

The directive implementation returns a `link` function that defines the directive's behavior. It gets passed the `scope`, the HTML `element,` and the HTML `attributes`. Since we passed the `visible` variable as an attribute to our `show` directive, we can access it directly via `attributes.show`. But since we want to respond to `visible` variable changes automatically, we have to use the `$watch` service, which provides us with a function callback whenever the value changes. In this callback, we change the CSS `display` property to either blank or `none` to hide the HTML element.

This was just a small glimpse into what can be achieved with directives; there is a whole chapter later that is dedicated to them, which goes into much more detail.

Chapter 2 Controllers

Controllers in Angular provide the business logic to handle view behavior; for example, responding to a user clicking a button or entering some text in a form. Additionally, controllers prepare the model for the view template.

As a general rule, a controller should not reference or manipulate the Document Object Model (DOM) directly. This has the benefit of simplifying unit testing controllers.

Assigning a Default Value to a Model

Problem

You wish to assign a default value to the scope in the controller's context.

Solution

Use the `ng-controller` directive in your template:

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

Next, define the scope variable in your controller function:

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

You can find the complete example on GitHub.

Discussion

Depending on where you use the ng-controller directive, you define its assigned scope. The scope is hierarchical and follows the DOM node 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 were moved outside the controller's scope:

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

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

In this case `{{value}}` will simply not be rendered at all due to the fact that expression evaluation in Angular.js is forgiving for `undefined` and `null` values.

Changing a Model Value with a Controller Function

Problem

You wish to increment a model value by 1 using a controller function.

Solution

Implement an increment function that changes the scope:

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

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

This function can be directly called in an expression; in our example we use `ng-init`:

```
<div ng-controller="MyCtrl">
  <p ng-init="incrementValue(1)">{{value}}</p>
</div>
```

You can find the complete example on GitHub.

Discussion

The `ng-init` directive is executed on page load and calls the function `incrementValue` defined in `MyCtrl`. Functions are defined on the scope very similarly to values but must be called with the familiar parenthesis syntax.

Of course, it would have been possible to increment the value right inside of the expression with `value = value +1` but imagine the function being much more complex! Moving this function into a controller separates our business logic from our declarative view template, and we can easily write unit tests for it.

Encapsulating a Model Value with a Controller Function

Problem

You wish to retrieve a model via a function (instead of directly accessing the scope from the template) that encapsulates the model value.

Solution

Define a getter function that returns the model value:

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

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

Then, in the template, we use an expression to call it:

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

You can find the complete example on GitHub.

Discussion

`MyCtrl` defines the `getIncrementedValue` function, which uses the current value and returns it incremented by 1. 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 controller’s behavior where a generic filter is not required.

Responding to Scope Changes

Problem

You wish to react on a model change to trigger some further actions. In our example, we simply want to set another model value depending on the value we are listening to.

Solution

Use the `$watch` function in your controller:

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

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

In our example, we use the text input value to print a friendly greeting:

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

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

You can find the complete example on GitHub.

Discussion

The first argument `name` of the `$watch` function is actually an Angular expression, so you can use more complex expressions (for example: `[value1, value2] | json`) or even a JavaScript function. In this case, you need to return a String in the watcher function:

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

The second argument is a function that is called whenever the expression evaluation returns a different value. The first parameter is the new value and the second parameter is the old value. Internally, this uses `angular.equals` to determine equality, which means both objects or values pass the === comparison.

Sharing Models between Nested Controllers

Problem

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

Solution

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

Our example template uses a controller `MyCtrl` 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 see how changes affect each other.

You can find the complete example on GitHub.

Discussion

All the default values are defined in `MyCtrl`, which is the parent of `MyNestedCtrl`. When making 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. Whether you change the nested or the `MyCtrl` scope's 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 `$parent.name` and instead always use objects to share model properties. If you use `$parent.name`, the `MyNestedCtrl` not only requires certain model attributes but also a correct scope hierarchy with which to work.

Tip: The Chrome plug-in Batarang simplifies debugging the scope hierarchy by showing you a tree of the nested scopes. It is awesome!

Sharing Code between Controllers using Services

Problem

You wish to share business logic between controllers.

Solution

Utilize a Service 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:

```
<div ng-controller="MyCtrl">
  <ul ng-repeat="user in users">
    <li>{{user}}</li>
  </ul>
  <div class="nested" ng-controller="AnotherCtrl">
    First user: {{firstUser}}
  </div>
</div>
```

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

```
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();
});
```

You can find the complete example on GitHub.

Discussion

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

Using dependency injection here is quite nice for testing your controllers since you can easily inject a `UserService` stub. The only downside is that you can't minify the code from above without breaking it, since the injection mechanism relies on the exact string representation of `UserService`. It is, therefore, recommended to define dependencies using inline annotations, which keep working even when minified:

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

The syntax looks a bit funny but, since strings in arrays are not changed during the minification process, it solves our problem. Note that you could change the parameter names of the function since the injection mechanism relies on the order of the array definition only.

Another way to achieve the same is using the `$inject` annotation:

```
var anotherCtrl = function($scope, UserService) {
  $scope.firstUser = UserService.first();
};

anotherCtrl.$inject = ["$scope", "UserService"];
```

This requires you to use a temporary variable to call the `$inject` service. Again, you could change the function parameter names. You will most likely see both versions applied in apps using Angular.

Testing Controllers

Problem

You wish to unit test your business logic.

Solution

Implement a unit test using Jasmine and the angular-seed project. Following our previous `$watch` recipe, this is how our spec would look:

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

  beforeEach(inject(function($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");
  });
});
```

You can find the complete example on GitHub.

Discussion

Jasmine specs use `describe` and `it` functions to group specs and `beforeEach` and `afterEach` to set up and tear down 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 close as possible to how our code would behave at run time, too. We can't just initialize the scope as a JavaScript object `{}` since we would then not be able to call `$watch` on it. Instead, `$rootScope.$new()` will do the trick. Note that the `$controller` service requires `MyCtrl` to be available and uses an object notation to pass in dependencies.

The `$digest` call is required in order to trigger a watch execution after we have changed the scope. We need to call `$digest` manually in our spec whereas, at run time, Angular will do this for us automatically.

Chapter 3 Directives

Directives are one of the most powerful concepts in Angular since they let you create custom HTML elements specific to your application. This allows you to develop reusable components, which encapsulate complex DOM structures, style sheets, and even behavior.

Enabling/Disabling DOM Elements Conditionally

Problem

You wish 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>
```

You can find the complete example on GitHub.

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, for example, `ng-hide`, `ng-checked`, or `ng-mouseenter`. I encourage you to go through the Application Programming Interface (API) Reference and explore all the directives Angular has to offer.

In the next recipes, we will focus on implementing directives.

Changing the DOM in Response to User Actions

Problem

You wish to change the CSS of an HTML element on a mouse click and encapsulate this behavior in a reusable component.

Solution

Implement a directive `my-widget` that contains an example paragraph of text you want to style:

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

Use a link function in the directive implementation to change the CSS of the paragraph:

```
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.

You can find the complete example on GitHub.

Discussion

In the HTML document, 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>
```

Whether or not you use the attribute or element mechanism will depend 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 behavior. Other available options are using the directive as a class attribute or a comment.

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

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

The link function is much more interesting since it defines the actual behavior. The scope, the actual HTML element `my-widget,` and the HTML attributes are passed as params. Note that this has nothing to do with Angular's 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 Angular's `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 particular 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 it is defined and will fall back to jqLite (shipped with Angular) otherwise. You can find all supported methods in the API Reference of element.

Following a slightly altered version of the code, using jQuery only:

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

In this case, `element` is already a jQuery element, and we can directly use the on function.

Rendering an HTML Snippet in a Directive

Problem

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

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>"
  };
});
```

You can find the complete example on GitHub.

Discussion

This will render the Hello World paragraph as a child node of your `my-widget` element. If you want to replace the element entirely with the paragraph, you will also have to 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 will need to use the `templateUrl` attribute, for example, as follows:

```
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 a 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 an `h3` element and append the directive's child node with the paragraph element below.

You can find the complete example on GitHub.

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 positioned depending on where you want your child nodes to be appended.

Passing Configuration Params Using HTML Attributes

Problem

You wish 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
  };
});
```

This renders a paragraph with the text passed as the param.

You can find the complete example on GitHub.

Discussion

The link function has access to the element and its attributes. It is, therefore, straightforward 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 already be defined in the parent scope and used in another part of your app. In order to isolate the context and thereby use it only locally inside 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.

Let's look at another way of passing params to the directive. This time we will define an HTML element `my-widget2`:

```
<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 to 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 option 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 wish to render an HTML snippet repeatedly using the directive's 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.

You can find the complete example on GitHub.

Discussion

The directive repeats the child nodes as often as configured in the `repeat` attribute. It works similarly to the ng-repeat directive. The implementation uses Angular's element methods to append the child nodes in a 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 can't use `$watch` to add behavior either. This is in comparison to the link function that has access to the DOM instance (after the compile phase) and has access to the scope to add behavior.

Use the compile function for template DOM manipulation only. Use the link function whenever you want to add behavior.

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" });
    });
  };
}
```

Clicking the header will change the background color to red.

Directive-to-Directive Communication

Problem

You wish a directive to communicate with another directive and augment each other's behavior using a well-definedAPI.

Solution

We implement a directive `basket` with a controller function and two other directives, `orange` and `apple,` which require this controller. Our example starts with an `apple` and `orange` directive used as attributes:

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

The `basket` directive manages an array to which one can add apples and oranges:

```
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);
      });
    }
  };
});
```

And finally, the apple and orange directives, which add themselves to the basket using the basket's controller:

```
app.directive("apple", function() {
  return {
    require: "basket",
    link: function(scope, element, attrs, basketCtrl) {
      basketCtrl.addApple();
    }
  };
});

app.directive("orange", function() {
  return {
    require: "basket",
    link: function(scope, element, attrs, basketCtrl) {
      basketCtrl.addOrange();
    }
  };
});
```

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

You can find the complete example on GitHub.

Discussion

`Basket` is the example directive that 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 wish to test your directive with a unit test. As an example, we will use a tab component directive implementation, which can easily be 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-transclude></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>' +
        '</tabs>' +
      '</div>');

    scope = $rootScope;
    $compile(elm)(scope);
    scope.$digest();
  }));

  it('should create clickable titles', function() {
    console.log(elm.find('ul.nav-tabs'));
    var titles = elm.find('ul.nav-tabs li a');

    expect(titles.length).toBe(2);
    expect(titles.eq(0).text()).toBe('First Tab');
    expect(titles.eq(1).text()).toBe('Second Tab');
  });

  it('should set active class on title', function() {
    var titles = elm.find('ul.nav-tabs li');

    expect(titles.eq(0)).toHaveClass('active');
    expect(titles.eq(1)).not.toHaveClass('active');
  });

  it('should change active pane when title clicked', function() {
    var titles = elm.find('ul.nav-tabs li');
    var contents = elm.find('div.tab-content div.tab-pane');

    titles.eq(1).find('a').click();

    expect(titles.eq(0)).not.toHaveClass('active');
    expect(titles.eq(1)).toHaveClass('active');

    expect(contents.eq(0)).not.toHaveClass('active');
    expect(contents.eq(1)).toHaveClass('active');
  });
});
```

You can find the complete example on GitHub.

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's GitHub example, the author of the awesome Testacular.

Chapter 4 Filters

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

Formatting a String with a Currency Filter

Problem

You wish to format the amount of currency with a localized currency label.

Solution

Use the built-in `currency` filter and make sure you load the corresponding locale file for the user's 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" }}</p>
  </body>
</html>
```

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

You can find the complete example on GitHub.

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.

Implementing a Custom Filter to Revert an Input String

Problem

You wish to revert a user's text input.

Solution

Implement a custom filter, which reverts 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;
  };
});
```

You can find the complete example on GitHub.

Discussion

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

Passing Configuration Params to Filters

Problem

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

Solution

Angular filters can be passed as 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;
  };
});
```

You can find the complete example on GitHub.

Discussion

The suffix `!` is passed as an option to the filter function and is appended to the output. Note that we check if an actual input exists since we don't want to render the suffix without any input.

Filtering a List of DOM Nodes

Problem

You wish to filter a `ul` list of names.

Solution

In addition to strings as input, Angular's filters also work 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, which will render all names except `Peter`.

You can find the complete example on GitHub.

Discussion

The filter implementation 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 wish to combine several filters to form 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. First, we will start with the initial array of names. After applying the `exclude` filter, the array contains only `['Anton', 'John']` and afterwards we will sort the names in ascending order.

I leave the implementation of the `sortAscending` filter as an exercise to the reader.

Testing Filters

Problem

You wish 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>
</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',
        inject(function(checkmarkFilter) {
       expect(checkmarkFilter(true)).toBe('\u2713');
       expect(checkmarkFilter(false)).toBe('\u2718');
     }));
   });
});
```

Discussion

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

Chapter 5 Consuming Externals Services

Angular has built-in support for communication with remote HTTP servers. The $http service handles low-level AJAX requests via the browser's XMLHttpRequest object or via JSON with padding (JSONP). The $resource service lets you interact with RESTful data sources and provides high-level behaviors, which naturally map to RESTful resources.

Requesting JSON Data with AJAX

Problem

You wish to fetch JSON data via an AJAX request and render it.

Solution

Implement a controller using the `$http` service 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;
    }).
    error(function(data, status, headers, config) {
      // log error
    });
});
```

You can find the complete example using the angular-seed project on GitHub.

Discussion

The controller defines a dependency to the `$scope` and the `$http` module. An HTTP GET request to the `data/posts.json` endpoint is carried out with the `get` method. It returns a $promise object with a `success` and an `error` method. Once successful, the JSON data is assigned to `$scope.posts` to make it available in the template.

The `$http` service supports the HTTP verbs `get`, `head`, `post`, `put`, `delete`, and `jsonp`. We are going to look into more examples in the following chapters.

The `$http` service automatically adds certain HTTP headers, like for example, `X-Requested-With: XMLHttpRequest`. But you can also set custom HTTP headers by yourself using the `$http.defaults` function:

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

Until now, the `$http` service does not really look particularly special. But if you look into the documentation, you will find a lot of nice features including, for example, request/response transformations, automatically deserializing JSON for you, response caching, response interceptors to handle global error handling, authentication or other preprocessing tasks, and, of course, promise support. We will look into the `$q` service, Angular's promise/deferred service, in a later chapter.

Consuming RESTful APIs

Problem

You wish to consume a RESTful data source.

Solution

Use Angular's high-level `$resource` service. 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">
```

Let us now start by defining the application module and our `Post` model as an Angular service:

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

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

Now we can use our service to retrieve a list of posts inside a controller (for example, HTTP GET /api/posts):

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

Or a specific post by `id` (for example, HTTP GET /api/posts/1):

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

We can create a new post using save (for example, HTTP POST /api/posts):

```
Post.save(data);
```

And we can delete a specific post by `id` (for example, DELETE /api/posts/1):

```
Post.delete({ id: id });
```

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

You can find the complete example on GitHub.

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`, `remove`, and `delete` to work with our resource. In the example above, we implemented several controllers to cover the typical use cases. The `get` and `query` methods expect three arguments, the request parameters, the success callback, and the error callback. The `save` method expects four arguments, the request parameters, the POST data, the success callback, and the error callback.

The `$resource` service currently does not support promises and therefore has a distinctly different interface to the `$http` service. But we don't have to wait very long for it, since work has already started in the 1.1 development branch to introduce promise support for the `$resource` service!

The returned object of a `$resource` query or get function is a `$resource` instance which provides `$save`, `$remove`, and `$delete` methods. This allows you to easily fetch a resource and update it as in the following example:

```
var post = Post.get({ id: 1 }, function() {
  post.title = "My new title";
  post.$save();
});
```

It is important to note that the `get` call immediately returns an empty reference—in our case the `post` variable. Once the data is returned from the server, the existing reference is populated; we can then change our post title and use the `$save` method conveniently.

Note that having an empty reference means that our post will not be rendered in the template. Once the data is returned, though, the view automatically re-renders itself showing the new data.

Configuration

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'
```

Angular seems to expect your service to return a JSON array. Have a look at the following JSON example which wraps a `posts` array in a JSON object:

```
{
  "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;
  });
});
```

We only change the configuration of the `query` action to not expect an array by setting the `isArray` attribute to `false`. Then, in our controller, we can directly access `data.posts`.

It is generally good practice to encapsulate your model and `$resource` usage in an Angular service module and inject that in your controller. This way, you can easily reuse the same model in different controllers and test it more easily.

Consuming JSONP APIs

Problem

You wish to call a JSONP API.

Solution

Use the `$resource` service and configure it to use JSONP. As an example, we will take the Twitter Search API here.

The HTML template lets you enter a search term in an input field; itwill render the search result in a list:

```
<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>
```

The `$resource` configuration can be done in a controller requesting the data:

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

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

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

You can find the complete example on GitHub.

Discussion

The Twitter Search API supports a `callback` attribute for the JSON format as described in their documentation. The `$resource` definition sets the `callback` attribute to `JSON_CALLBACK`, which is a convention from Angular when using JSONP. It is a placeholder that is replaced with the real callback function generated by Angular. Additionally, we configure the `get` method to use `JSONP`. Now, when calling the API, we use the `q` URL parameter to pass the entered `searchTerm`.

Deferred and Promise

Problem

You wish to synchronize multiple asynchronous functions and avoid JavaScript callback hell.

Solution

As an example, we want to call three services in sequence and combine the result of them. Let us start with a nested approach:

```
tmp = [];

$http.get("/app/data/first.json").success(function(data) {
  tmp.push(data);
  $http.get("/app/data/second.json").success(function(data) {
    tmp.push(data);
    $http.get("/app/data/third.json").success(function(data) {
      tmp.push(data);
      $scope.combinedNestedResult = tmp.join(", ");
    });
  });
});
```

We call the `get` function three times to retrieve three JSON documents, each with an array of Strings. We haven't even started adding error handling but already, from using nested callbacks, the code becomes messy and can be simplified using the `$q` service:

```
var first  = $http.get("/app/data/first.json"),
    second = $http.get("/app/data/second.json"),
    third  = $http.get("/app/data/third.json");

$q.all([first, second, third]).then(function(result) {
  var tmp = [];
  angular.forEach(result, function(response) {
    tmp.push(response.data);
  });
  return tmp;
}).then(function(tmpResult) {
  $scope.combinedResult = tmpResult.join(", ");
});
```

You can find the complete example on GitHub.

Discussion

The `all` function combines multiple promises into a single promise and solves our problem quite elegantly.

Let's have a closer look at the `then` method. It is rather contrived but should give you an idea of how to use `then` to sequentially call functions and pass data along. Since the `all` function returns a promise, again we can call `then` on it. By returning the `tmp` variable, it will be passed along to the then function as a `tmpResult` argument.

Before finishing this recipe, let us quickly discuss an example in which we have to create our own deferred object:

```
function deferredTimer(success) {
  var deferred = $q.defer();

  $timeout(function() {
    if (success) {
      deferred.resolve({ message: "This is great!" });
    } else {
      deferred.reject({ message: "Really bad" });
    }
  }, 1000);

  return deferred.promise;
}
```

Using the `defer` method, we create a deferred instance. As an example of an asynchronous operation, we will use the `$timeout` service which will either resolve or reject our operation depending on the Boolean success parameter. The function will immediately return the `promise` and therefore not render any result in our HTML template. After one second, the timer will execute and return our success or failure response.

This `deferredTimer` can be triggered in our HTML template by wrapping it into a function defined on the scope:

```
$scope.startDeferredTimer = function(success) {
  deferredTimer(success).then(
    function(data) {
      $scope.deferredTimerResult = "Successfully finished: " +
        data.message;
    },
    function(data) {
      $scope.deferredTimerResult = "Failed: " + data.message;
    }
  );
};
```

Our `startDeferredTimer` function will get a `success` parameter which it passes along to the `deferredTimer`. The `then` function expects a success and an error callback as arguments which we use to set a scope variable `deferredTimerResult` to show our result.

This is just one of many examples of how promises can simplify your code, but you can find many more examples by looking into Kris Kowal's Q implementation.

Testing Services

Problem

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

Let's have a look again at our example we wish 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() {
    $scope.searchResult = TwitterAPI.get({ q: $scope.searchTerm });
  };
});
```

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.

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

  beforeEach(module("MyApp"));

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

        var mockData = { key: "test" };
        var url = "http://search.twitter.com/search.json?" +
          "callback=JSON_CALLBACK&q=angularjs";
        httpBackend.whenJSONP(url).respond(mockData);
      }
    )
  );

  it('should set searchResult on successful search', function() {
    scope.searchTerm = "angularjs";
    scope.search();
    httpBackend.flush();

    expect(scope.searchResult.key).toBe("test");
  });

});
```

You can find the complete example on GitHub.

Discussion

Since we now have a clear separation between the service and the controller, we can simply inject the `Twitter API` into 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 module for more details.

Chapter 6 URLs, Routing, and Partials

The $location service in Angular.js parses the current browser URL and makes it available to your application. Changes in either the browser address bar or the `$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. Later, we will look at the new HTML 5-based routing.

Client-Side Routing with Hashbang URLs

Problem

You wish the browser address bar to reflect your application's 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 main template uses the `ng-view` directive:

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

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

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

It is set up to render either the `person_list.html` or the `person_details.html` partial depending on the URL. The partial `person_list.html` renders a list of `persons`:

```
<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 `person_details.html` shows more detailed information for a specific `person`:

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

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

This example is based on the angular-seed bootstrap again and will not work without starting the development server.

You can find the complete example on GitHub.

Discussion

Let's give our app a try and open the `index.html` file. The `otherwise` defined route redirects us from `index.html` to `index.html#!/persons`. This is the default behavior in case other `when` conditions don't apply.

Take a closer look at the `index.html#!/persons` URL and note how the hashbang (#!) separates the `index.html` from the dynamic, client-side part `/persons`. By default, Angular would use the hash (#) character but we configured it to use the hashbang instead, following Google's Making AJAX applications crawlable guide.

The `/persons` route loads the `person_list.html` partially 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 resolves to a specific person's details, for example, `/persons/1`. The `person_details.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 `person_details.html` has a back link to `#!persons` which leads back to the `person_list.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 finally, the HTTP request for the partials happens only once and is then cached via `$templateCache` service.

Finally, the hashbang-based routing is client-side only and doesn't require server-side configuration. Let us look into the HTML 5-based approach next.

Using Regular URLs with the HTML 5 History API

Problem

You want good- looking URLs and want to provide server-side support.

Solution

We will use the same example but use the Express framework to serve all content and handle the URL rewriting.

Let us start with the route configuration:

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

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

There are no changes except for 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 though. Our `Express` app will have to serve the partials for us. The initial typical boilerplate for an `Express` app loads the module and creates a server:

```
var express = require('express');
var app     = module.exports = express.createServer();
```

We will skip the configuration here and jump directly to the server-side route definition:

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

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

When supporting HTML 5 routing, our server has to redirect all other URLs to the entry point of our application, the `index` page. First, we define the rendering of the `index` page which contains the `ng-view` directive:

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

Then, the catchall route which redirects to the same page:

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

Let us quickly check the partials again. Note that they use the Jade template engine, which relies on indentation to define the HTML document:

```
p This is the index partial
ul(ng-repeat="person in persons")
  li
    a(href="/persons/{{person.id}}"){{person.name}}
```

The index page creates a list of persons and the show page shows more details:

```
h3 Person Details {{person.name}}
p Age: {{person.age}}
a(href="/persons") Back
```

The person details link `/persons/{{person.id}}` and the back link `/persons` are both now much cleaner, in my opinion, as compared to the hashbang URLs.

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

You can find the complete example on GitHub.

Discussion

If we didn't redirect all requests to the root, what would happen if we were to 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 URL (`/`) 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 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 carry out any server requests. Everything our app needs is loaded once from the server and cached client-side.

If you have a hard time understanding the server implementation, I suggest you read this excellent Express Guide. Additionally, there will be a chapter in this e-book that will go into more detail about how to integrate Angular.js with server-side frameworks.

Using Route Location to Implement a Navigation Menu

Problem

You wish to implement a navigation menu which shows the selected menu item to the user.

Solution

Use the `$location` service in a controller to compare the address bar URL to the navigation menu item the user selected.

The navigation menu is the classic ul/li menu using a class attribute to mark one of the `li` elements as `active`:

```
<body ng-controller="MainCtrl">
  <ul class="menu">
    <li ng-class="menuClass('persons')"><a href="#!persons">Home</a></li>
    <li ng-class="menuClass('help')"><a href="#!help">Help</a></li>
  </ul>
  ...
</body>
```

The controller implements the `menuClass` function:

```
app.controller("MainCtrl", function($scope, $location) {
  $scope.menuClass = function(page) {
    var current = $location.path().substring(1);
    return page === current ? "active" : "";
  };
});
```

You can find the complete example on GitHub.

Discussion

When the user selects a menu item, the client-side navigation will kick in as expected. The `menuClass` function is bound using the `ngClass` directive and updates the CSS class automatically for us depending on the current route.

Using `$location.path()` we get the current route. The `substring` operation removes the leading slash (`/`) and converts `/persons` to `persons`.

Listening on Route Changes to Implement a Login Mechanism

Problem

You wish to ensure that a user has to log in before navigating to protected pages.

Solution

Implement a listener on the `$routeChangeStart` event to track the next route navigation. Redirect to a login page if the user is not yet logged in.

The most interesting part is the implementation of the route change listener:

```
var app = angular.module("MyApp", []).
  config(function($routeProvider, $locationProvider) {
    $routeProvider.
      when("/persons",
        { templateUrl: "partials/index.html" }).
      when("/login",
        { templateUrl: "partials/login.html", controller: "LoginCtrl" }).
      // event more routes here ...
      otherwise( { redirectTo: "/persons" });
  }).
  run(function($rootScope, $location) {
    $rootScope.$on( "$routeChangeStart", function(event, next, current) {
      if ($rootScope.loggedInUser == null) {
        // no logged user, redirect to /login
        if ( next.templateUrl === "partials/login.html") {
        } else {
          $location.path("/login");
        }
      }
    });
  });
```

Next, we will define a login form to enter the username, skipping the password for the sake of simplicity:

```
<form ng-submit="login()">
  <label>Username</label>
  <input type="text" ng-model="username">
  <button>Login</button>
</form>
```

And finally, the login controller, which sets the logged in user and redirects to the person’s URL:

```
app.controller("LoginCtrl", function($scope, $location, $rootScope) {
  $scope.login = function() {
    $rootScope.loggedInUser = $scope.username;
    $location.path("/persons");
  };
});
```

You can find the complete example on GitHub.

Discussion

This is, of course, not a full-fledged login system, so please don't use it in any production system. But it exemplifies how to generally handle access to specific areas of your web app. When you open the app in your browser, you will be redirected to the login app in all cases. Only after you have entered a username can you access the other areas.

The `run` method is defined in Module and is a good place for such a route change listener since it runs only once on initialization after the injector is finished loading all the modules. We check the `loggedInUser` in the `$rootScope`, and if it is not set, we redirect the user to the login page. Note that in order to skip this behavior when already navigating to the login page, we have to explicitly check the next `templateUrl`.

The login controller sets the `$rootScope` to the username and redirects to `/persons`. Generally, I try to avoid using the `$rootScope` since it basically is a kind of global state. But in our case, it fits nicely since there should be a current user globally available.

Chapter 7 Using Forms

Every website eventually uses some kind of form for users to enter data. Angular makes it particularly easy to implement client-side form validations to give immediate feedback for an improved user experience.

Implementing a Basic Form

Problem

You wish 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" novalidate>
      <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>
```

The `novalidate` attribute disables the HTML 5 validations, which are client-side validations supported by modern browsers. In our example, we only want the Angular.js validations running to have complete control over the look and feel.

The controller binds the form data to your user model and implements the `submit()` function:

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

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

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

You can find the complete example on GitHub.

Discussion

The initial idea when using forms would be to implement them in the traditional way by serializing the form data and submitting 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 submission to the server was actually done. The default behavior 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. We are going to look into the behavior of the `form` or `ng-form` directive in the next recipe.

Validating a Form Model Client-Side

Problem

You wish to validate the form client-side using HTML 5 form attributes.

Solution

Angular.js works in tandem with HTML 5 form attributes. Let us start with the same form but let us add some HTML 5 attributes to make the input required:

```
<form name="form" ng-submit="submit()">
  <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 made the input `required` for the firstname.

Let us add more debug output below the form:

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

You can find the complete example on GitHub.

Discussion

When starting with a fresh, empty form, you will notice that Angular adds the CSS class `ng-pristine` and `ng-valid` to the form tag and to each input tag. When editing the form, the `ng-pristine` class will be removed from the changed input field and also from the form tag. It will be replaced by the `ng-dirty` class. This is very useful because it allows you to 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 input fields. To demonstrate this, I've added the `required` HTML 5 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's 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 the `$valid`, `$invalid`, `$error`, `$pristine`, and `$dirty` variables.

For validation, Angular provides built-in directives including `required`, `pattern`, `minlength`, `maxlength`, `min`, and `max`.

Let us use Angular's form integration to actually show validation errors in the next recipe.

Displaying Form Validation Errors

Problem

You wish to show validation 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 element's 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 an 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>
```

You can find the complete example on GitHub.

Discussion

The CSS classes ensure that we initially show the fresh form without any classes. When the user starts typing in some input for the first time, we change it to either green or red. That is a good example of using 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 starts typing for the first time.

Displaying Form Validation Errors with the Twitter Bootstrap Framework

Problem

You wish to display form validation errors but the form is styled using Twitter Bootstrap.

Solution

When using the `.horizontal-form` class, Twitter Bootstrap uses `div` elements to structure label, input fields, and help messages into groups. The group div has the class `control-group` and the actual controls are further nested in another `div` element with the CSS class `controls`. Twitter Bootstrap shows a nice validation status when adding the CSS class `error` on the div with the `control-group` class.

Let us start with the form:

```
<div ng-controller="User">
  <form name="form" ng-submit="submit()" novalidate>

    <div class="control-group" ng-class="error('firstname')">
      <label class="control-label" for="firstname">Firstname</label>
      <div class="controls">
        <input id="firstname" name="firstname" type="text"
          ng-model="user.firstname" placeholder="Firstname" required/>
        <span class="help-block"
          ng-show="form.firstname.$invalid && form.firstname.$dirty">
          First Name is required
        </span>
      </div>
    </div>

    <div class="control-group" ng-class="error('lastname')">
      <label class="control-label" for="lastname">Lastname</label>
      <div class="controls">
        <input id="lastname" name="lastname" type="text"
          ng-model="user.lastname" placeholder="Lastname" required/>
        <span class="help-block"
          ng-show="form.lastname.$invalid && form.lastname.$dirty">
          Last Name is required
        </span>
      </div>
    </div>

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

Note that we use the `ng-class` directive on the `control-group` div. So, let's look at the controller implementation of the `error` function:

```
app.controller("User", function($scope) {
  // ...
  $scope.error = function(name) {
    var s = $scope.form[name];
    return s.$invalid && s.$dirty ? "error" : "";
  };
});
```

The error function gets the input name attribute passed as a string and checks for the `$invalid` and `$dirty` flags to return either the error class or a blank string.

You can find the complete example on GitHub.

Discussion

Again, we check both the invalid and dirty flags because we only show the error message in case the user has actually changed the form. Note that this `ng-class` function usage is pretty typical in Angular since expressions do not support ternary checks.

Only Enabling the Submit Button if the Form is Valid

Problem

You wish 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 is the changed submit button:

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

You can find the complete example on GitHub.

Discussion

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

Note that you have to assign a `name` attribute to the form element, otherwise `form.$invalid` won't be available.

Implementing Custom Validations

Problem

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

Solution

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

Let us look at the template first, with the usage of the `ui-validate` directive:

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

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

And the controller with the `notBlackListed` implementation:

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

app.controller("User", function($scope) {
  $scope.blacklist = ['idiot','loser'];

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

You can find the complete example on GitHub.

Discussion

First we need to explicitly list our module dependency to the Angular UI directives module. Make sure you actually download the JavaScript file and load it via script tag.

Our blacklist contains the words we do not want to accept as user input and the `notBlackListed` function checks to see if the user input matches any of the words defined in the blacklist.

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 to know even more, look at how to implement custom directives yourself in Angular's excellent guide.

Chapter 8 Common User Interface Patterns

Filtering and Sorting a List

Problem

You wish to filter and sort a relatively small list of items, all available on the client.

Solution

For this example, we will render a list of friends using the `ng-repeat` directive. Using the built-in `filter` and `orderBy` filters, we will filter and sort the friends list client-side:

```
<body ng-app="MyApp">
  <div ng-controller="MyCtrl">
    <form class="form-inline">
      <input ng-model="query" type="text"
        placeholder="Filter by" autofocus>
    </form>
    <ul ng-repeat="friend in friends | filter:query | orderBy: 'name' ">
      <li>{{friend.name}}</li>
    </ul>
  </div>
</body>
```

A plain text input field is used to enter the filter query and bound to the `filter`. Any changes are therefore directly used to filter the list.

The controller defines the default friends array:

```
app.controller("MyCtrl", function($scope) {
  $scope.friends = [
    { name: "Peter",   age: 20 },
    { name: "Pablo",   age: 55 },
    { name: "Linda",   age: 20 },
    { name: "Marta",   age: 37 },
    { name: "Othello", age: 20 },
    { name: "Markus",  age: 32 }
  ];
});
```

You can find the complete example on GitHub.

Discussion

Chaining filters is a fantastic way of implementing such a use case, as long as you have all the data available on the client.

The filter Angular.js Filter works on an array and returns a subset of items as a new array. It supports a String, Object, or Function parameter. In this example, we only use the String parameter but, given that the `$scope.friends` is an array of objects, we could think of more complex examples in which we use the Object param. For example:

```
<ul ng-repeat="friend in friends |
  filter: { name: query, age: '20' } |
  orderBy: 'name' ">
  <li>{{friend.name}} ({{friend.age}})</li>
</ul>
```

That way, we can filter by name and age at the same time. And lastly, you could call a function defined in the controller, which does the filtering for you:

```
<ul ng-repeat="friend in friends |
  filter: filterFunction |
  orderBy: 'name' ">
  <li>{{friend.name}} ({{friend.age}})</li>
</ul>

$scope.filterFunction = function(element) {
  return element.name.match(/^Ma/) ? true : false;
};
```

The `filterFunction` must return either `true` or `false`. In this example, we use a regular expression on the name starting with `Ma` to filter the list.

Pagination through Client-Side Data

Problem

You have a table of data completely client-side and want to paginate through the data.

Solution

Use an HTML table element with the `ng-repeat` directive to render only the items for the current page. All the pagination logic should be handled in a custom filter and controller implementation.

Let us start with the template using Twitter Bootstrap for the table and pagination elements:

```
<div ng-controller="PaginationCtrl">
  <table class="table table-striped">
    <thead>
      <tr>
        <th>Id</th>
        <th>Name</th>
        <th>Description</th>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="item in items |
        offset: currentPage*itemsPerPage |
        limitTo: itemsPerPage">
        <td>{{item.id}}</td>
        <td>{{item.name}}</td>
        <td>{{item.description}}</td>
      </tr>
    </tbody>
    <tfoot>
      <td colspan="3">
        <div class="pagination">
          <ul>
            <li ng-class="prevPageDisabled()">
              <a href ng-click="prevPage()">« Prev</a>
            </li>
            <li ng-repeat="n in range()"
              ng-class="{active: n == currentPage}" ng-click="setPage(n)">
              <a href="#">{{n+1}}</a>
            </li>
            <li ng-class="nextPageDisabled()">
              <a href ng-click="nextPage()">Next »</a>
            </li>
          </ul>
        </div>
      </td>
    </tfoot>
  </table>
</div>
```

The `offset` Filter is responsible for selecting the subset of items for the current page. It uses the `slice` function on the array given the start param as the index:

```
app.filter('offset', function() {
  return function(input, start) {
    start = parseInt(start, 10);
    return input.slice(start);
  };
});
```

The controller manages the actual `$scope.items` array and handles the logic for enabling/disabling the pagination buttons:

```
app.controller("PaginationCtrl", function($scope) {

  $scope.itemsPerPage = 5;
  $scope.currentPage = 0;
  $scope.items = [];

  for (var i=0; i<50; i++) {
    $scope.items.push({
      id: i, name: "name "+ i, description: "description " + i
    });
  }

  $scope.prevPage = function() {
    if ($scope.currentPage > 0) {
      $scope.currentPage--;
    }
  };

  $scope.prevPageDisabled = function() {
    return $scope.currentPage === 0 ? "disabled" : "";
  };

  $scope.pageCount = function() {
    return Math.ceil($scope.items.length/$scope.itemsPerPage)-1;
  };

  $scope.nextPage = function() {
    if ($scope.currentPage < $scope.pageCount()) {
      $scope.currentPage++;
    }
  };

  $scope.nextPageDisabled = function() {
    return $scope.currentPage === $scope.pageCount() ? "disabled" : "";
  };

});
```

You can find the complete example on GitHub.

Discussion

The initial idea of this pagination solution can be best explained by looking into the usage of `ng-repeat` to render the table rows for each item:

```
<tr ng-repeat="item in items |
  offset: currentPage*itemsPerPage |
  limitTo: itemsPerPage">
  <td>{{item.id}}</td>
  <td>{{item.name}}</td>
  <td>{{item.description}}</td>
</tr>
```

The `offset` filter uses the `currentPage*itemsPerPage` to calculate the offset for the array slice operation. This will generate an array from the offset to the end of the array. Then, use the built-in `limitTo` filter to subset the array to the number of `itemsPerPage`. All this is done on the client side, with filters only.

The controller is responsible for supporting a `nextPage` and `prevPage` action and the accompanying functions to check the disabled state of these actions via the `ng-class` directive: `nextPageDisabled` and `prevPageDisabled`. The `prevPage` function first checks if it has not reached the first page yet before decrementing the `currentPage,` and the `nextPage` does the same for the last page;the same logic is applied for the disabled checks.

This example is already quite involved and I intentionally omitted an explanation of the rendering of links between the previous and next buttons. The full implementation is online though for you to investigate.

Pagination through Server-Side Data

Problem

You wish to paginate through a large, server-side result set.

Solution

You cannot use the previous method with a filter since that would require all data to be available on the client. Instead, we use an implementation with a controller only.

The template has not changed much, only the `ng-repeat` directive looks simpler now:

```
<tr ng-repeat="item in pagedItems">
  <td>{{item.id}}</td>
  <td>{{item.name}}</td>
  <td>{{item.description}}</td>
</tr>
```

In order to simplify the example, we will fake a server-side service by providing an Angular service implementation for the items:

```
app.factory("Item", function() {

  var items = [];
  for (var i=0; i<50; i++) {
    items.push({
      id: i, name: "name "+ i, description: "description " + i
    });
  }

  return {
    get: function(offset, limit) {
      return items.slice(offset, offset+limit);
    },
    total: function() {
      return items.length;
    }
  };
});
```

The service manages a list of items, and has methods for retrieving a subset of items for a given offset including limit and total number of items.

The controller uses dependency injection to access the `Item` service and contains almost the same methods as our previous recipe:

```
app.controller("PaginationCtrl", function($scope, Item) {

  $scope.itemsPerPage = 5;
  $scope.currentPage = 0;

  $scope.prevPage = function() {
    if ($scope.currentPage > 0) {
      $scope.currentPage--;
    }
  };

  $scope.prevPageDisabled = function() {
    return $scope.currentPage === 0 ? "disabled" : "";
  };

  $scope.nextPage = function() {
    if ($scope.currentPage < $scope.pageCount() - 1) {
      $scope.currentPage++;
    }
  };

  $scope.nextPageDisabled = function() {
    return $scope.currentPage === $scope.pageCount() - 1 ? "disabled" : "";
  };

  $scope.pageCount = function() {
    return Math.ceil($scope.total/$scope.itemsPerPage);
  };

  $scope.$watch("currentPage", function(newValue, oldValue) {
    $scope.pagedItems =
      Item.get(newValue*$scope.itemsPerPage, $scope.itemsPerPage);
    $scope.total = Item.total();
  });

});
```

You can find the complete example on GitHub.

Discussion

When you select the next/previous page, you will change the `$scope.currentPage` value and the `$watch` function will be triggered. It fetches fresh items for the current page as well as the total number of items. So, on the client side, we only have five items available as defined in `itemsPerPage` and, when paginating, we throw away the items of the previous page and fetch new items.

If you want to try this with a real backend, you only have to swap out the `Item` service implementation.

Pagination Using Infinite Results

Problem

You wish to paginate through server-side data with a "Load More" button, which just keeps appending more data until no more data is available.

Solution

Let's start by looking at how the item table is rendered with the `ng-repeat` directive:

```
<div ng-controller="PaginationCtrl">
  <table class="table table-striped">
    <thead>
      <tr>
        <th>Id</th>
        <th>Name</th>
        <th>Description</th>
      </tr>
    </thead>
    <tbody>
      <tr ng-repeat="item in pagedItems">
        <td>{{item.id}}</td>
        <td>{{item.name}}</td>
        <td>{{item.description}}</td>
      </tr>
    </tbody>
    <tfoot>
      <td colspan="3">
        <button class="btn" href="#" ng-class="nextPageDisabledClass()"
          ng-click="loadMore()">Load More</button>
      </td>
    </tfoot>
  </table>
</div>
```

The controller uses the same `Item` Service as used for the previous recipe and handles the logic for the "Load More" button:

```
app.controller("PaginationCtrl", function($scope, Item) {

  $scope.itemsPerPage = 5;
  $scope.currentPage = 0;
  $scope.total = Item.total();
  $scope.pagedItems = Item.get($scope.currentPage*$scope.itemsPerPage,
    $scope.itemsPerPage);

  $scope.loadMore = function() {
    $scope.currentPage++;
    var newItems = Item.get($scope.currentPage*$scope.itemsPerPage,
      $scope.itemsPerPage);
    $scope.pagedItems = $scope.pagedItems.concat(newItems);
  };

  $scope.nextPageDisabledClass = function() {
    return $scope.currentPage === $scope.pageCount()-1 ? "disabled" : "";
  };

  $scope.pageCount = function() {
    return Math.ceil($scope.total/$scope.itemsPerPage);
  };

});
```

You can find the complete example on GitHub.

Discussion

The solution is actually similar to the previous recipe and again uses only a controller. The `$scope.pagedItems` is retrieved initially to render the first five items.

When pressing the "Load More" button, we fetch another set of items incrementing the `currentPage` to change the `offset` of the `Item.get` function. The new items will be concatenated with the existing items using the array `concat` function. The changes to `pagedItems` will be automatically rendered by the `ng-repeat` directive.

The `nextPageDisabledClass` checks whether or not there is more data available by calculating the total number of pages in `pageCount` and comparing that to the current page.

Displaying a Flash Notice/Failure Message

Problem

You wish to display a flash confirmation message after a user submitted a form successfully.

Solution

In a web framework like Ruby on Rails, the form submission will lead to a redirect with the flash confirmation message, relying on the browser session. For our client-side approach, we bind to route changes and manage a queue of flash messages.

In our example, we use a homepage with a form, and on form submit we navigate to another page and show the flash message. We use the `ng-view` directive and define the two pages as script tags here:

```
<body ng-app="MyApp" ng-controller="MyCtrl">

  <ul class="nav nav-pills">
    <li><a href="#/">Home</a></li>
    <li><a href="#/page">Next Page</a></li>
  </ul>

  <div class="alert" ng-show="flash.getMessage()">
    <b>Alert!</b>
    <p>{{flash.getMessage()}}</p>
  </div>

  <ng-view></ng-view>

  <script type="text/ng-template" id="home.html">
    <h3>Home</h3>

    <form ng-submit="submit(message)" class="form-inline">
      <input type="text" ng-model="message" autofocus>
      <button class="btn">Submit</button>
    </form>

  </script>

  <script type="text/ng-template" id="page.html">
    <h3>Next Page</h3>

  </script>

</body>
```

Note that the flash message (just like the navigation) is always shown but conditionally hidden depending on whether or not there is a flash message available.

The route definition defines the pages; nothing new here for us:

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

app.config(function($routeProvider) {
  $routeProvider.
    when("/home", { templateUrl: "home.html" }).
    when("/page", { templateUrl: "page.html" }).
    otherwise({ redirectTo: "/home" });
});
```

The interesting part is the `flash` service, which handles a queue of messages and listens for route changes to provide a message from the queue to the current page:

```
app.factory("flash", function($rootScope) {
  var queue = [];
  var currentMessage = "";

  $rootScope.$on("$routeChangeSuccess", function() {
    currentMessage = queue.shift() || "";
  });

  return {
    setMessage: function(message) {
      queue.push(message);
    },
    getMessage: function() {
      return currentMessage;
    }
  };
});
```

The controller handles the form submit and navigates to the other page:

```
app.controller("MyCtrl", function($scope, $location, flash) {
  $scope.flash = flash;
  $scope.message = "Hello World";

  $scope.submit = function(message) {
    flash.setMessage(message);
    $location.path("/page");
  }
});
```

The flash service is dependency-injected into the controller and made available to the scope since we want to use it in our template.

When you press the submit button, you will be navigated to the other page and see the flash message. Note that using the navigation to go back and forth between pages doesn't show the flash message.

You can find the complete example on GitHub.

Discussion

The controller uses the `setMessage` function of the `flash` service and the service stores the message in an array called `queue`. When the controller uses `$location` service to navigate the service `routeChangeSuccess, the` listener will be called and can retrieve the message from the queue.

In the template, we use `ng-show` to hide the div element with the flash messaging using `flash.getMessage()`.

Since this is a service, it can be used anywhere in your code and it will show a flash message on the next route change.

Editing Text InPlace Using HTML 5 ContentEditable

Problem

You wish to make a div element editable in place using the HTML 5 `contenteditable` attribute.

Solution

Implement a directive for the `contenteditable` attribute and use `ng-model` for data binding.

In this example, we use a div and a paragraph to render the content:

```
<div contenteditable ng-model="text"></div>
<p>{{text}}</p>
```

The directive is especially interesting since it uses `ng-model` instead of custom attributes:

```
app.directive("contenteditable", function() {
  return {
    restrict: "A",
    require: "ngModel",
    link: function(scope, element, attrs, ngModel) {

      function read() {
        ngModel.$setViewValue(element.html());
      }

      ngModel.$render = function() {
        element.html(ngModel.$viewValue || "");
      };

      element.bind("blur keyup change", function() {
        scope.$apply(read);
      });
    }
  };
});
```

You can find the complete example on GitHub.

Discussion

The directive is restricted for usage as an HTML attribute since we want to use the HTML 5 `contenteditable` attribute as it is instead of defining a new HTML element.

It requires the `ngModel` controller for data binding in conjunction with the link function. The implementation binds an event listener, which executes the `read` function with apply. This ensures that, even though we call the `read` function from within a DOM event handler, we notify Angular about it.

The `read` function updates the model based on the view's user input. And the `$render` function is doing the same in the other direction, updating the view for us whenever the model changes.

The directive is surprisingly simple, leaving the `ng-model` aside. But without the `ng-model` support, we would have to come up with our own model-attribute handling, which would not be consistent with other directives.

Displaying a Modal Dialog

Problem

You wish to use a modal dialog using the Twitter Bootstrap framework. A dialog is called modal when it is blocking the rest of your web application until it is closed.

Solution

Use the `angular-ui` module's nice `modal` plug-in, which directly supports Twitter Bootstrap.

The template defines a button to open the modal and the modal code itself:

```
<body ng-app="MyApp" ng-controller="MyCtrl">

  <button class="btn" ng-click="open()">Open Modal</button>

  <div modal="showModal" close="cancel()">
    <div class="modal-header">
        <h4>Modal Dialog</h4>
    </div>
    <div class="modal-body">
        <p>Example paragraph with some text.</p>
    </div>
    <div class="modal-footer">
      <button class="btn btn-success" ng-click="ok()">Okay</button>
      <button class="btn" ng-click="cancel()">Cancel</button>
    </div>
  </div>

</body>
```

Note that, even though we don't specify it explicitly, the modal dialog is hidden initially via the `modal` attribute. The controller only handles the button click and the `showModal` value used by the `modal` attribute.

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

$scope.open = function() {
  $scope.showModal = true;
};

$scope.ok = function() {
  $scope.showModal = false;
};

$scope.cancel = function() {
  $scope.showModal = false;
};
```

Do not forget to download and include the angular-ui.js file in a script tag. The module dependency is defined directly to "ui.bootstrap.modal". The full example is available on GitHub, including the angular-ui module.

You can find the complete example on GitHub.

Discussion

The modal as defined in the template is straight from the Twitter Bootstrap documentation. We can control the visibility with the `modal` attribute. Additionally, the `close` attribute defines a `close` function, which is called whenever the dialog is closed. Note that this could happen if the user presses the `escape` key or clicks outside the modal.

Our own cancel button uses the same function to close the modal manually, whereas the okay button uses the `ok` function. This makes it easy for us to distinguish between a user who simply cancelled the modal or one who actually pressed the okay button.

Displaying a Loading Spinner

Problem

You wish to display a loading spinner while waiting for an AJAX request to be finished.

Solution

We will use the Twitter Search API for our example to render a list of search results. When pressing the button, the AJAX request is run and the spinner image should be shown until the request is done:

```
<body ng-app="MyApp" ng-controller="MyCtrl">

  <div>
    <button class="btn" ng-click="load()">Load Tweets</button>
    <img id="spinner" ng-src="img/spinner.gif" style="display:none;">
  </div>

  <div>
    <ul ng-repeat="tweet in tweets">
      <li>
        <img ng-src="{{tweet.profile_image_url}}" alt="">
        &nbsp; {{tweet.from_user}}
        {{tweet.text}}
      </li>
    </ul>
  </div>

</body>
```

An Angular.js interceptor for all AJAX calls is used, which allows you to execute code before the actual request is started and when it is finished:

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

app.config(function ($httpProvider) {
  $httpProvider.responseInterceptors.push('myHttpInterceptor');

  var spinnerFunction = function spinnerFunction(data, headersGetter) {
    $("#spinner").show();
    return data;
  };

  $httpProvider.defaults.transformRequest.push(spinnerFunction);
});

app.factory('myHttpInterceptor', function ($q, $window) {
  return function (promise) {
    return promise.then(function (response) {
      $("#spinner").hide();
      return response;
    }, function (response) {
      $("#spinner").hide();
      return $q.reject(response);
    });
  };
});
```

Note that we use jQuery to show the spinner in the configuration step and hide the spinner in the interceptor.

Additionally, we use a controller to handle the button click and execute the search request:

```
app.controller("MyCtrl", function($scope, $resource, $rootScope) {

  $scope.resultsPerPage = 5;
  $scope.page = 1;
  $scope.searchTerm = "angularjs";

  $scope.twitter = $resource('http://search.twitter.com/search.json',
    { callback:'JSON_CALLBACK',
      page: $scope.page,
      rpp: $scope.resultsPerPage,
      q: $scope.searchTerm },
    { get: { method:'JSONP' } });

  $scope.load = function() {
    $scope.twitter.get(function(data) {
      $scope.tweets = data.results;
    });
  };
});
```

Don't forget to add `ngResource` to the module and load it via script tag.

You can find the complete example on GitHub.

Discussion

The template is the easy part of this recipe since it renders a list of tweets using the `ng-repeat` directive. Let us jump straight to the interceptor code.

The interceptor is implemented using the factory method and attaches itself to the promise function of the AJAX response to hide the spinner on success or failure. Note that, on failure, we use the `reject` function of the $q service, Angular's promise/deferred implementation.

Now, in the `config` method, we add our interceptor to the list of responseInterceptors of `$httpProvider` to register it properly. In a similar manner, we add the `spinnerFunction` to the default `transformRequest` list in order to call it before each AJAX request.

The controller is responsible for using a `$resource` object and handling the button click with the `load` function. We are using JSONP here to allow this code to be executed locally even though it is served by a different domain.

Chapter 9 Backend Integration with Ruby on Rails

In this chapter, we will have a look into solving common problems when combining Angular.js with the Ruby on Rails framework. The examples used in this chapter are based on an example application to manage a list of contacts.

Consuming REST APIs

Problem

You wish to consume a JSON REST API implemented in your Rails application.

Solution

Using the `$resource` service is a great start and it can be tweaked to feel more natural to a Rails developer by configuring the methods in accordance with the Rails actions:

```
app.factory("Contact", function($resource) {
  return $resource("/api/contacts/:id", { id: "@id" },
    {
      'create':  { method: 'POST' },
      'index':   { method: 'GET', isArray: true },
      'show':    { method: 'GET', isArray: false },
      'update':  { method: 'PUT' },
      'destroy': { method: 'DELETE' }
    }
  );
});
```

We can now fetch a list of contacts using `Contact.index()` and a single contact with `Contact.show(id)`. These actions can be directly mapped to the `ContactsController` actions in your Rails backend:

```
class ContactsController < ApplicationController
  respond_to :json

  def index
    @contacts = Contact.all
    respond_with @contacts
  end

  def show
    @contact = Contact.find(params[:id])
    respond_with @contact
  end

  ...
end
```

The Rails action controller uses a `Contact` ActiveRecord model with the usual contact attributes like firstname, lastname, age, etc. By specifying `respond_to :json`, the controller only responds to the JSON resource format; we can use `respond_with` to automatically transform the `Contact` model to a JSON response.

The route definition uses the Rails default resource routing and an `api` scope to separate the API requests from other requests:

```
Contacts::Application.routes.draw do
  scope "api" do
    resources :contacts
  end
end
```

This will generate paths like, for example, `api/contacts` and `api/contacts/:id` for the HTTP GET method.

You can find the complete example on GitHub.

Discussion

If you want to get up to speed with Ruby on Rails, I suggest that you look into the Rails Guides, which will help you understand how all the pieces fit together.

Rails Security using Authenticity Token

The example code above works nicely until we use the HTTP methods `POST`, `PUT`, and `DELETE` with the resource. As a security mechanism, Rails expects an authenticity token to prevent a Cross Site Request Forgery (CSRF) attack. We need to submit an additional HTTP header `X-CSRF-Token` with the token. It is conveniently rendered in the HTML meta tag `csrf-token` by Rails. Using jQuery, we can fetch that meta tag definition and configure the `$httpProvider` appropriately:

```
var app = angular.module("Contacts", ["ngResource"]);
app.config(function($httpProvider) {
  $httpProvider.defaults.headers.common['X-CSRF-Token'] =
    $('meta[name=csrf-token]').attr('content');
});
```

Rails JSON Response Format

If you are using a Rails version prior to 3.1, you'll notice that the JSON response will use a `contact` namespace for the model attributes, which breaks your Angular.js code. To disable this behavior, you can configure your Rails app accordingly:

```
ActiveRecord::Base.include_root_in_json = false
```

There are still inconsistencies between the Ruby and JavaScript world. For example, in Ruby we use underscored attribute names (display_name), whereas in JavaScript we use camelCase (displayName).

There is a custom `$resource` implementation, angularjs-rails-resource, available to streamline consuming Rails resources. It uses transformers and inceptors to rename the attribute fields and handles the root wrapping behavior for you.

Implementing Client-Side Routing

Problem

You wish to use client-side routing in conjunction with a Ruby on Rails backend.

Solution

Every request to the backend should initially render the complete page in order to load our Angular app. Only then will the client-side rendering take over. Let us first have a look at the route definition for this “catchall” route:

```
Contacts::Application.routes.draw do
  root :to => "layouts#index"
  match "*path" => "layouts#index"
end
```

It uses Route Globbing to match all URLs and defines a root URL. Both will be handled by a layout controller with the sole purpose of rendering the initial layout:

```
class LayoutsController < ApplicationController
  def index
    render "layouts/application"
  end
end
```

The actual layout template defines our `ng-view` directive and resides in `app/views/layouts/application.html`—nothing new here. So, let's skip ahead to the Angular route definition in `app.js.erb`:

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

app.config(function($routeProvider, $locationProvider) {
  $locationProvider.html5Mode(true);
  $routeProvider
    .when("/contacts",
      { templateUrl: "<%= asset_path('contacts/index.html') %> ",
        controller: "ContactsIndexCtrl" })
    .when("/contacts/new",
      { templateUrl: "<%= asset_path('contacts/edit.html') %> ",
        controller: "ContactsEditCtrl" })
    .when("/contacts/:id",
      { templateUrl: "<%= asset_path('contacts/show.html') %> ",
        controller: "ContactsShowCtrl" })
    .when("/contacts/:id/edit",
      { templateUrl: "<%= asset_path('contacts/edit.html') %> ",
        controller: "ContactsEditCtrl" })
    .otherwise({ redirectTo: "/contacts" });
});
```

We set the `$locationProvider` to use the HTML 5 mode and define our client-side routes for listing, showing, editing and creating new contacts.

You can find the complete example on GitHub.

Discussion

Let us have a look into the route definition again. First of all, the file name ends with `erb` since it uses ERB tags in the JavaScript file, courtesy of the Rails Asset Pipeline. The `asset_path` method is used to retrieve the URL to the HTML partials since it will change depending on the environment. On production, the file name contains an MD5 checksum and the actual ERB output will change from `/assets/contacts/index.html` to `/assets/contacts/index-7ce113b9081a20d93a4a86e1aacce05f.html`. If your Rails app is configured to use an asset host, the path will, in fact, be absolute.

Validating Forms Server-Side

Problem

You wish to validate forms using a server-side REST API provided by Rails.

Solution

Rails already provides out-of-the-box model validation support. Let us start with the Contact ActiveRecord model:

```
class Contact < ActiveRecord::Base
  attr_accessible :age, :firstname, :lastname

  validates :age, :numericality => {
    :only_integer => true, :less_than_or_equal_to => 50 }
end
```

It defines a validation on the `age` attribute. It must be an integer, and less than or equal to 50 years.

In the `ContactsController,` we can use that to make sure the REST API returns proper error messages. As an example, let us look into the `create` action:

```
class ContactsController < ApplicationController
  respond_to :json

  def create
    @contact = Contact.new(params[:contact])
    if @contact.save
      render json: @contact, status: :created, location: @contact
    else
      render json: @contact.errors, status: :unprocessable_entity
    end
  end

end
```

On success it will render the contact model using a JSON presentation, and on failure it will return all validation errors transformed to JSON. Let us have a look at an example JSON response:

```
{ "age": ["must be less than or equal to 50"] }
```

It is a hash with an entry for each attribute, with validation errors. The value is an array of Strings since there might be multiple errors at the same time.

Let us move on to the client side of our application. The Angular.js contact `$resource` calls the create function and passes the failure callback function:

```
Contact.create($scope.contact, success, failure);

function failure(response) {
  _.each(response.data, function(errors, key) {
    _.each(errors, function(e) {
      $scope.form[key].$dirty = true;
      $scope.form[key].$setValidity(e, false);
    });
  });
}
```

Note that ActiveRecord attributes can have multiple validations defined. That is why the `failure` function iterates through each validation entry and each error, and uses `$setValidity` and `$dirty` to mark the form fields as invalid.

Now we are ready to show some feedback to our users, using the same approach discussed already in the forms chapter:

```
<div class="control-group" ng-class="errorClass('age')">
  <label class="control-label" for="age">Age</label>
  <div class="controls">
    <input ng-model="contact.age" type="text" name="age"
      placeholder="Age" required>
    <span class="help-block"
      ng-show="form.age.$invalid && form.age.$dirty">
      {{errorMessage('age')}}
    </span>
  </div>
</div>
```

The `errorClass` function adds the `error` CSS class if the form field is invalid and dirty. This will render the label, input field, and the help block in red:

```
$scope.errorClass = function(name) {
  var s = $scope.form[name];
  return s.$invalid && s.$dirty ? "error" : "";
};
```

The `errorMessage` will print a more detailed error message and is defined in the same controller:

```
$scope.errorMessage = function(name) {
  result = [];
  _.each($scope.form[name].$error, function(key, value) {
    result.push(value);
  });
  return result.join(", ");
};
```

It iterates over each error message and creates a comma-separated string out of it.

You can find the complete example on GitHub.

Discussion

Finally, the `errorMessage` handling is, of course, pretty primitive. A user would expect a localized failure message instead of this technical presentation. The Rails Internationalization Guide describes how to translate validation error messages in Rails, and might prove helpful for further use in your client-side code.

Chapter 10 Backend Integration with Node Express

In this chapter, we will have a look into solving common problems when combining Angular.js with the Node.js Express framework. The examples used in this chapter are based on a Contacts app to manage a list of contacts. As an extra, we use MongoDB as a backend for our contacts since it requires further customization to make it work in conjunction with Angular's `$resource` service.

Consuming REST APIs

Problem

You wish to consume a JSON REST API implemented in your Express application.

Solution

Using the `$resource` service, we will begin by defining our Contact model and all RESTful actions:

```
app.factory("Contact", function($resource) {
  return $resource("/api/contacts/:id", { id: "@_id" },
    {
      'create':  { method: 'POST' },
      'index':   { method: 'GET', isArray: true },
      'show':    { method: 'GET', isArray: false },
      'update':  { method: 'PUT' },
      'destroy': { method: 'DELETE' }
    }
  );
});
```

We can now fetch a list of contacts using `Contact.index()` and a single contact with `Contact.show(id)`. These actions can be directly mapped to the API routes defined in `app.js`:

```
var express = require('express'),
        api = require('./routes/api');

var app = module.exports = express();

app.get('/api/contacts', api.contacts);
app.get('/api/contacts/:id', api.contact);
app.post('/api/contacts', api.createContact);
app.put('/api/contacts/:id', api.updateContact);
app.delete('/api/contacts/:id', api.destroyContact);
```

I like to keep routes in a separate file (`routes/api.js`) and just reference them in `app.js` in order to keep it small. The API implementation first initializes the Mongoose library and defines a schema for our Contact model:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/contacts_database');

var contactSchema = mongoose.Schema({
  firstname: 'string', lastname: 'string', age: 'number'
});
var Contact = mongoose.model('Contact', contactSchema);
```

We can now use the `Contact` model to implement the API. Let's start with the index action:

```
exports.contacts = function(req, res) {
  Contact.find({}, function(err, obj) {
    res.json(obj)
  });
};
```

Skipping the error handling, we retrieve all contacts with the `find` function provided by Mongoose and render the result in the JSON format. The show action is pretty similar, except it uses `findOne` and the `id` from the URL parameter to retrieve a single contact.

```
exports.contact = function(req, res) {
  Contact.findOne({ _id: req.params.id }, function(err, obj) {
    res.json(obj);
  });
};
```

As a final example, we will create a new Contact instance passing in the request body and call the `save` method to persist it:

```
exports.createContact = function(req, res) {
  var contact = new Contact(req.body);
  contact.save();
  res.json(req.body);
};
```

You can find the complete example on GitHub.

Discussion

Let’s have a look again at the example for the contact function, which retrieves a single Contact. It uses `_id` instead of `id` as the parameter for the `findOne` function. This underscore is intentional and used by MongoDB for its auto-generated IDs. In order to automatically map from `id` to the `_id` parameter, we used a nice trick of the `$resource` service. Take a look at the second parameter of the Contact `$resource` definition: `{ id: "@_id" }`. Using this parameter, Angular will automatically set the URL parameter `id` based on the value of the model attribute `_id`.

Implementing Client-Side Routing

Problem

You wish to use client-side routing in conjunction with an Express backend.

Solution

Every request to the backend should initially render the complete layout in order to load our Angular app. Only then will the client-side rendering take over. Let us first have a look at the route definition for this “catchall” route in our `app.js`:

```
var express = require('express'),
     routes = require('./routes');

app.get('/', routes.index);
app.get('*', routes.index);
```

It uses the wildcard character to catch all requests in order to get processed with the `routes.index` module. Additionally, it defines the route to use the same module. The module again resides in `routes/index.js`:

```
exports.index = function(req, res){
  res.render('layout');
};
```

The implementation only renders the layout template. It uses the Jade template engine:

```
!!!
html(ng-app="myApp")
  head
    meta(charset='utf8')
    title Angular Express Seed App
    link(rel='stylesheet', href='/css/bootstrap.css')
  body
    div
      ng-view

    script(src='js/lib/angular/angular.js')
    script(src='js/lib/angular/angular-resource.js')
    script(src='js/app.js')
    script(src='js/services.js')
    script(src='js/controllers.js')
```

Now that we can actually render the initial layout, we can get started with the client-side routing definition in `app.js`:

```
var app = angular.module('myApp', ["ngResource"]).
  config(['$routeProvider', '$locationProvider',
    function($routeProvider, $locationProvider) {
      $locationProvider.html5Mode(true);
      $routeProvider
        .when("/contacts", {
          templateUrl: "partials/index.jade",
          controller: "ContactsIndexCtrl" })
        .when("/contacts/new", {
          templateUrl: "partials/edit.jade",
          controller: "ContactsEditCtrl" })
        .when("/contacts/:id", {
          templateUrl: "partials/show.jade",
          controller: "ContactsShowCtrl" })
        .when("/contacts/:id/edit", {
          templateUrl: "partials/edit.jade",
          controller: "ContactsEditCtrl" })
        .otherwise({ redirectTo: "/contacts" });
    }
  ]
);
```

We define route definitions to list, show and edit contacts, and use a set of partials and corresponding controllers. In order for the partials to get loaded correctly, we need to add another express route in the backend, which serves all these partials:

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

It uses the name of the partial as an URL param and renders the partial with the given name from the `partial` directory. Keep in mind that you must define that route before the catchall route, otherwise it will not work.

You can find the complete example on GitHub.

Discussion

Compared to Rails, the handling of partials is explicit by defining a route for partials. On the other hand, it is quite nice to be able to use Jade templates for our partials, too.