CHAPTER 4

Forms, Inputs, and Services

In the previous chapters, we first covered the most basic AngularJS directives and dealt with creating controllers and getting our data from the controllers into the UI. We then looked at how to write tests for the same, using Karma and Jasmine. In this chapter, we will build on the work from Chapter 2 and work on getting the user's data out of forms in the UI into our controller so that we can then send it to the server, validate it, or do whatever else we might need to.

We will then get into using AngularJS services, and see how we can leverage some of the common existing services as well as create our own. We will also briefly cover when and why you should create AngularJS services.

Working with ng-model

In the previous chapter, we saw the `ng-bind` directive, or its equivalent double-curly `{{ }}` notation, which allowed us to take the data from our controllers and display it in the UI. That gives us our one-way data-binding, which is powerful in its own regard. But most applications we develop also have user interaction, and parts where the user has to feed in data. From registration forms to profile information, forms are a staple of web applications, and AngularJS provides the `ng-model` directive for us to deal with inputs and two-way data-binding:

```
<!-- File: chapter4/simple-ng-model.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <input type="text" ng-model="ctrl.username"/>
  You typed {{ctrl.username}}

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
```

```
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      this.username = 'nothing';
    }]);
</script>
</body>
</html>
```

In this example, we define a controller with a variable exposed on its instance called `username`. Now, we get its value out into the HTML using the `ng-controller` and the double-curly syntax for one-way data-binding. What we have introduced in addition is an `input` element. It is a plain text box, but on it we have attached the `ng-model` directive. We pointed the value for the `ng-model` at the same `username` variable on the `MainCtrl`. This accomplishes the following things:

- When the HTML is instantiated and the controller is attached, it gets the current value (in this case, *`nothing`* as a string) and displays it in our UI.
- When the user types, updates, or changes the value in the input box, it updates the model in our controller.
- When the value of the variable changes in the controller (whether because it came from the server, or due to some internal state change), the input field gets the value updated automatically.

The beauty of this is twofold:

- If we need to update the form element in the UI, all we need to do is update the value in the controller. No need to go looking for input fields by IDs or CSS class selectors; just update the model.
- If we need to get the latest value that the user entered into the form or input to validate or send to the server, we just need to grab it from our controller. It will have the latest value in it.

Now let's add some complexity, and actually deal with forms. Let's see if we can bring this concept together with an example:

```
<!-- File: chapter4/simple-ng-model-2.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <input type="text" ng-model="ctrl.username">
  <input type="password" ng-model="ctrl.password">
  <button ng-click="ctrl.change()">Change Values</button>
  <button ng-click="ctrl.submit()">Submit</button>
```

```
<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.change = function() {
        self.username = 'changed';
        self.password = 'password';
      };
      self.submit = function() {
        console.log('User clicked submit with ',
            self.username, self.password);
      };
    }]);
</script>
</body>
</html>
```

We introduced one more input field, which is bound to a field called `password` on the controller's instance. And we added two buttons:

- The first button, Change Values, is to simulate the server sending some data that needs to be updated in the UI. All it does is reassign the values to the username and password fields in the controller with the latest values.
- The second button, Submit, simulates submitting the form to the server. All it does for now is log the value to the console.

The most important thing in both of these is that the controller never reached out into the UI. There was no jQuery selector, no `findElementById`, or anything like that. When we need to update the UI, we just update the model fields in the controller. When we need to get the latest and greatest value, we just grab it from the controller. Again, this is the AngularJS way.

Let's now build on this, and see how we can integrate and work with forms in AngularJS.

Working with Forms

When we work with forms in AngularJS, we heavily leverage the `ng-model` directive to get our data into and out of the form. In addition to the data-binding, it is also recommended to structure your model and bindings in such a way to reduce your own effort, as well as the lines of code you write. Let's take a look at an example:

```
<!-- File: chapter4/simple-form.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">
```

```
  <form ng-submit="ctrl.submit()">
    <input type="text" ng-model="ctrl.user.username">
    <input type="password" ng-model="ctrl.user.password">
    <input type="submit" value="Submit">
  </form>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.submit = function() {
        console.log('User clicked submit with ', self.user);
      };
    }]);
</script>
</body>
</html>
```

We are still using the same two input fields as last time, but we made a few changes:

- We wrapped our text fields and button inside a form. And instead of an `ng-click` on the button, we added an `ng-submit` directive on the form itself. The `ng-submit` directive has a few advantages over having an `ng-click` on a button when it comes to forms. A form submit event can be triggered in multiple ways: clicking the Submit button, or hitting Enter on a text field. The `ng-submit` gets triggered on all those events, whereas the `ng-click` will only be triggered when the user clicks the button.
- Instead of binding to `ctrl.username` and `ctrl.password`, we bind to `ctrl.user.username` and `ctrl.user.password`. Notice that we did not declare a user object in the controller (that is, `self.user = {}`). When you use `ng-model`, AngularJS automatically creates the objects and keys necessary in the chain to instantiate a data-binding connection. In this case, until the user types something into the username or password field, there is no user object. The first letter typed into either the username or password field causes the user object to be created, and the value to be assigned to the correct field in it.

Leverage Data-Binding and Models

When designing your forms and deciding which fields to bind the `ng-model` to, you should always consider what format you need the data in. Let's take the following example to demonstrate:

```
<!-- File: chapter4/two-forms-databinding.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <form ng-submit="ctrl.submit1()">
    <input type="text" ng-model="ctrl.username">
    <input type="password" ng-model="ctrl.password">
    <input type="submit" value="Submit">
  </form>

  <form ng-submit="ctrl.submit2()">
    <input type="text" ng-model="ctrl.user.username">
    <input type="password" ng-model="ctrl.user.password">
    <input type="submit" value="Submit">
  </form>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.submit1 = function() {
        // Create user object to send to the server
        var user = {username: self.username, password: self.password};
        console.log('First form submit with ', user);
      };
      self.submit2 = function() {
        console.log('Second form submit with ', self.user);
      };
    }]);
</script>
</body>
</html>
```

There are two forms in this example, both with the same fields. The first form is bound to a `username` and `password` directly on the controller, while the second form is bound to a `username` and `password` key on a `user` object in the controller. Both of them trigger an `ng-submit` function on submission of a function. Now in the case of the first form, we have to take those fields from the controller and put them into an object, or something similar, before we can send it to the server. In the second case, we can directly take the `user` object from the controller and pass it around.

The second flow makes more sense, because we are directly modeling how we want to represent the form as an object in the controller. This removes any additional work we might have to do when we work with the values of the form.

Form Validation and States

We have seen how to create forms, and enable (and leverage) data-binding to get our data in and out of the UI. Now let's proceed to see how else AngularJS can benefit us when working with forms, and especially with validation and various states of the forms and inputs:

```
<!-- File: chapter4/form-validation.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <form ng-submit="ctrl.submit()" name="myForm">
    <input type="text"
           ng-model="ctrl.user.username"
           required
           ng-minlength="4">
    <input type="password"
           ng-model="ctrl.user.password"
           required>
    <input type="submit"
           value="Submit"
           ng-disabled="myForm.$invalid">
  </form>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.submit = function() {
        console.log('User clicked submit with ', self.user);
      };
    }]);
</script>
</body>
</html>
```

In this example, we reworked our old example to add some validation. In particular, we want to disable the Submit button if the user has not filled out all the required fields. How do we accomplish this?

1. We give the form a name, which we can refer to later. In this case, it is `myForm`.
2. We leverage HTML5 validation tags and add the `required` attribute on each input field.
3. We add a validator, `ng-minlength`, which enforces that the minimum length of the value in the input field for the username is four characters.

4. On the Submit button, we add an `ng-disabled` directive. This disables the element if the condition is true.
5. For the `disable` condition, we leverage the form, which exposes a controller with the current state of the form. In this case, we tell the button to disable itself if the form with the name `myForm` is `$invalid`.

When you use forms (and give them names), AngularJS creates a `FormController` that holds the current state of the form as well as some helper methods. You can access the `FormController` for a form using the form's name, as we did in the preceding example using `myForm`. Things that are exposed as the state and kept up to date with data-binding are shown in Table 4-1.

Table 4-1. Form states in AngularJS

Form state	Description
`$invalid`	AngularJS sets this state when any of the validations (`required`, `ng-minlength`, and others) mark any of the fields within the form as invalid.
`$valid`	The inverse of the previous state, which states that all the validations in the form are currently evaluating to correct.
`$pristine`	All forms in AngularJS start with this state. This allows you to figure out if a user has started typing in and modifying any of the form elements. Possible usage: disabling the reset button if a form is pristine.
`$dirty`	The inverse of `$pristine`, which states that the user made some changes (he can revert it, but the `$dirty` bit is set).
`$error`	This field on the form houses all the individual fields and the errors on each form element. We will talk more about this in the following section.

Each of the states mentioned in the table (except `$error`) are Booleans and can be used to conditionally hide, show, disable, or enable HTML elements in the UI. As the user types or modifies the form, the values are updated as long as you are leveraging `ng-model` and the form name.

Error Handling with Forms

We looked at the types of validation you can do at a form level, but what about individual fields? In our previous example, we ensured that both input fields were required fields, and that the minimum length on the username was four. What else can we do? Table 4-2 contains some built-in validations that AngularJS offers.

Table 4-2. Built-in AngularJS validators

Validator	Description
`required`	As previously discussed, this ensures that the field is required, and the field is marked invalid until it is filled out.
`ng-required`	Unlike `required`, which marks a field as always required, the `ng-required` directive allows us to conditionally mark an input field as required based on a Boolean condition in the controller.
`ng-minlength`	We can set the minimum length of the value in the input field with this directive.
`ng-maxlength`	We can set the maximum length of the value in the input field with this directive.
`ng-pattern`	The validity of an input field can be checked against the regular expression pattern specified as part of this directive.
`type="email"`	Text input with built-in email validation.
`type="number"`	Text input with number validation. Can also have additional attributes for min and max values of the number itself.
`type="date"`	If the browser supports it, shows an HTML datepicker. Otherwise, defaults to a text input. The `ng-model` that this binds to will be a `date` object. This expects the date to be in yyyy-mm-dd format (e.g., 2009-10-24).
`type="url"`	Text input with URL validation.

In addition to this, we can write our own validators, which we cover in Chapter 13.

Displaying Error Messages

What can we do with all these validators? We can of course check the validity of the form, and disable the Save or Update button accordingly. But we also want to tell the user what went wrong and how to fix it. AngularJS offers two things to solve this problem:

- A model that reflects what exactly is wrong in the form, which we can use to display nicer error messages
- CSS classes automatically added and removed from each of these fields allow us to highlight problems in the form

Let's first take a look at how to display specific error messages based on the problem with the following example:

```
<!-- File: chapter4/form-error-messages.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

  <form ng-submit="ctrl.submit()" name="myForm">
    <input type="text"
           name="uname"
           ng-model="ctrl.user.username"
```

```
            required
            ng-minlength="4">
      <span ng-show="myForm.uname.$error.required">
        This is a required field
      </span>
      <span ng-show="myForm.uname.$error.minlength">
        Minimum length required is 4
      </span>
      <span ng-show="myForm.uname.$invalid">
        This field is invalid
      </span>
      <input type="password"
            name="pwd"
            ng-model="ctrl.user.password"
            required>
      <span ng-show="myForm.pwd.$error.required">
        This is a required field
      </span>
      <input type="submit"
            value="Submit"
            ng-disabled="myForm.$invalid">
    </form>

  <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
  </script>
  <script type="text/javascript">
    angular.module('notesApp', [])
      .controller('MainCtrl', [function () {
        var self = this;
        self.submit = function () {
          console.log('User clicked submit with ', self.user);
        };
      }]);
  </script>
  </body>
  </html>
```

Nothing in the controller has changed in this example. Instead, we can just focus on the form HTML. Let's see what changed with the form:

1. First, we added the `name` attribute to both the input fields where we needed validation: `uname` for the username box, and `pwd` for the password text field.
2. Then we leverage AngularJS's form bindings to be able to pick out the errors for each individual field. When we add a name to any input, it creates a model on the form for that particular input, with the error state.
3. So for the username field, we can access it if the field was not entered by accessing `myForm.uname.$error.required`. Similarly, for `ng-minlength`, the field would be

`myForm.uname.$error.minlength`. For the password, we look at `myForm.pwd.$error.required` to see if the field was filled out or not.

4. We also accessed the state of the input, similar to the form, by accessing `myForm.uname.$invalid`. All the other form states (`$valid`, `$pristine`, `$dirty`) we saw earlier are also available similarly on `myForm.uname`.

With this, we now have an error message that shows only when a certain type of error is triggered. Each of the validators we saw in Table 4-2 exposes a key on the `$error` object, so that we can pick it up and display the error message for that particular error to the user. Need to show the user that a field is required? Then when the user starts typing, show the minimum length, and then finally show a message when he exceeds the maximum length. All these kinds of conditional messages can be shown with the AngularJS validators.

Styling Forms and States

We saw the various states of the forms (and the inputs): `$dirty`, `$valid`, and so on. We saw how to display specific error messages and disable buttons based on these conditions, but what if we want to highlight certain input fields or form states using UI and CSS? One option would be to use the form and input states along with the `ng-class` directive to, say, add a class `dirty` when `myForm.$dirty` is true. But AngularJS provides an easier option.

For each of the states we described previously, AngularJS adds and removes the CSS classes shown in Table 4-3 to and from the forms and input elements.

Table 4-3. Form state CSS classes

Form state	CSS class applied
`$invalid`	`ng-invalid`
`$valid`	`ng-valid`
`$pristine`	`ng-pristine`
`$dirty`	`ng-dirty`

Similarly, for each of the validators that we add on the input fields, we also get a CSS class in a similarly named fashion, as demonstrated in Table 4-4.

Table 4-4. Input state CSS classes

Input state	CSS class applied
`$invalid`	`ng-invalid`
`$valid`	`ng-valid`
`$pristine`	`ng-pristine`
`$dirty`	`ng-dirty`

Input state	CSS class applied
required	ng-valid-required or ng-invalid-required
min	ng-valid-min or ng-invalid-min
max	ng-valid-max or ng-invalid-max
minlength	ng-valid-minlength or ng-invalid-minlength
maxlength	ng-valid-maxlength or ng-invalid-maxlength
pattern	ng-valid-pattern or ng-invalid-pattern
url	ng-valid-url or ng-invalid-url
email	ng-valid-email or ng-invalid-email
date	ng-valid-date or ng-invalid-date
number	ng-valid-number or ng-invalid-number

Other than the basic input states, AngularJS takes the name of the validator (number, maxlength, pattern, etc.) and depending on whether or not that particular validator has been satisfied, adds the `ng-valid-validator_name` or `ng-invalid-validator_name` class, respectively.

Let's take an example of how this might be used to highlight the input in different ways:

```
<!-- File: chapter4/form-styling.html -->
<html ng-app="notesApp">
<head>
  <title>Notes App</title>
  <style>
    .username.ng-valid {
      background-color: green;
    }
    .username.ng-dirty.ng-invalid-required {
      background-color: red;
    }
    .username.ng-dirty.ng-invalid-minlength {
      background-color: lightpink;
    }
  </style>
</head>
<body ng-controller="MainCtrl as ctrl">

  <form ng-submit="ctrl.submit()" name="myForm">
    <input type="text"
           class="username"
           name="uname"
           ng-model="ctrl.user.username"
           required
           ng-minlength="4">
    <input type="submit"
           value="Submit"
           ng-disabled="myForm.$invalid">
```

```
  </form>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
      self.submit = function() {
        console.log('User clicked submit with ', self.user);
      };
    }]);
</script>
</body>
</html>
```

In this example, we kept the existing functionality of the validators, though we removed the specific error messages. Instead, what we try to do is mark out the required field using CSS classes. So here is what the example accomplishes:

- When the field is correctly filled out, it turns the input box green. This is done by setting the background color when the CSS class `ng-valid` is applied to our input field.
- We want to display the background as dark red if the user starts typing in, and then undoes it. That is, we want to set the background as red, marking it as a required field, but only after the user modifies the field. So we set the background color to be red if the CSS classes `ng-dirty` (which marks that the user has modified it) and `ng-invalid-minlength` (which marks that the user has not typed in the necessary amount of characters) are applied.

Similarly, you could add a CSS class that shows a * mark in red if the field is required but not dirty. Using a combination of these CSS classes (and the form and input states) from before, you can easily style and display all the relevant and actionable things to the user about your form.

Nested Forms with ng-form

By this point, we know how to create forms and get data into and out of our controllers (by binding it to a model). We have also seen how to perform simple validation, and style and display conditional error messages in AngularJS.

The next part that we want to cover is how to deal with more complicated form structures, and grouping of elements. We sometimes run into cases where we need subsections of our form to be valid as a group, and to check and ascertain its validity. This is not possible with the HTML `form` tag because `form` tags are not meant to be nested.

AngularJS provides an `ng-form` directive, which acts similar to `form` but allows nesting, so that we can accomplish the requirement of grouping related form fields under sections:

```
<!-- File: chapter4/nested-forms.html -->
<html ng-app>
<head>
  <title>Notes App</title>
</head>

<body>
  <form novalidate name="myForm">
    <input type="text"
           class="username"
           name="uname"
           ng-model="ctrl.user.username"
           required=""
           placeholder="Username"
           ng-minlength="4" />
    <input type="password"
           class="password"
           name="pwd"
           ng-model="ctrl.user.password"
           placeholder="Password"
           required="" />


    <ng-form name="profile">
      <input type="text"
             name="firstName"
             ng-model="ctrl.user.profile.firstName"
             placeholder="First Name"
             required>
      <input type="text"
             name="middleName"
             placeholder="Middle Name"
             ng-model="ctrl.user.profile.middleName">
      <input type="text"
             name="lastName"
             placeholder="Last Name"
             ng-model="ctrl.user.profile.lastName"
             required>
      <input type="date"
             name="dob"
             placeholder="Date Of Birth"
             ng-model="ctrl.user.profile.dob">
    </ng-form>

    <span ng-show="myForm.profile.$invalid">
        Please fill out the profile information
      </span>
```

```
    <input type="submit"
           value="Submit"
           ng-disabled="myForm.$invalid"/>
  </form>

<script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
</body>

</html>
```

In this example, we nest a subform inside our main form, but because the HTML form element cannot be nested, we use the `ng-form` directive to do it. Now we can have substate within our form, evaluate quickly if each section is valid, and leverage the same binding and form states that we have looked at so far. A quick highlight of the features in the example:

- A subform using the `ng-form` directive. We can give this a name to identify and grab the state of the subform.
- The state of the subform can be accessed directly (`profile.$invalid`) or through the parent form (`myForm.profile.$invalid`).
- Individual elements of the form can be accessed as normal (`profile.firstName.$error.required`).
- Subforms and nested forms still affect the outer form (the `myForm.$invalid` is `true` because of the use of the required tags).

You could have subforms and groupings that have their own way of checking and deciding validity, and `ng-form` allows you to model that grouping in your HTML.

Other Form Controls

We have dealt with forms, `ng-model`s, and bindings, but mostly we have only looked at regular text boxes. Let's see how to interact and work with other form elements in AngularJS.

Textareas

Textareas in AngularJS work exactly the same as text inputs. That is, to have two-way data-binding with a textarea, and make it a required field, you would do something like:

```
<textarea ng-model="ctrl.user.address" required></textarea>
```

All the data-binding, error states, and CSS classes remain as we saw it with text inputs.

Checkboxes

Checkboxes are in some ways easier to deal with because they can only have one of two values: true or false. So an `ng-model` two-way data-binding to the checkbox basically takes a Boolean value and assigns the checked state based on it. After that, any changes to the checkbox toggles the state of the model:

```
<input type="checkbox" ng-model="ctrl.user.agree">
```

But what if we didn't have just Boolean values? What if we wanted to assign the string YES or NO to our model, or have the checkbox checked when the value is YES? AngularJS gives two attribute arguments to the checkbox that allow us to specify our custom values for the true and false values. We could accomplish this as follows:

```
<input type="checkbox"
           ng-model="ctrl.user.agree"
           ng-true-value="YES"
           ng-false-value="NO">
```

This sets the value of the `agree` field to YES if the user checks the checkbox, and NO if the user unchecks it.

But what if we didn't want the two-way data-binding, and just want to use the checkbox to display the current value of a Boolean? That is, one-way data-binding where the state of the checkbox changes when the value behind it changes, but the value doesn't change on checking or unchecking the checkbox.

We can accomplish this using the `ng-checked` directive, which binds to an AngularJS expression. Whenever the value is true, AngularJS will set the checked property for the input element, and remove and unset it when the value is false. Let's use the following example to demonstrate all these together:

```
<!-- File: chapter4/checkbox-example.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">
  <div>
    <h2>What are your favorite sports?</h2>
    <div ng-repeat="sport in ctrl.sports">
      <label ng-bind="sport.label"></label>
      <div>
        With Binding:
        <input type="checkbox"
               ng-model="sport.selected"
               ng-true-value="YES"
               ng-false-value="NO">
      </div>
      <div>
        Using ng-checked:
        <input type="checkbox"
               ng-checked="sport.selected === 'YES'">
```

```
      </div>
      <div>
        Current state: {{sport.selected}}
      </div>
    </div>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      var self = this;
       self.sports = [
         {label: 'Basketball', selected: 'YES'},
         {label: 'Cricket', selected: 'NO'},
         {label: 'Soccer', selected: 'NO'},
         {label: 'Swimming', selected: 'YES'}
       ];
    }]);
</script>
</body>
</html>
```

With this example, we have an `ng-repeat`, which has a checkbox with `ng-model`, a checkbox with `ng-checked`, and a div with the current state bound to it. The first checkbox uses the traditional two-way data-binding with the `ng-model` directive. The second checkbox uses `ng-checked`. This means that:

- When the user checks the first checkbox, the value of `selected` becomes `YES` because the true value, set using `ng-true-value`, is `YES`. This triggers the `ng-checked` and sets the second box as checked (or unchecked).
- When the user unchecks the first box, the value of `selected` is set to `NO` because of the `ng-false-value`.
- The second checkbox in each repeater element displays the state of the `ng-model` using `ng-checked`. This updates the state of the checkbox whenever the model backing `ng-model` changes. Checking or unchecking the second checkbox itself has no effect on the value of the model.

So if you need two-way data-binding, use `ng-model`. If you need one-way data-binding with checkboxes, use `ng-checked`.

Radio Buttons

Radio buttons behave similarly to checkboxes, but are slightly different. You can have multiple radio buttons (and you normally do) that each assigns a different value to a

model depending on which one is selected. You can specify the value using the traditional value attribute of the input element. Let's see how that would look:

```
<div ng-init="user = {gender: 'female'}">
    <input type="radio"
           name="gender"
           ng-model="user.gender"
           value="male">
    <input type="radio"
           name="gender"
           ng-model="user.gender"
           value="female">
</div>
```

In this example, we have two radio buttons. We gave them both the same name so that when one is selected, the other gets deselected. Both of them are bound to the same `ng-model` (`user.gender`). Next, each of them has a value, which is the value that gets stored in `user.gender` (male if it is the first radio button; female, otherwise). Finally, we have an `ng-init` block surrounding it, which sets the value of `user.gender` to be female by default. This has the effect of ensuring that the second checkbox is selected when this snippet of HTML loads.

But what if our values are dynamic? What if the value we needed to assign was decided in our controller, or some other place? For that, AngularJS gives you the `ng-value` attribute, which you can use along with the radio buttons. `ng-value` takes an AngularJS expression, and the return value of the expression becomes the value that is assigned to the model:

```
<div ng-init="otherGender = 'other'">
    <input type="radio"
           name="gender"
           ng-model="user.gender"
           value="male">Male
    <input type="radio"
           name="gender"
           ng-model="user.gender"
           value="female">Female
    <input type="radio"
           name="gender"
           ng-model="user.gender"
           ng-value="otherGender">{{otherGender}}
</div>
```

In this example, the third option box takes a dynamic value. In this case, we assign it as part of the initialization block (`ng-init`), but in a real application, the initialization could be done from within a controller instead of in the HTML directly. When we say `ng-value="otherGender"`, it doesn't assign `otherGender` as a string to `user.gender`, but the value of the `otherGender` variable, which is *`other`*.

Combo Boxes/Drop-Downs

The final HTML form element (which can be used outside forms as well) is the select box, or the drop-down/combo box as it is commonly known. Let's take a look at the simplest way you can use select boxes in AngularJS:

```
<div ng-init="location = 'India'">
    <select ng-model="location">
      <option value="USA">USA</option>
      <option value="India">India</option>
      <option value="Other">None of the above</option>
    </select>
</div>
```

In this example, we have a simple select box that is data-bound to the variable `location`. We also initialize the value of `location` to India, so when the HTML loads, India is the selected option. When the user selects any of the other options, the value of the value attribute gets assigned to the `ng-model`. The standard validators and states also apply to this field, so those can be applied (`required`, etc.).

This has a few restrictions, though:

- You need to know the values in the drop-down up front.
- They need to be hardcoded.
- The values can only be strings.

In a truly dynamic app, one or none of these might be true. In such a case, the `select` HTML element also has a way of dynamically generating the list of options, and working with objects instead of pure string `ng-models`. This is done through the `ng-options` directive. Let's take a look at how this might work:

```
<!-- File: chapter4/select-example.html -->
<html ng-app="notesApp">
<head><title>Notes App</title></head>
<body ng-controller="MainCtrl as ctrl">

<div>
  <select ng-model="ctrl.selectedCountryId"
          ng-options="c.id as c.label for c in ctrl.countries">
  </select>
  Selected Country ID : {{ctrl.selectedCountryId}}
</div>

<div>
  <select ng-model="ctrl.selectedCountry"
          ng-options="c.label for c in ctrl.countries">
  </select>

  Selected Country : {{ctrl.selectedCountry}}
```

```
</div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', [function() {
      this.countries = [
        {label: 'USA', id: 1},
        {label: 'India', id: 2},
        {label: 'Other', id: 3}
      ];
      this.selectedCountryId = 2;
      this.selectedCountry = this.countries[1];
    }]);
</script>
</body>
</html>
```

In this example, we have two select boxes, both bound to different models in our controller. The first `select` element is bound to `ctrl.selectedCountryId` and the second one is bound to `ctrl.selectedCountry`. Note that one is a number, while the other is an actual object. How do we achieve this?

- We use the `ng-options` attribute on the select dialog, which allows us to repeat an array (or object, similar to `ng-repeat` from "Working with and Displaying Arrays" on page 22) and display dynamic options.
- The syntax is similar to `ng-repeat` as well, with some additional ability to select what is displayed as the label, and what is bound to the model.
- In the first select box, we have `ng-options="c.id as c.label for c in ctrl.countries"`. This tells AngularJS to create one option for each country in the array of countries. The syntax is as follows: `modelValue as labelValue for item in array`. In this case, we tell AngularJS that our `modelValue` is the ID of each element, the label value is the label key of each array item, and then our typical `for each` loop.
- In the second select box, we have `ng-options="c.label for c in ctrl.countries"`. Here, when we omit the `modelValue`, AngularJS assumes that each item in the repeat is the actual model value, so when we select an item from the second select box, the country object (c) of that option box gets assigned to `ctrl.selectedCountry`.
- Because the backing model for the two select boxes are different, changing one does not affect the value or the selection in the other drop-down.

- You can also optionally give a grouping clause, for which the syntax would be `ng-options="modelValue as labelValue group by groupValue for item in array"`. Similar to how we specified the model and label values, we can point the `groupValue` at another key in the object (say, continent).
- When you use objects, the clause changes as follows: `modelValue as labelValue group by groupValue for (key, value) in object`.

AngularJS compares the `ng-options` individual values with the `ng-model` by reference. Thus, even if the two are objects that have the same keys and values, AngularJS will not show that item as selected in the drop-down unless and until they are the same object. We accomplished this in our example by using an item from the array `countries` to assign the initial value of the model.

There is a better way to accomplish this, which is through the use of the track by syntax with `ng-options`. We could have written the `ng-options` as:

```
ng-options="c.label for c in ctrl.countries track by c.id"
```

This would ensure that the object `c` is compared using the ID field, instead of by reference, which is the default.

Conclusion

We started with the most common requirements, which is getting data in and out of UI forms. We played around with `ng-model`, which gives us two-way data-binding to remove most of the boilerplate code we would write when working with forms. We then saw how we could leverage form validation, and show and style error messages. Finally, we saw how to deal with other types of form elements, and the kinds of options AngularJS gives to work with them.

In the next chapter, we start dealing with AngularJS services and then jump into server communication using the `$http` service in AngularJS.

CHAPTER 5

All About AngularJS Services

Until now, we have dealt with data-binding in AngularJS. We have seen how to take data from our controllers and get it into the UI, and ensure that whenever the user interacts with or types in any data, we get it back into our controllers. We used and worked with some common directives, and dealt with forms and error handling.

In this chapter, we dive into AngularJS services. By the end of the chapter, we will have a thorough understanding of AngularJS services and get some hands-on experience in using core built-in AngularJS services. After that, we will learn why and when we should create AngularJS services, and actually create a simple service ourselves.

AngularJS Services

AngularJS services are functions or objects that can hold behavior or state across our application. Each AngularJS service is instantiated only once, so each part of our application gets access to the same instance of the AngularJS service. Repeated behavior, shared state, caches, factories, etc. are all functionality that can be implemented using AngularJS services.

Service Versus Service

In AngularJS, when we say service, we are actually referring to the conceptual service that is a reusable API or substitutable objects, which can be shared across our applications. A service in AngularJS can be implemented as a factory, service, or provider.

This is one of the badly named concepts in AngularJS and thus can lead to confusion. We end up calling all of the above services. We will see the difference between them in a bit.

Let's first take a look at why we need them.

Why Do We Need AngularJS Services?

So far we have only created AngularJS controllers, which create state and functions that our HTML then uses for a variety of tasks. AngularJS controllers are great for tasks that relate to the following:

- Which model and data fields to fetch and show in the HTML
- User interaction, as in what needs to happen when a user clicks something
- Presentation logic, such as how a particular UI element should be styled, or whether it should be hidden

Controllers are stateful, but ephemeral. That is, they can be destroyed and re-created multiple times throughout the course of navigating across a Single Page Application. Let's take a look at an example to clarify this:

```
<!-- File: chapter5/need-for-service/index.html -->
<html ng-app="notesApp">

<head>
  <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
  </script>
  <script src="app.js"></script>
</head>

<body ng-controller="MainCtrl as mainCtrl">
  <h1>Hello Controllers!</h1>
  <button ng-click="mainCtrl.open('first')">Open First</button>
  <button ng-click="mainCtrl.open('second')">Open Second</button>
  <div ng-switch on="mainCtrl.tab">

    <div ng-switch-when="first">
      <div ng-controller="SubCtrl as ctrl">
        <h3>First tab</h3>
        <ul>
          <li ng-repeat="item in ctrl.list">
            <span ng-bind="item.label"></span>
          </li>
        </ul>

        <button ng-click="ctrl.add()">Add More Items</button>
      </div>

    </div>
    <div ng-switch-when="second">
      <div ng-controller="SubCtrl as ctrl">
        <h3>Second tab</h3>
        <ul>
          <li ng-repeat="item in ctrl.list">
            <span ng-bind="item.label"></span>
```

```
          </li>
        </ul>

        <button ng-click="ctrl.add()">Add More Items</button>
      </div>
    </div>
  </div>
</body>

</html>
```

```
// File: chapter5/need-for-service/app.js
angular.module('notesApp', [])
  .controller('MainCtrl', [function() {
    var self = this;
    self.tab = 'first';
    self.open = function(tab) {
      self.tab = tab;
    };
  }])
  .controller('SubCtrl', [function() {
    var self = this;
    self.list = [
      {id: 1, label: 'Item 0'},
      {id: 2, label: 'Item 1'}
    ];

    self.add = function() {
      self.list.push({
        id: self.list.length + 1,
        label: 'Item ' + self.list.length
      });
    };
  }]);
```

In this example, we introduced two controllers for the first time: a `MainCtrl` and a `SubCtrl`. The `MainCtrl` controls the overall page, and the `SubCtrl` controls a subsection of the page and holds the data we want to display.

We also have two tabs, which are shown and hidden depending on which button the user clicks. This is accomplished using a new directive, `ng-switch`. `ng-switch` acts like a `switch` statement in the HTML. It takes a variable (using the `on` attribute, which in this case is `MainCtrl`'s tab), and then, depending on the state, hides and shows elements (using the `ng-switch-when` attribute, used as children of the `ng-switch`). The `ng-switch-when` takes the value that the variable should take.

Finally, the `SubCtrl` has a function to add more items to the array, which is triggered by a button in the UI.

Now, moving on to our HTML, the body is controlled by the `MainCtrl`, and holds state on which tab in the HTML is shown. We then have two buttons that allow us to change

which tab is currently shown. Notice again that this is done by changing the model and letting AngularJS update the UI automatically.

Finally, we have a `div` element on which we have the `ng-switch`. Both tabs (each one has the `ng-switch-when`) are exactly the same except for the header. They show the list of items, and add items when the button in that tab is clicked.

With this out of the way, let's take a look at some key behaviors:

- Both of the tabs, First and Second, are using the same controller, `SubCtrl`. But each one has its own instance of the `list` variable. Adding items in one tab does not add them to the other, and vice versa.
- If we add items to the first tab and then switch to the second tab, we will see the items the controller starts with. But then if we navigate back to the first tab, we will see that those items disappear from the first controller as well.

We could still achieve the functionality we were aiming for by having a parent-level controller, and moving our `list` variable into the parent controller (such as `MainCtrl`). Each `SubCtrl` would then have to access the variable through the top controller explicitly. This solves our problem but adds global, implicit state, which is never ideal for a large, maintainable application.

When we use controllers, they are instances that get created and destroyed as we navigate across our application. This is especially true when we start working on routing and multiple URLs in a Single Page Application in Chapter 6. Also, one controller cannot directly communicate with another controller to share state or behavior.

Services Versus Controllers

In the applications we develop, we will end up using both controllers as well as services. Now, when we say "services" in AngularJS, we include factories, services, and providers. We'll see the difference between the three in "The Difference Between Factory, Service, and Provider" on page 82.

That said, both controllers and services fill a certain need in our application, and attempting to do too much or do in one what ideally belongs in the other can lead to bloated, unmaintainable, and untestable code. Table 5-1 gives a quick overview of the types of responsibilites and needs for which we would use controllers versus services.

Table 5-1. Controllers versus services

Controllers	Services
Presentation logic	Business logic
Directly linked to a view	Independent of views
Drives the UI	Drives the application
One-off, specific	Reusable
Responsible for decisions like what data to fetch, what data to show, how to handle user interactions, and styling and display of UI	Responsible for making server calls, common validation logic, application-level stores, and reusable business logic

We'll dive into AngularJS services next, including how to use existing services and create our own. After we finish that, we'll come back to some examples of what belongs in services.

Dependency Injection in AngularJS

The entire service concept in AngularJS is heavily dependent on and driven by its Dependency Injection system. Any service known to AngularJS (internal or our own) can be simply injected into any other service, directive, or controller by stating it as a dependency. AngularJS will automatically figure out what the service is, what it further depends on, and create the entire chain before injecting a fully instantiated service.

Dependency Injection is a concept that started more on the server side, to basically propagate reuse, modularity, and testability of code. Dependency Injection states that instead of creating an instance of a dependent service when we need it, the class or function should ask for it instead. Something else (usually known as an injector) would then be responsible for figuring out how to create it and pass it in.

Consider a case where we had a service called `$http` that could make server calls. Now let's take two cases, with and without Dependency Injection:

```
// Without Dependency Injection
function fetchDashboardData() {
  var $http  = new HttpService();
  return $http.get('my/url');
}

// With Dependency Injection
function fetchDashboardData($http) {
  return $http.get('my/url');
}
```

In the first function, a new instance of `$http` is created whenever a server call needs to happen. In the second, the `$http` service instance itself gets passed in.

What are the disadvantages of the former?

- Because it creates a new instance using the `new` keyword, any test we write for this function is dependent on `HttpService` implicitly.
- If we need to extend `HttpService` to provide for offline functionality, or change it to sockets, we will be forced to change each implementation where `new` is called.
- It is inherently tied to `HttpService`, making it hard to reuse for other cases, such as with sockets or offline as mentioned previously.

Dependency Injection allows us to:

- Change the underlying implemention of a dependency without manually changing each dependent function
- Change the underlying implementation just for the test, to prevent it from making server calls
- Explicitly state what needs to be included and present before this function or constructor can execute

AngularJS guarantees that the function we provide to the service declaration will be executed only once (*lazily*, the first time something that needs the dependency is loaded), and future dependents will get that very same instance. That is, AngularJS services are singletons for the scope of our application. Two controllers or services that ask for ServiceA will get the very same instance, instead of two different instances.

Using Built-In AngularJS Services

Before we go off and try to create our own services, let's take a look at some existing core AngularJS services and how we might use them in our own applications. The simplest one we can start working with is the `$log` service.

The $ Prefix in AngularJS

AngularJS prefixes all the services that are provided by the AngularJS library with the `$` sign. So you will see services like `$log`, `$http`, `$window`, and so on. This is used as a namespacing technique so that when you see a service, you can immediately figure out whether a service is coming from AngularJS or somewhere else. Conversely, when you create your own services, do not prefix them with a `$` sign. It will just end up confusing you and your team at some point in time.

Before we can use any AngularJS service (in a controller, service, or otherwise), we need to inject it in. Let's see how we can write a very simple controller that pulls in the `$log` service:

```
<!-- File: chapter5/log-example.html -->
<html ng-app="notesApp">
<body ng-controller="MainCtrl as mainCtrl">
  <h1>Hello Services!</h1>
  <button ng-click="mainCtrl.logStuff()">Log something</button>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script type="text/javascript">
  angular.module('notesApp', [])
    .controller('MainCtrl', ['$log', function($log) {
      var self = this;
      self.logStuff = function() {
        $log.log('The button was pressed');
      };
    }])
</script>
</body>
</html>
```

In this example, the HTML has been simplified down to a single button, which triggers `MainCtrl.logStuff()`. It uses the `ng-click` directive that we've seen before.

Our first major change is in the controller definition. So far, we have had the name of the controller as the first argument to the `controller()` function, and an array with the controller definition inside it. Now when we depend on a service, we first add the dependency as a string in the array (this is what we call the safe style of Dependency Injection). After we declare it as a string, we then inject it as a variable (the name of our choosing) into the function that is passed as the last argument in the array.

AngularJS will pick up the individual strings in the array, look up the services internally, and inject them into the function in the order in which we have defined the strings.

As soon as we have the service, we can use it as the API permits within our controller, so our `logStuff` function just logs a string to the console.

Safe Style of Dependency Injection

In the example, we defined our dependencies as follows:

```
myModule.controller("MainCtrl", ["$log", function($log) {}]);
```

We could have also defined them as:

```
myModule.controller("MainCtrl", function($log) {});
```

That is, ditch the array syntax and directly provide our controller function. Why then would we go to this extra effort of typing in boilerplate if it doesn't have any effect?

The reason for preferring the first syntax over the latter is that when we build our application for deployment, we often run our JavaScript through a step known as minification or uglification. In this step, our JavaScript is globbed into one single file, comments are dropped, spaces are removed, and finally, variables are renamed to make them shorter. So the `$log` variable might get renamed to `xz` (or some other random, shorter name).

When we normally run our application and use the latter syntax (without the arrays), AngularJS is able to look at the name of the variable and figure out what service we need. When the uglification has finished, AngularJS has no clue what the variable `xz` previously referred to.

The uglification and minification processes do not touch string constants. Therefore, the first example would get translated to something like:

```
myModule.controller("MainCtrl", ["$log", function(xz) {}]);
```

while the latter example would translate to something like:

```
myModule.controller("MainCtrl", function(xz) {});
```

In the former, AngularJS still has the string `"$log"` to tell it what the service originally was, while it doesn't have that in the latter.

Recent developments like the `ng-min` library allow us to write code in the latter way and have it automatically convert to the former, but it can have edge cases. So it might be preferable to always use the safer style of Dependency Injection in case you don't want to risk it.

In this book, we will always use the safer style of Dependency Injection.

Order of Injection

We define our dependencies as strings. AngularJS inspects the strings, and injects the dependencies in the order in which they are listed:

```
myModule.controller("MainCtrl",
    ["$log", "$window", function($l, $w) {}]);
```

In this line of code, the `$log` service would be injected into the `$l` variable in the function, and the `$window` service would get injected into the `$w` variable:

```
myModule.controller("MainCtrl",
    ["$log", "$window", function($w, $l) {}]);
```

In this line of code, it is almost the exact same thing, except the `$w` and `$l` variables have been switched inside the function. AngularJS will ignore this and take its cue from the strings. So the `$w` variable will actually hold the `$log` service, and the `$l` variable would in fact hold the `$window` service.

So just be careful to keep the strings and the variables in sync and in the same order, or expect some craziness with your code.

Common AngularJS Services

Some other AngularJS services that we will see or use on a common basis are:

`$window`
: The `$window` service in AngularJS is nothing but a wrapper around the global window object. The sole reason for its existence is to avoid global state, especially in tests. Instead of directly working with the window object, we can ask for and work with `$window`. In the unit tests, the `$window` service can be easily mocked out (available for free with the AngularJS mocking library).

`$location`
: The `$location` service in AngularJS allows us to interact with the URL in the browser bar, and get and manipulate its value. Any changes made to the `$location` service get reflected in the browser, and any changes in the browser are immediately captured in the `$location` service. The `$location` service has the following functions, which allow us to work with the URL:

  `absUrl`
  : A getter that gives us the absolute URL in the browser (called `$location.absUrl()`).

  `url`
  : A getter and setter that gets or sets the URL. If we give it an argument, it will set the URL; otherwise, it will return the URL as a string.

  `path`
  : Again, a getter and setter that sets the path of the URL. Automatically adds the forward slash at the beginning. So `$location.path()` would give us the current path of the application, and `$location.path("/new")` would set the path to `/new`.

  `search`
  : Sets or gets the search or query string of the current URL. Calling `$location.search()` without any arguments returns the search parameter as an object. Calling `$location.search("test")` removes the search parameter from the URL, and calling `$location.search("test", "abc");` sets the search parameter test to `abc`.

`$http`
: We will deal with `$http` extensively in Chapter 6, but it is the core AngularJS service used to make XHR requests to the server from the application. Using the `$http`

service, we can make GET and POST requests, set the headers and caching, and deal with server responses and failures.

Creating Our Own AngularJS Service

We saw how to use AngularJS services through the use of some built-in AngularJS services. We will be using the ones previously mentioned extensively throughout the book going forward, so don't worry that we didn't get to see them all in action yet.

The core AngularJS services only touch the tip of the iceberg in terms of the functionality we will need when we start creating our own AngularJS applications. But how do we decide between embedding our functionality right in the controller or putting it in a service? We should consider creating an AngularJS service if what we are implementing falls into one of the following broad criteria:

It needs to be reusable
: More than one controller or service will need to access the particular function that is being implemented.

Application-level state
: Controllers get created and destroyed. If we need state stored across our application, it belongs in a service.

It is independent of the view
: If what we are implementing is not directly linked to a view, it probably belongs in a service.

It integrates with a third-party service
: We need to integrate a third-party service (think SocketIO, BreezeJS, etc.), but we want to be able to mock or replace it in our unit tests. A service makes that easy.

Caching/factories
: Do we need an object cache? Or something that creates model objects? Services are our best bet.

Services themselves can depend on other built-in services or our own services. So traditional software engineering concepts like modularity, composite services, and even hierarchy of services are still applicable.

Creating a Simple AngularJS Service

Let's take an example of how to create a simple service. We will take the very first example from this chapter, which demonstrated the problem with using just controllers, and use a service to share the state between the two views:

```
<!-- File: chapter5/simple-angularjs-service/index.html -->
<html ng-app="notesApp">
```

```
<body ng-controller="MainCtrl as mainCtrl">
  <h1>Hello Controllers!</h1>
  <button ng-click="mainCtrl.open('first')">
    Open First
  </button>
  <button ng-click="mainCtrl.open('second')">
    Open Second
  </button>
  <div ng-switch on="mainCtrl.tab">
    <div ng-switch-when="first">
      <div ng-controller="SubCtrl as ctrl">
        <h3>First tab</h3>
        <ul>
          <li ng-repeat="item in ctrl.list()">
            <span ng-bind="item.label"></span>
          </li>
        </ul>

        <button ng-click="ctrl.add()">
          Add More Items
        </button>
      </div>

    </div>
    <div ng-switch-when="second">
      <div ng-controller="SubCtrl as ctrl">
        <h3>Second tab</h3>
        <ul>
          <li ng-repeat="item in ctrl.list()">
            <span ng-bind="item.label"></span>
          </li>
        </ul>

        <button ng-click="ctrl.add()">
          Add More Items
        </button>
      </div>
    </div>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script src="app.js"></script>
</body>

</html>
```

The *app.js* file, which houses the controllers and services, looks something like this:

```
// File: chapter5/simple-angularjs-service/app.js

angular.module('notesApp', [])
```

```
.controller('MainCtrl', [function() {
  var self = this;
  self.tab = 'first';
  self.open = function(tab) {
    self.tab = tab;
  };
}])
.controller('SubCtrl', ['ItemService',
    function(ItemService) {
  var self = this;
  self.list = function() {
    return ItemService.list();
  };

  self.add = function() {
    ItemService.add({
      id: self.list().length + 1,
      label: 'Item ' + self.list().length
    });
  };
}])
.factory('ItemService', [function() {
  var items = [
    {id: 1, label: 'Item 0'},
    {id: 2, label: 'Item 1'}
  ];
  return {
    list: function() {
      return items;
    },
    add: function(item) {
      items.push(item);
    }
  };
}]);
```

We changed the following things in the previous example:

- Instead of the list being instantiated and stored in the `SubCtrl` (and getting destroyed and re-created), we are storing the list in a service called `ItemService`.
- The `SubCtrl` has a function called `list()`, which just delegates and returns the value of `ItemService.list()` function.
- The `SubCtrl` has a function called `add()` that delegates and adds an item to the `ItemService`.
- The HTML now binds the `ng-repeat` to `ctrl.list()` instead of `ctrl.list`. So it calls the function and uses its return value to display the array in the UI.

We created the `ItemService` using an AngularJS module function called `factory`:

- The `factory` function follows a similar declaration style like the controller. So we declare the name of the service, `ItemService`, in the first argument and then the array syntax for Dependency Injection with our actual service function as the second argument.
- In the service definition function, we return an object, which becomes the API for the service. In this case, the `ItemService` defines two functions, `list` and `add`, which all users of the service can access.
- In the service definition function, we also declare some local variables (in this case, the `items` array). These are private to the service, and cannot be accessed directly (though they are accessible through the `list` function here) by any users of the service. Therefore, no controller can access `ItemService.items` directly.

The `ItemService` gets instantiated once when the application loads and the `SubCtrl` is loaded, at which point AngularJS decides it needs an instance of the `ItemService`. After it is created, all other controllers that ask for the `ItemService` will get the exact same instance that was returned the very first time.

This is why both the tabs in our example show the exact same list, and if we click Add in one tab and then move to the other tab, the items still show up.

To summarize, when we create our own AngularJS service:

- Use the `angular.module().factory` function to declare the service's name and dependencies.
- Return an object, or a function from within the service definition, which becomes the public API for our service.
- Hold internal state as local variables inside the service. This is important because in a Single Page Application where controllers can get created and destroyed, the service can act as an application-level store.

AngularJS guarantees the following:

- The service will be lazily instantiated. The very first time a controller, service, or directive asks for the service, it will be created.
- The service definition function will be called once, and the instance stored. Every caller of this service will get this same, singleton instance handed to them.

This is important because in a Single Page Application, the HTML and controllers can get destroyed and created multiple times in an application.

In this way, we can create our own AngularJS service, and define the API of how someone interacts with our own service. Notice that we call `ItemService` a service, even though we defined it using a function called `factory`. We will touch upon this in the next section.

The Difference Between Factory, Service, and Provider

AngularJS provides a few different ways in which we can to create and register services (and constants and values), depending on our preference and style of programming. In the previous section, we used the `factory` function to define our services.

You should use `module.factory()` to define your services if:

- You follow a functional style of programming
- You prefer to return functions and objects

JavaScript (and AngularJS) also allow us to follow a Class/OO style of programming, where we define classes and types instead of functions and objects. When we use a service, AngularJS assumes that the function definition passed in as part of the array of dependencies is actually a JavaScript type/class. So instead of just invoking the function and storing its return value, AngularJS will call `new` on the function to create an instance of the type/class.

Let's see how the service we defined in the previous example changes if we use the `service()` function:

```
// File: chapter5/item-service-using-service/app.js

function ItemService() {
  var items = [
    {id: 1, label: 'Item 0'},
    {id: 2, label: 'Item 1'}
  ];
  this.list = function() {
    return items;
  };
  this.add = function(item) {
    items.push(item);
  };
}

angular.module('notesApp', [])
  .service('ItemService', [ItemService])
  .controller('MainCtrl', [function() {
    var self = this;
    self.tab = 'first';
    self.open = function(tab) {
      self.tab = tab;
    };
  }])
```

```
.controller('SubCtrl',
    ['ItemService', function(ItemService) {
  var self = this;
  self.list = function() {
    return ItemService.list();
  };

  self.add = function() {
    ItemService.add({
      id: self.list().length + 1,
      label: 'Item ' + self.list().length
    });
  };
}]);
```

In this example, we can use the exact same controllers and HTML from before. We can use this *app.js* with the *index.html* from the previous example. We only display the part that is changed, which is the service definition:

- The first thing of note is that we now use `service` instead of `factory` for defining our AngularJS service.
- Our service definition function is now a JavaScript class function. It doesn't return anything.
- Our service defines the public API by defining methods (`add`, `list`) on its instance (using the `this` keyword).
- Private state for the service is still defined as local variables inside the function definition.
- AngularJS will perform `new ItemService()` (with possible dependencies injected in) and then return that instance to all functions that depend on `ItemService`.

The third and final way of defining services is using the `provider` function. This is not a very common approach, but can be useful when we need to set up some configuration for our service before our application loads. We will deal with application-level configuration in Chapter 6, but with the provider, we can have functions that can be called to set up how our service works based on the language, environment, or other things that are applicable to our service. Let's see how that might look:

```
// File: chapter5/item-service-using-provider/app.js

function ItemService(opt_items) {
  var items = opt_items || [];

  this.list = function() {
    return items;
  };
  this.add = function(item) {
    items.push(item);
```

```
  };
}

angular.module('notesApp', [])
  .provider('ItemService', function() {
    var haveDefaultItems = true;

    this.disableDefaultItems = function() {
      haveDefaultItems = false;
    };

    // This function gets our dependencies, not the
    // provider above
    this.$get = [function() {
      var optItems = [];
      if (haveDefaultItems) {
        optItems = [
          {id: 1, label: 'Item 0'},
          {id: 2, label: 'Item 1'}
        ];
      }
      return new ItemService(optItems);

    }];
  })
  .config(['ItemServiceProvider',
    function(ItemServiceProvider) {
      // To see how the provider can change
      // configuration, change the value of
      // shouldHaveDefaults to true and try
      // running the example
      var shouldHaveDefaults = false;

      // Get configuration from server
      // Set shouldHaveDefaults somehow
      // Assume it magically changes for now
      if (!shouldHaveDefaults) {
        ItemServiceProvider.disableDefaultItems();
      }
  }])
  .controller('MainCtrl', [function() {
    var self = this;
    self.tab = 'first';
    self.open = function(tab) {
      self.tab = tab;
    };
  }])
  .controller('SubCtrl',
      ['ItemService', function(ItemService) {
    var self = this;
    self.list = function() {
      return ItemService.list();
```

```
  };

  self.add = function() {
    ItemService.add({
      id: self.list().length + 1,
      label: 'Item ' + self.list().length
    });
  };
}]);
```

We introduced two new concepts in this example. The rest of the controllers and HTML remain the same as before, so use the same *index.html* from the factory example:

- `ItemService` now takes in the list of default items as an argument to the constructor.
- `ItemService` is declared using a provider. We define a function in the provider called `disableDefaultItems`. This can be called in the configuration phase of an AngularJS application. That is, this can be called before the AngularJS app has loaded and the service has been initialized.
- Note that the provider does not use the same notation as factory and service. It doesn't take an array as the second argument because providers cannot have dependencies on other services.
- The provider also declares a `$get` function on its instance, which is what gets called when the service needs to be initialized. At this point, it can use the state that has been set up in the configuration to instantiate the service as needed.

Let's take a look at the `config` function that we have defined:

- The `config` function executes before the AngularJS app executes. So we can be assured that this executes before our controllers, services, and other functions.
- The `config` function follows the same Dependency Injection pattern, but gets providers injected in. In this case, we ask for the `ItemServiceProvider`.
- At this point, we can now call functions and set values that the provider exposes. We are able to the call the `disableDefaultItems` function that we defined in the provider.
- The `config` function could also set up URL endpoints, locale information, routing configuration for our application, and so on: things that need to be executed and initialized before our application starts.
- We can try changing the value of `shouldHaveDefaults` to `true` (this would come from the server or URL, or some other way usually) to see the effect it has on our application.

Conclusion

We saw the limitations of just using controllers for our entire application, and saw how and when to use AngularJS services. We saw the Dependency Injection syntax in AngularJS when we used the built-in `$log` service. We then covered some of other built-in services before diving into creating our own AngularJS services.

We then created the same AngularJS service (a data store for an array of items) in three different ways, depending on the need and preference. We created the functional form using the factory method, the OO style using the service method, and the configurable version using the provider method.

In the next chapter, we will build on these concepts and start working with server communication in the context of AngularJS.

CHAPTER 6

Server Communication Using $http

In Chapter 5, we looked at AngularJS services and how they differ from controllers. We also explored some basic core AngularJS built-in services, and saw how to create our own AngularJS service as well.

In this chapter, we explore how to start creating applications that can communicate with a server to fetch and store data. In particular, we will work with the `$http` service and save and update information. By the end of the chapter, we as developers should be extremely comfortable working with asynchronous tasks in AngularJS and with server communication, because we have built the infrastructure we might need for a full-fledged application.

Fetching Data with $http Using GET

The traditional way of making a request to the server from AJAX applications (using `XMLHttpRequests`) involves getting a handle on the `XMLHttpRequest` object, making the request, reading the response, checking the error codes, and finally processing the server response. It goes something like this:

```
var xmlhttp = new XMLHttpRequest();

xmlhttp.onreadystatechange = function() {
 if (xmlhttp.readystate == 4 && xmlhttp.status == 200) {
   var response = xmlhttp.responseText;
 } else if (xmlhttp.status == 400) { // or really anything in the 4 series
   // Handle error gracefully
 }
};

// Set up connection
xmlhttp.open("GET", "http://myserver/api", true);
```

```
// Make the request
xmlhttp.send();
```

This is a lot of work for such a simple, common, and often repeated task. More often than not, we will likely end up creating wrappers or using a library.

`$http` is a core AngularJS service that allows us to communicate with server endpoints using XHR. The AngularJS XHR API follows what is commonly known as the *Promise interface*. Because XHRs are asynchronous method calls, the response from the server will come back at an unknown future date and time (hopefully almost immediately). The Promise interface guarantees how such responses will be dealt with, and allows consumers of the Promise to use them in a predictable manner.

Reducing Code with ngResource

In case you have a RESTFUL API on your server, you can further reduce the amount of code you write by using AngularJS's optional module, `ngResource`. `ngResource` allows us to take an API endpoint and create an AngularJS service around it. For example, consider an API for projects on the server side that behaves like the following:

- GET request to */api/project/* returned an array of projects
- GET request to */api/project/17* returned the project with ID 17
- POST request to */api/project/* with a project object as JSON created a new project
- POST request to */api/project/19* with a project object as JSON updated the project with ID 19
- DELETE request to */api/project/* deleted all the projects
- DELETE request to */api/project/23* deleted the project with ID 23

If we have such an API, then instead of manually creating a project resource, and wrapping up `$http` requests individually, we could just create a service as follows:

```
angular.module('resourceApp', ['ngResource'])
  .factory('ProjectService', ['$resource', function($resource) {
      return $resource('/api/project/:id');
}]);
```

This would automatically give us methods on `ProjectService` like:

- `ProjectService.query()` to get a list of projects
- `ProjectService.save({id: 15}, projectObj)` to update a project with ID 15
- `ProjectService.get({id: 19})` to get an individual project with ID 19

and so on. You can read more about the configuration and options you get with `ngResource` at the official AngularJS docs for `ngResource`.

We have a simple server that we can use to create and list some notes that we have made available at our GitHub repository. To set up the server, clone the repository, open the *chapter6* folder, and execute the following commands in succession:

- **`npm install`**
- **`node server.js`**

Then navigate to *http://localhost:8000* and click the first link to see this in action.

We need to serve the HTML and the JavaScript for this chapter from this server so that it can make GET and POST requests successfully. The browser prevents us from making XHR requests to any other domain for security reasons.

Our server exposes the following endpoints:

/api/note
: GET request gives back an array of notes present on the server.

/api/note
: POST request creates a note.

/api/note/:id
: GET request with the `:id` replaced with a numeric ID returns the note with given ID.

/api/note/:id
: POST request with the `id` updates the note with the given ID.

The entire data is stored in memory so if we kill and restart the server, any data we might have added will be destroyed.

Given this, let's see how our app would fetch the list of notes from the server and display them:

```
<!-- File: chapter6/public/http-get-example.html -->
<html ng-app="notesApp">

<head>
  <title>$http get example</title>
  <style>
    .item {
      padding: 10px;
    }
  </style>
</head>
```

```
<body ng-controller="MainCtrl as mainCtrl">
  <h1>Hello Servers!</h1>
  <div ng-repeat="todo in mainCtrl.items" class="item">
    <div><span ng-bind="todo.label"></span></div>
    <div>- by <span ng-bind="todo.author"></span></div>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script>
  angular.module('notesApp', [])
    .controller('MainCtrl', ['$http', function($http) {
      var self = this;
      self.items = [];
      $http.get('/api/note').then(function(response) {
        self.items = response.data;
      }, function(errResponse) {
        console.error('Error while fetching notes');
      });
    }]);
</script>
</body>
</html>
```

In this example, our HTML is quite simple. We have a `div` that has the `MainCtrl` attached to it. Inside the `div`, we have an `ng-repeat` over the `items` array in our controller in which we `ng-bind` to the *label* and *author* fields.

Our controller has a dependency on `$http` as a service. Then, when the controller loads, we make a GET request to the */api/notes* server endpoint. `$http.get()` returns what we call a *Promise object* (more on this in just a bit), which allows us to chain functions as if they were synchronous. Our server call might execute in a jiffy, or might take a few seconds to execute. With a Promise object, we can say, when the server returns a response (whether it is a success or failure), then execute the following function.

This is important because promises (just like callbacks) allow us to deal with scalability issues. Both of these concepts keep JavaScript nonblocking and event-driven, which allows us to let the browser continue to do its work while the server request is in flight. Now, regardless of whether the server has only one request, ten requests, or even a million requests, the client is not stuck waiting for the response. It can continue to do other stuff without making the UI hang.

The `then` function takes two arguments, a *success handler* and an *error handler*. If the server returns a non-200 response, the error handler is called. Otherwise, the success handler is triggered. Both these handlers get passed in a response object, which has the following keys:

headers
: The headers for the call

status
: The status code for the response

config
: The configuration with which the call was made

data
: The body of the response from the server

In the case of success, we just assign the data from the server to the items array and let AngularJS take care of updating the UI through data-binding. In the case of an error, we just log it to the console.

A Deep Dive into Promises

At this point, let's quickly dive into the powerful concept that are *promises*. Promises in AngularJS are based on Kris Kowal's Q proposal, which is a standardized, convenient way of dealing with asynchronous calls in JavaScript.

The traditional way to deal with asynchronous calls in JavaScript has been with callbacks. Say we had to make three calls to the server, one after the other, to set up our application. With callbacks, the code might look something like the following (assuming a `xhrGET` function to make the server call):

```
// Fetch some server configuration
xhrGET('/api/server-config', function(config) {
    // Fetch the user information, if he's logged in
    xhrGET('/api/' + config.USER_END_POINT, function(user) {
        // Fetch the items for the user
        xhrGET('/api/' + user.id + '/items', function(items) {
           // Actually display the items here
        });
    });
});
```

In this example, we first fetch the server configuration. Then based on that, we fetch information about the current user, and then finally get the list of items for the current user. Each `xhrGET` call takes a callback function that is executed when the server responds.

Now of course the more levels of nesting we have, the harder the code is to read, debug, maintain, upgrade, and basically work with. This is generally known as callback hell. Also, if we needed to handle errors, we need to possibly pass in another function to each `xhrGET` call to tell it what it needs to do in case of an error. If we wanted to have just one common error handler, that is not possible.

The Promise API was designed to solve this nesting problem and the problem of error handling. The Promise API proposes the following:

1. Each asynchronous task will return a `promise` object.
2. Each `promise` object will have a `then` function that can take two arguments, a success handler and an error handler.
3. The success or the error handler in the `then` function will be called only once, after the asynchronous task finishes.
4. The `then` function will also return a `promise`, to allow chaining multiple calls.
5. Each handler (success or error) can return a value, which will be passed to the next function in the chain of promises.
6. If a handler returns a promise (makes another asynchronous request), then the next handler (success or error) will be called only after that request is finished.

So the previous example code might translate to something like the following, using promises and the `$http` service:

```
$http.get('/api/server-config').then(function(configResponse) {
    return $http.get('/api/' + configResponse.data.USER_END_POINT);
}).then(function(userResponse) {
    return $http.get('/api/' + userResponse.data.id + '/items');
}).then(function(itemResponse) {
    // Display items here
  }, function(error) {
    // Common error handling
});
```

In this example, we use the `$http` service to make a series of server calls. Each server call using `$http.get` returns a `promise`, and we use the `then` function to add a success handler. In the first two success handlers, we use the response from the server to make another server call.

Each success handler in the promise returns another promise using `$http.get`. AngularJS then waits for that server call to return before proceeding to the next function in the promise chain. Also, the server response value for that promise will be passed as an argument to the next success handler in the chain. So, the first `then` will get the `configResponse`. The second `then` will get the return value of the `configResponse` success handler, which is the `userResponse`, and so on.

Also, we have one error handler, which we pass as the second argument to the very last function in the promise chain. Because of this, if any error happens in any of the functions in the promise chain, AngularJS will find the next closest error handler and trigger it. So regardless of whether the error happens in the config request or the user request, the common error handling function will get called.

Propagating Success and Error

Chaining promises is a very powerful technique that allows us to accomplish a lot of functionality, like having a service make a server call, do some postprocessing of the data, and then return the processed data to the controller. But when we work with promise chains, there are a few things we need to keep in mind.

Consider the following hypothetical promise chain with three promises, P1, P2, and P3. Each promise has a success handler and an error handler, so S1 and E1 for P1, S2 and E2 for P2, and S3 and E3 for P3:

```
xhrCall()
    .then(S1, E1) //P1
    .then(S2, E2) //P2
    .then(S3, E3) //P3
```

In the normal flow of things, where there are no errors, the application would flow through S1, S2, and finally, S3. But in real life, things are never that smooth. P1 might encounter an error, or P2 might encounter an error, triggering E1 or E2.

Now, depending on the return value of any of these handlers, AngularJS will decide which function in the chain to execute next. At each of these handlers, we as developers have control. We can decide, given the current handler, which function in the chain to execute next. Consider the following cases:

- We receive a successful response from the server in P1, but the data returned is not correct, or there is no data available on the server (think empty array). In such a case, for the next promise P2, it should trigger the error handler E2.
- We receive an error for promise P2, triggering E2. But inside the handler, we have data from the cache, ensuring that the application can load as normal. In that case, we might want to ensure that after E2, S3 is called.

So each time we write a success or an error handler, we need to make a call—given our current function, is this promise a success or a failure for the next handler in the promise chain?

If we want to trigger the success handler for the next promise in the chain, we can just return a value from the success or the error handler, and AngularJS will treat it as us successfully resolving any errors.

If, on the other hand, we want to trigger the error handler for the next promise in the chain, we can leverage the `$q` service in AngularJS. Just ask for `$q` as a dependency in our controller and service, and return `$q.reject(data)` from the handler. This will ensure that the next promise in the chain goes into the error condition, and will get the data passed to it as an argument.

The $q Service

The $q service in AngularJS has the following APIs for us to use in our application:

`$q.defer()`
: Creates a deferred object when we need to create a promise for our own asynchronous task. Most asynchronous tasks in AngularJS (server calls, timeouts, and intervals) return a promise, but if we are integrating with a third-party library, then we might need our own promise. The `$q.defer()` is useful in those times because the deferred object has a `promise` attribute that can be returned from a function.

`deferredObject.resolve`
: The deferred object created by the previous function can be resolved successfully at any point by calling the `resolve()` function on it with the argument being the data passed to the success handler in the promise chain.

`deferredObject.reject`
: The deferred object can also be rejected, thus denoting that the promise was a failure and triggering the failure handler in the promise. Again, the argument passed to it will be passed to the error handler as is.

`$q.reject`
: The `$q.reject()` can be called from within any of the promise handlers (success or error) with an optional argument that denotes the value to be passed along in the promise chain. The return value of this should be returned to ensure that the promise continues to the next error handler instead of the success handler in the promise chain.

Making POST Requests with $http

Given this background, let's now build out the rest of the UI for this server communication example. We will now add a section that allows users to add notes to display in this list:

```
<!-- File: chapter6/public/http-post-example.html -->
<html ng-app="notesApp">

<head>
  <title>HTTP Post Example</title>
  <style>
    .item {
      padding: 10px;
    }
  </style>
</head>

<body ng-controller="MainCtrl as mainCtrl">
  <h1>Hello Servers!</h1>
```

```
  <div ng-repeat="todo in mainCtrl.items"
       class="item">
    <div><span ng-bind="todo.label"></span></div>
    <div>- by <span ng-bind="todo.author"></span></div>
  </div>

  <div>
    <form name="addForm"
          ng-submit="mainCtrl.add()">
      <input type="text"
             placeholder="Label"
             ng-model="mainCtrl.newTodo.label"
             required>
      <input type="text"
             placeholder="Author"
             ng-model="mainCtrl.newTodo.author"
             required>
      <input type="submit"
             value="Add"
             ng-disabled="addForm.$invalid">
    </form>
  </div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>
<script>
  angular.module('notesApp', [])
    .controller('MainCtrl', ['$http', function($http) {
      var self = this;
      self.items = [];
      self.newTodo = {};
      var fetchTodos = function() {
        return $http.get('/api/note').then(
            function(response) {
          self.items = response.data;
        }, function(errResponse) {
          console.error('Error while fetching notes');
        });
      };

      fetchTodos();

      self.add = function() {
        $http.post('/api/note', self.newTodo)
             .then(fetchTodos)
             .then(function(response) {
               self.newTodo = {};
             });
      };

    }]);
```

```
</script>
</body>
</html>
```

In this example, we added a new section to the HTML that has a standard form with two input fields. We bind these two input fields to our model, `newTodo`, in the controller. On submit of the form, we trigger the `add()` function in our controller.

Our controller has slightly changed from the previous example as well. The fetching of notes from the server is now wrapped inside the `fetchTodos()` function, which in addition to making the `$http.get` server call also returns the `promise` for the async call. This function is triggered once when the controller loads.

The `add` function also uses the `$http` service, and calls `$http.post`. Unlike the GET, which takes one argument, the URL of the server, the POST request takes two arguments: the URL and the post data. We chain this server call to call `fetchTodos` on a successful creation of the todo. We then finally add another promise to the chain, which will clear the `newTodo` object. This last promise handler will only get triggered after the server call to create the todo, and the server call to get the list of todos (because of the promise returned by the `fetchTodos` function) both finish.

$http API

We have been using `$http` to get and save data, so let's take a look at the actual API that the `$http` service in AngularJS provides.

`$http` provides the following convenience methods to make certain types of requests:

- GET
- HEAD
- POST
- DELETE
- PUT
- JSONP

So just like `$http.get`, we can use `$http.put` or `$http.delete`. Each of these method signatures are in one of two patterns:

- For requests without any post data (think GET), the function takes two arguments: the URL as the first argument, and a configuration object as the second.
- For requests with post data (POST, PUT), the function takes three arguments: the URL as the first argument, the post data as the second, and a configuration object as the third and final argument.

Each of these is a convenience method, and can actually be directly called through `$http` itself. That is:

```
$http.get(url, config)
```

can be replaced with:

```
$http(config)
```

where the `url` and the `method` (GET, in this case) become part of the configuration object itself. In each of the convenience methods (`$http.get`, `$http.post`, etc.), the `config` object, which is the last parameter, is optional. So we can call `$http.get` with only the URL like we did in these examples.

Configuration

We have been mentioning this configuration object for the past few paragraphs, so let's take a look at some acceptable parameters and values for it. The following is a basic pseudocode template for the configuration object, which details the keys that are acceptable and the type of value that it expects:

```
{
  method: string,
  url: string,
  params: object,
  data: string or object,
  headers: object,
  xsrfHeaderName: string,
  xsrfCookieName: string,
  transformRequest: function transform(data, headersGetter) or
                    an array of functions,
  transformResponse: function transform(data, headersGetter) or
                     an array of functions,
  cache: boolean or Cache object,
  timeout: number,
  withCredentials: boolean
}
```

The GET, POST, and other convenience methods set the method parameter, so we don't need to. Similarly, if we give the GET or POST requests a URL, it gets set in the config automatically.

We can change the request or how it behaves by passing the `config` object set with the following keys:

`method`
: A string representing the HTTP request type, like GET or POST.

`url`
: A URL string representing the absolute or relative URL of the resource being requested.

`params`
: A JavaScript object with keys and values translating to URL query parameter keys and values. For example:

```
[{key1: 'value1', key2: 'value2'}]
```

would be converted to:

```
?key1=value1&key2=value2
```

after the URL. If we use an object instead of a string or a number for the value, the object will be converted to a JSON string.

`data`
: A string or an object that will be sent as the request message data. This basically becomes the POST data for the server.

`headers`
: An object (or map) with each key being the name of the header, and the value being the value of that particular header. So passing `{'Content-Type': 'text/csv'}` would set the Content-Type header to be text/csv.

`xsrfHeaderName`
: We can set the XSRF header that the server will be setting to prevent XSRF attacks on our website. This will then be used in the request to ensure the XSRF handshake happens with our server.

`xsrfCookieName`
: The name of the cookie that has the `xsrf` token to be used for the XSRF handshake.

`transformRequest` *and* `transformResponse`
: These provide a way for us to change the data for the request going out or the response coming in. These take a single function (which gets passed the data and a way to get the headers) or an array of these functions. Each of these functions can take the data (which is either the post data being sent, or the data of the response), and then return the converted, transformed data from it. A simple `transformRequest` that takes the JSON post data and converts it into jQuery like a post data string is as follows:

```
transformRequest: function(data, headers) {
  var requestStr;
  for (var key in data) {
    if (requestStr) {
      requestStr += '&' + key + '=' + data[key];
    } else {
      requestStr = key + '=' + data[key];
    }
  }
```

```
        return requestStr;
      }
```

`cache`
: A Boolean or a cache object to use for an application-level caching mechanism. This would be over and above the browser-level caching. If set to true, AngularJS will automatically cache server responses and return them for subsequent requests to the same URL.

`timeout`
: The time in milliseconds to wait before the request is treated as timed out. This can also be a promise object, which when rejected tells AngularJS to abandon the server call.

Advanced $http

Until now, we have seen how to make simple GET and POST requests using the `$http` service and have looked into some of the configuration we can do at a request level. Until this point though, we have been dealing with the `$http` service on a request level. The `$http` service also allows us to configure defaults, or intercept each and every request going out and response coming in to have some common handling. We will deal with these in this section.

Configuring $http Defaults

The first thing we want to look at is how to configure `$http` defaults. We saw how to add transformations and headers as part of the config of a single `$http` request in the previous section. If we needed to add a caching header as part of each and every request, it could quickly become annoying if we did it each time we call `$http`. For these kinds of requirements, we can use the `config` section of our module, and use the `$httpProvider` to configure these defaults. Let's see how we might configure some headers and a default `transformRequest` using the `$httpProvider`:

```
// File: chapter6/public/http-defaults.js

angular.module('notesApp', [])
  .controller('LoginCtrl', ['$http', function($http) {
    var self = this;
    self.user = {};
    self.message = 'Please login';
    self.login = function() {
      $http.post('/api/login', self.user).then(
        function(resp) {
         self.message = resp.data.msg;
      });
    };
  }])
```

```
.config(['$httpProvider', function($httpProvider) {
  // Every POST data becomes jQuery style
  $httpProvider.defaults.transformRequest.push(
      function(data) {
    var requestStr;
    if (data) {
      data = JSON.parse(data);
      for (var key in data) {
        if (requestStr) {
          requestStr += '&' + key + '=' + data[key];
        } else {
          requestStr = key + '=' + data[key];
        }
      }
    }

    return requestStr;
  });
  // Set the content type to be FORM type for all post requests
  // This does not add it for GET requests.
  $httpProvider.defaults.headers.post['Content-Type'] =
      'application/x-www-form-urlencoded';
}]);
```

In this example, we set up some application-level configuration for the `$http` service. We do this by creating a config section for our module and getting the `$httpProvider` injected into it. As part of the configuration, we first add a global request transformer, which changes the post data for any outgoing request from a JSON object into a jQuery post string format. Notice that we push this transformation function into the default transformers.

We would do something like the preceding example when we are dealing with a backend that is configured to accept Content-Type *text/www-form-urlencoded*, which is what jQuery defaults to. AngularJS defaults to *application/json*, which is recommended for web applications. If you can't change your backend to accept *application/json*, then you can add a transformer and header to get your AngularJS application talking to your backend.

We can have multiple transformers for requests and responses, both at an individual request level as well as a global level, so the `transformRequest` and `transformResponse` are arrays by default. We just push (and unshift) as necessary the functions we want to add.

Then we also add a default header to all outgoing GET requests. The `$httpProvider.defaults.headers` object allows us to set default headers for `common`, `get`, `post`, and `put` requests. Each one (`$httpProvider.defaults.headers.post`, for example) is an object, where the key is the header name and the value is the value of the header. In this example, we set the Content-Type for all outgoing POST requests.

The following is the list of keys and values that can have defaults set using `$httpProvider` (using `$httpProvider.defaults`):

- `headers.common`
- `headers.get`
- `headers.put`
- `headers.post`
- `transformRequest`
- `transformResponse`
- `xsrfHeaderName`
- `xsrfCookieName`

`transformRequest` and `transformResponse` are arrays of functions. The XSRF-related keys take pure string values. The headers are all object maps with keys being the header names and the values being the value of the header.

Interceptors

Handling request-level actions (such as logging, authentication check, and handling certain types of responses) globally has always been challenging. It usually required planning to create a layer through which all requests would be channeled so that we could add global hooks.

AngularJS immensely simplifies this using `$httpProvider` to set up *interceptors*. AngularJS interceptors allow us to hook and check each request and response and handle certain events (like the server returning 403s for authorization issues) in a common way.

Do note that the old style of creating response interceptors has been deprecated and is not expected to be available in future versions of AngularJS. Therefore, do not use `$httpProvider.responseInterceptors` in your code anymore.

When we create an interceptor (and we can create multiple), AngularJS makes sure that it is called before any request is made to the server. Similarly, AngularJS makes sure that it is called first before the controller or service that makes the `$http` call. So it gives us a common pipe to work with. Let's take a quick look at how we implement one:

```
// File: chapter6/public/logging-interceptor.js
angular.module('notesApp', [])
  .controller('MainCtrl', ['$http', function($http) {
    var self = this;
    self.items = [];
```

```
  self.newTodo = {};
  var fetchTodos = function() {
    return $http.get('/api/note').then(function(response) {
      self.items = response.data;
    }, function(errResponse) {
      console.log('Error while fetching notes');
    });
  };

  fetchTodos();

  self.add = function() {
    $http.post('/api/note', self.newTodo)
      .then(fetchTodos)
      .then(function(response) {
        self.newTodo = {};
      });
  };

}]).factory('MyLoggingInterceptor', ['$q', function($q) {
  return {
    request: function(config) {
      console.log('Request made with ', config);
      return config;
      // If an error, not allowed, or my custom condition,
      // return $q.reject('Not allowed');
    },
    requestError: function(rejection) {
      console.log('Request error due to ', rejection);
      // Continue to ensure that the next promise chain
      // sees an error
      return $q.reject(rejection);
      // Or handled successfully?
      // return someValue
    },
    response: function(response) {
      console.log('Response from server', response);
      // Return a promise
      return response || $q.when(response);
    },
    responseError: function(rejection) {
      console.log('Error in response ', rejection);
      // Continue to ensure that the next promise chain
      // sees an error
      // Can check auth status code here if need to
      // if (rejection.status === 403) {
      //   Show a login dialog
      //   return a value to tell controllers it has
      //   been handled
      // }
      // Or return a rejection to continue the
      // promise failure chain
```

```
        return $q.reject(rejection);
      }
    };
  }])
  .config(['$httpProvider', function($httpProvider) {
    $httpProvider.interceptors.push('MyLoggingInterceptor');
  }]);
```

In this example, we implement an interceptor that simply logs every single outgoing request and incoming response from the server. We implement interceptors in AngularJS as factories, which return an object with any or all of the following four methods:

`request`
: Any outgoing request passes through the request function, which is also passed the configuration with which the request is being made. At this point, we can take a look at the URL, the post data, the method (whether it is a GET or POST), etc., and then decide to continue with the request (in which case we return the config), or we can decide to reject it to prevent the request from being made (using `return $q.reject`, which rejects the promise).

`requestError`
: This is triggered if there are multiple interceptors and one of them rejected the request going out. In that case, the reason for the rejection (the argument to `$q.reject`) is passed to this function.

`response`
: When the server eventually returns, this function is called with the response object (which holds the configuration, status code, headers, and data). If we need to check the validity of the data or a particular header, or log the response, this is the place to do it.

`responseError`
: If the server returns with a non-200 series status code, AngularJS treats it as a response error. We get the same response configuration handled here, where we can check the status, do additional work (like show a login dialog if the status is a 403), and then finally continue returning a rejection (to tell future promises to treat it as a failure), or return a value to say all errors have been handled successfully.

This factory basically dictates how our interceptor works, and how it handles each of these four cases. In any interceptor, we might decide that we only care about `responseErrors`, so we can implement a factory that returns an object with only that function.

We finally hook it up with the `$http` service through the `$httpProvider` in the `config` function. The `$httpProvider` has an interceptors array on which we can push interceptors by name. So we simply push the `MyLoggingInterceptor` onto it, and that automatically adds the interceptor after the AngularJS app has finished loading.

Best Practices

We have dived through the depths of `$http`, seen how to configure requests, intercept all responses, and much more. With all that behind us, here are a few things we should keep in mind when we work with `$http`:

Wrap `$http` *in services*
: In the previous examples, we directly called `$http.get` or `post` in our controllers. In a real application, we should do this: instead of calling `$http.get(/api/notes)` directly from our controller, we should wrap that call in a service so that we can do something like `NoteService.query()`, which in turn would do the `$http.get` call. This service call can then return the promise so that the controller can chain and handle the response correctly:

```
angular.module('notesApp', [])
    .factory('NoteService', ['$http', function($http) {
        return {
            query: function() {
                return $http.get('/api/notes');
            }
        };
    }]);
```

: This example code shows how such a `NoteService` might look. All it does is wrap the `$http` call inside a service method, and return the promise from it to which controllers and other services can add their functionality on the chain.

Use interceptors
: There are some common tasks that we might want to do every time a request goes out from the client, such as logging the request or adding some authorization headers to the request. Or tasks we might need accomplished or conditions we might want to check on every response. In such a case, interceptors are our best bet. A simple interceptor that handles 403s, as well as adds the authorization header on every request, might look something like the following:

```
angular.module('notesApp', [])
  .factory('AuthInterceptor',
    ['AuthInfoService', '$q', function(AuthInfoService, $q) {
      return {
        request: function(config) {
          if (AuthInfoService.hasAuthHeader()) {
            config.headers['Authorization'] =
                AuthInfoService.getAuthHeader();
```

```
        }
        return config;
      },
      responseError: function(responseRejection) {
        if (responseError.status === 403) {
          // Authorization issue, access forbidden
          AuthInfoService.redirectToLogin();
        }
        return $q.reject(responseRejection);
      }
    };
  }])
  .config(['$httpProvider', function($httpProvider) {
    $httpProvider.interceptors.push('AuthInterceptor');
  }]);
```

In this example, we added an interceptor that only intercepts outgoing requests and incoming responses with a non-200 status code. In the case of an outgoing request, we add the authorization header if it is present in a service called `AuthInfoService`. In the case of the responses, we check if the status is a 403 and if so, redirect the user to the login page. We ensure that the promise is rejected so that the controller or service still sees a failure. The implementation of `AuthInfoService` can be as per the project's needs.

Chain interceptors
: Instead of creating one giant interceptor to do all our intercepting work, we create multiple tiny interceptors, each with individual responsibility. We have a separate interceptor for authorization, a separate one for logging, and so on. The interceptors will be called in the order we add them to the provider, so we can also control the order in which they are called.

Leverage defaults
: If we find ourselves setting the same headers again and again, or adding the same request or response transformation, then we should heavily consider using defaults. If all our endpoints return XML instead of JSON, add a default `transformResponse` to the `$httpProvider` that takes the XML and converts it into JSON (or vice versa if our server only accepts XML). We can also set defaults for only GET or POST requests, so we can be as specific or generic with our defaults as needed.

Conclusion

We saw how to do the very simple task of making GET and POST requests to the server using the `$http` service in AngularJS. We used the `$http` service to fetch a list of notes from the server as well as add notes to be persisted on the server. We then dove into some of the configuration parts of AngularJS, to see how to change request options such as headers and transformers. We then looked at how to set defaults for HTTP requests, as well how and when to use interceptors and transformers. We demonstrated a few examples of interceptors, incuding logging interceptors and even authorization interceptors. At this point, any task using `$http` should be straightforward.

In the next chapter, we will leverage `$http` and start building out a full-fledged application with multiple URLs and routes, and show views depending on the context. We will use the core AngularJS `ngRoute` module to accomplish this.

CHAPTER 7

Unit Testing Services and XHRs

In Chapters 5 and 6, we learned how to leverage existing AngularJS services, as well as create our very own AngularJS services. We created simple AngularJS services that we used to store state and communicate across different parts of the application, and services to allow for HTTP communication with our servers.

In Chapter 3, we saw how we might unit test our controllers. Now that we have started creating our own AngularJS services, we will look at how to unit test them. In particular, we will test controllers that use built-in AngularJS services, as well as our very own services. Finally, we will write unit tests for services and controllers that make HTTP requests, and see how we can mock out and leverage the AngularJS Dependency Injection.

Dependency Injection in Our Unit Tests

In Chapter 3, we saw how to leverage AngularJS Dependency Injection in our unit tests to test controllers. We asked for the `$controller` service, and then created controller instances as and when we needed them. `$controller` is actually an AngularJS service that we ask for in the unit test.

Similary, we can ask for any service that AngularJS knows about in our unit test, whether it comes from core AngularJS or is one of our own creations. AngularJS will figure out how to create it, what its dependencies are, and give us a fully instantiated service for testing.

Don't forget that to run these tests, you need to:

1. Switch to the *chapter7* folder.
2. Run **`npm install karma karma-jasmine karma-chrome-launcher`**.
3. Run **`karma start`**.

This will ensure that all the dependencies are installed correctly for you to run the Karma unit tests.

The examples and tests in this book were run using Karma version 0.12.16 and AngularJs version 1.2.19 (both the *angular.js* and *angular-mocks.js* files). If you are having trouble running them for any reason, ensure that you are using the same versions.

We will use the following Karma configuration for this chapter:

```
// File:  chapter7/karma.conf.js
// Karma configuration

module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      'angular.min.js',
      'angular-mocks.js',
      '*.js'
    ],
    exclude: [],
    port: 8080,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  });
};
```

Let's see how we might test whether the controller redirects us to a new URL when a function is called in the controller:

```
// File: chapter7/simpleCtrl1.js

angular.module('notesApp', [])
  .controller('SimpleCtrl', ['$location', function($location) {
    var self = this;
    self.navigate = function() {
      $location.path('/some/where/else');
    };
  }]);
```

This controller is as simplistic as it can get while depending on a core AngularJS service. All it does is provide a function called `navigate`, which changes the current location in the browser to */some/where/else*. Now let's see how its unit test might look:

```
// File: chapter7/simpleCtrl1Spec.js

describe('SimpleCtrl', function() {
  beforeEach(module('notesApp'));

  var ctrl, $loc;
  beforeEach(inject(function($controller, $location) {
    ctrl = $controller('SimpleCtrl');
    $loc = $location;
  }));

  it('should navigate away from the current page', function() {
    $loc.path('/here');
    ctrl.navigate();
    expect($loc.path()).toEqual('/some/where/else');
  });
});
```

This code snippet is a simple unit test for the `SimpleCtrl` that we defined earlier. All we attempt to do is to test the `navigate` function in the controller. All the navigate function does is redirect the user to */some/where/else*. Now, if this were a real live browser, the URL would change in the browser bar and an actual page navigation would happen. We don't want this happening in our unit test, so the *angular-mocks.js* file that we included as part of the Karma configuration provides mocked-out versions of services like `$location` and `$window`. This allows us to unit test without worrying about affecting the browser or things like global state that might affect our unit tests.

The mocked-out version of the `$location` (for which we did not have to change a single line of production code, thanks to Dependency Injection!) allows us to set an initial state of the browser's location (*/here* in this case). After executing the `ctrl.navigate()` function, we can then set an expectation that the `$location.path` be set to */some/where/else*. Neither of these will change the browser's actual URL, so the unit tests will complete as normal.

State Across Unit Tests

Let's modify the previous code and unit tests to add two functions and two tests to demonstrate how state is shared (or not shared) between tests:

```
// File: chapter7/simpleCtrl2.js

angular.module('simpleCtrl2App', [])
  .controller('SimpleCtrl2', ['$location', '$window',
    function($location, $window) {
      var self = this;
```

```
      self.navigate1 = function() {
        $location.path('/some/where');
      };
      self.navigate2 = function() {
        $location.path('/some/where/else');
      };
    }]);
// File: chapter7/simpleCtrl2Spec.js

describe('SimpleCtrl2', function() {
  beforeEach(module('simpleCtrl2App'));

  var ctrl, $loc;
  beforeEach(inject(function($controller, $location) {
    ctrl = $controller('SimpleCtrl2');
    $loc = $location;
  }));

  it('should navigate away from the current page', function() {
    expect($loc.path()).toEqual('');
    $loc.path('/here');
    ctrl.navigate1();
    expect($loc.path()).toEqual('/some/where');
  });

  it('should navigate away from the current page', function() {
    expect($loc.path()).toEqual('');
    $loc.path('/there');
    ctrl.navigate2();
    expect($loc.path()).toEqual('/some/where/else');
  });
});
```

We added two functions to our controller. Both the `navigate1` and `navigate2` functions in the controller navigate to a URL. If we call `navigate1` first, it navigates to */some/where*. If we call `navigate2` next, the URL changes to */some/where/else*. The change from the first `navigate1` function (the redirected URL) is visible at the beginning of the `navigate2` function call.

With this context, let's now look at our unit tests. There are two unit tests, one for each of the two functions. And each one checks that the `$location` path changes to the correct URL after the function is called. What is noteworthy is the pre-function call check we do. In both cases, we can expect that the browser URL is an empty string.

The reason we can do that is because there is no global state in our unit test. If we execute the first test in a real live application, that changes the value of the browser URL. The second test would then see that. This makes the order of the tests important, because something that sets a variable that the other test uses could cause the tests to fail if they are in a certain order, but pass in another.

AngularJS avoids that by getting rid of global state in the unit tests. The `$location` service is destroyed and created between our unit tests. All of this happens because we instantiate our module before each unit test. This is responsible for creating a fresh version of each of the service that our test uses.

Mocking Out Services

What if we had a service that was really heavy, or we did not want to test the service? We saw mocked-out versions of `$location` and `$window` that the AngularJS mock file provides. In this section, we see how to create our own mocks. Let's consider the very simple `ItemService` from Chapter 5:

```
// File: chapter7/notesApp1.js
angular.module('notesApp1', [])
  .factory('ItemService', [function() {
    var items = [
      {id: 1, label: 'Item 0'},
      {id: 2, label: 'Item 1'}
    ];
    return {
      list: function() {
        return items;
      },
      add: function(item) {
        items.push(item);
      }
    };
  }])
  .controller('ItemCtrl', ['ItemService', function(ItemService) {
    var self = this;
    self.items = ItemService.list();
  }]);
```

This code snippet uses the `ItemService` we defined in Chapter 5, and has a simple controller that fetches the list of items when it loads. Now for the purpose of our unit test, we want to mock out `ItemService` so that we can override the default implementation for our unit test. There are two ways to accomplish this.

The first way is to override the service during the unit test, as an inline mock:

```
// File: chapter7/notesApp1Spec.js

describe('ItemCtrl with inline mock', function() {
  beforeEach(module('notesApp1'));

  var ctrl, mockService;

  beforeEach(module(function($provide) {
    mockService = {
      list: function() {
```

```
        return [{id: 1, label: 'Mock'}];
      }
    };

    $provide.value('ItemService', mockService);
  }));

  beforeEach(inject(function($controller) {
    ctrl = $controller('ItemCtrl');
  }));

  it('should load mocked out items', function() {
    expect(ctrl.items).toEqual([{id: 1, label: 'Mock'}]);
  });

});
```

In this unit test, the start of the test is similar, where we instantiate our module, the `notesApp1`. After that, we have another `beforeEach`, which is where we override the `ItemService` with our own mock. We use the `module` function, but instead of giving it the name of the module, we give it a function that gets injected with a `$provide`. This provider shares its namespace with the modules loaded before. So now we create our `mockService` and tell the provider that when any controller or service asks for `ItemService`, give it our value. Because we do this *after* the `notesApp1` module is loaded, it overwrites the original `ItemService` definition.

The rest of the unit test proceeds the same as before, except we now check that the value of items in the controller is returned by our mock instead of the original service.

The second option to override services would be at a global level instead of a unit test level. To decide whether to create the mocks as we did in the previous example using a local variable and the `$provide.value` function, or whether to do it globally like AngularJS does it, the question we need to answer is whether or not other tests could reuse the mock.

The mock we created before would only be usable within this particular `describe` block. To change the preceding to be a more reusable, general-purpose mock of the `ItemService`, we could do the following:

```
// File: chapter7/notesApp1-mocks.js

angular.module('notesApp1Mocks', [])
  .factory('ItemService', [function() {
    return {
      list: function() {
        return [{id: 1, label: 'Mock'}];
      }
    };
  }]);
```

What we had hardcoded in the `mockService` has been extracted out into a service with the same name, but in a different module named `notesApp1Mocks`. This file will reside in the test folder, and be included by *karma.conf.js*, but not in our live application. Our tests would now change as follows:

```
// File: chapter7/notesApp1SpecWithMock.js

describe('ItemCtrl With global mock', function() {

  var ctrl;
  beforeEach(module('notesApp1'));
  beforeEach(module('notesApp1Mocks'));

  beforeEach(inject(function($controller) {
    ctrl = $controller('ItemCtrl');
  }));

  it('should load mocked out items', function() {
    expect(ctrl.items).toEqual([{id: 1, label: 'Mock'}]);
  });

});
```

This ensures that after `notesApp1` is loaded, we load the `notesApp1Mocks` module, which overrides the `ItemService`. After that, when our test loads the controller, which then calls the service, it defers to the mocked-out `ItemService` that we created.

We can use this approach when we need a global reusable mock, and defer to the `describe`-level mock when we need to mock just one particular test.

Spies

But what if we didn't want to implement an entire mocked-out service? What if we just wanted to know in the case of `ItemService` whether or not the `list` function was called, and not worry about the actual value from it? For those kinds of cases, we have *Jasmine spies*. Spies allow us to hook into certain functions, and check whether they were called, how many times they were called, what arguments they were called with, and so on.

So let's see how to change our mock to use spies instead:

```
// File: chapter7/notesApp1SpecWithSpies.js

describe('ItemCtrl with spies', function() {
  beforeEach(module('notesApp1'));

  var ctrl, itemService;

  beforeEach(inject(function($controller, ItemService) {
    spyOn(ItemService, 'list').andCallThrough();
    itemService = ItemService;
```

```
    ctrl = $controller('ItemCtrl');
  }));

  it('should load mocked out items', function() {
    expect(itemService.list).toHaveBeenCalled();
    expect(itemService.list.callCount).toEqual(1);
    expect(ctrl.items).toEqual([
      {id: 1, label: 'Item 0'},
      {id: 2, label: 'Item 1'}
    ]);
  });
});
```

We call the `spyOn` function with an object as the first argument, and a string with the function name that we want to hook on to as the second argument. In this example, we tell Jasmine to spy on the `list` function of the `ItemService`. We also tell it to continue calling the actual service underneath by calling `andCallThrough` on the spy. This means we can use Jasmine to check whether or not the function was called, and have the function work as it used to underneath.

This adds a wrapper around the existing `ItemService.list` function. Jasmine lets the existing code continue as is while giving us a window into what is happening, and letting us know whether the right functions were called. The data that is returned is still from the original service, as we can see in the expectation on the controllers items. Note that it is recommended that you set up all your mocks and spies before instantiating your controllers.

What if we wanted to not have the existing method execute as normal? Let's see how we might override the method using spies:

```
// File: chapter7/notesApp1SpecWithSpyReturn.js

describe('ItemCtrl with SpyReturn', function() {
  beforeEach(module('notesApp1'));

  var ctrl, itemService;

  beforeEach(inject(function($controller, ItemService) {

    spyOn(ItemService, 'list')
        .andReturn([{id: 1, label: 'Mock'}]);
    itemService = ItemService;
    ctrl = $controller('ItemCtrl');
  }));

  it('should load mocked out items', function() {
    expect(itemService.list).toHaveBeenCalled();
    expect(itemService.list.callCount).toEqual(1);
    expect(ctrl.items).toEqual([{id: 1, label: 'Mock'}]);
```

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

In this example, we override the `list` method in the `ItemService`, and replace it with our Jasmine spy. The `spyOn` function returns a spy that's called with the `andReturn` function on the spy created by `createSpy`, and gives it the value to return. Note that we do this before creating our controller, which is *recommended*. Then, in our unit test, we can check if `ItemService.list` was called, and if it was called once. Also, we specify that our spy return the value in the controller's `items` array (specified with the `andReturn` on the `createSpy` function).

Unit Testing Server Calls

We covered how to unit test simple services, as well as mock services and functions depending on our need. With this in our toolbelt, now let's explore how we might test controllers and services using the `$http` service to make server calls.

In unit tests, we focus on testing a single unit of code and checking whether it behaves correctly under all conditions. In a unit test, we want to mock out the larger system at play, whether that be a server, other third-party dependencies, the DOM and browser, or whatever.

With AngularJS, as long as we include the *angular-mocks.js* file as part of the Karma configuration, AngularJS takes care of ensuring that when we use the `$http` service, it doesn't actually make server calls. All server calls are intercepted, and we can test them all within the context of a unit test. Because they are intercepted and mocked out, our unit tests remain fast and stable.

Let's take a sample controller that makes server calls using `$http`, and see how we might unit test it:

```
// File: chapter7/serverApp.js

angular.module('serverApp', [])
  .controller('MainCtrl', ['$http', function($http) {
    var self = this;
    self.items = [];
    self.errorMessage = '';

    $http.get('/api/note').then(function(response) {
      self.items = response.data;
    }, function(errResponse) {
      self.errorMessage = errResponse.data.msg;
    });
  }]);
```

In this code snippet, we have a very simple controller, which makes a GET request to */api/note* when it loads, and saves the response into the `items` array on the controller.

In case of an error, it saves the error message on the controller's instance. Now, let's see how we might test this:

```
// File: chapter7/serverAppSpec.js

describe('MainCtrl Server Calls', function() {
  beforeEach(module('serverApp'));

  var ctrl, mockBackend;

  beforeEach(inject(function($controller, $httpBackend) {

    mockBackend = $httpBackend;
    mockBackend.expectGET('/api/note')
        .respond([{id: 1, label: 'Mock'}]);
    ctrl = $controller('MainCtrl');
    // At this point, a server request will have been made
  }));

  it('should load items from server', function() {
    // Initially, before the server responds,
    // the items should be empty
    expect(ctrl.items).toEqual([]);

    // Simulate a server response
    mockBackend.flush();

    expect(ctrl.items).toEqual([{id: 1, label: 'Mock'}]);
  });

  afterEach(function() {
    // Ensure that all expects set on the $httpBackend
    // were actually called
    mockBackend.verifyNoOutstandingExpectation();

    // Ensure that all requests to the server
    // have actually responded (using flush())
    mockBackend.verifyNoOutstandingRequest();
  });
});
```

To unit test our controller that makes XHR calls, we leverage a service called `$httpBackend`. The `$http` service internally uses the `$httpBackend` to make the actual XHR requests. The *angular-mocks.js* file provides a mock `$httpBackend` service that prevents server calls, and gives us hooks to set expectations and trigger responses.

As part of `beforeEach`, we ask for the `$httpBackend` service to be injected into the test. Because our controller makes a server call as part of the loading behavior, it is important for us to set our expectations on what server calls will be made before the controller is instantiated.

There are two ways to set expectations on what server calls will be made on the `$httpBackend`:

`expect`
: The `expect` function is used when we want to control exactly how many requests will be made and to what URLs, and then control the response. The `expect` function has a series of functions, one for each method of HTTP, such as `expectGET` or `expectPOST`. The first argument to the function is the URL and the second argument, if provided, acts as the POST data. So `expectGET('/api/notes')` in the previous example says that there will be a GET request to the given URL. Similarly, `expectPOST('/api/notes', {label: 'Hi'})` tells the service to expect a POST request, and that the POST data should exactly match what is passed as the second argument.

`when`
: Similar to `expect`, `when` also takes a URL and potential POST data. The syntax is also exactly the same. The difference is that the `when` does not care about the order of requests or how many times the call was made. It simply sees a request and sends a response. With the `expect`, a test can fail if the expectation was not satisfied. With `when`, even if the test never makes the call, the test will pass.

The difference really comes down to the fact that `expect` is more fine-grained and sets expectations. `when` stubs out the backend (a stub is something that returns the same response, regardless of the request), allowing it to respond in a consistent manner without any expectations for any and all requests.

After we use either `expect` (like `expectGET`) or `when`, we can define the response for that particular server call by chaining the `respond` function on it. If `respond` is given one argument, it is treated as the server response. You can optionally give it two arguments, in which case the first argument will be the status code, and the second argument will be the body of the response (like `respond(404, {msg: 'Invalid'})`). In our example, we respond with a list of items from the server.

Now, on to our actual unit test. When the controller loads, the `items` array is initialized to an empty array. Our first expectation in the test is whether the `items` array is empty. If we consider a server request in a real live application, a request is made first, and then the response comes back at some later point in time. The server requests are asynchronous in nature. To simulate this, AngularJS gives a `flush` method on the backend service. So by default, when a server request happens, AngularJS tracks it against the expectations and holds on to the request without returning the response. Then, when you as a developer finally call `flush()` on the `$httpBackend`, AngularJS sends back the responses for all the requests that the client has received so far.

`flush` allows us to test asynchronous behavior without actually writing asynchronous tests. `$httpBackend.flush()` also takes an integer argument, which can tell the mock backend how many server requests it needs to return. This is useful if we want to check that the controller makes four server calls, but does some work only after at least three of them return. In such a case, we can flush the requests one at a time (using `$httpBackend.flush(1)`), or flush three of them (`$httpBackend.flush(3)`) at once.

At this point, now we can check whether the data that the server responded with has been stored in the right variable in the controller (the `items` array).

As a good practice, it is *recommended* that when you write tests using the `$httpBackend` service, you add an `afterEach` block with the two function calls in the previous code snippet:

- The first function, `verifyNoOutstandingExpectations()`, checks whether you specified any expects on the `$httpBackend` that were not satisfied as part of your test. So if you added another expectation but the controller never made that server call, your test fails. This adds a good check to ensure that everything that you expected actually happened
- The second function, `verifyNoOutstandingRequests()`, is to ensure that you fully tested all the cases. As mentioned earlier, AngularJS splits all server requests into a request and a response. And we trigger the responses using the `flush()` function. `verifyNoOutstandingRequests` ensures that for each server call made, the response has also been triggered using `flush()`. If not, the test fails.

Integration-Level Unit Tests

What if we followed the best practices, and didn't have our `$http` calls right in our controller? Instead, we had our `$http` calls in a `NoteService`, and our controller delegated the `NoteService` to fetch the list of notes.

In such a case, we have two options:

- Option 1 is to write a focused unit test and mock out (or spy on) the `NoteService`, and ensure that our controller delegates to the correct APIs on the `NoteService` and that the flow and arguments are correct.
- Option 2 is to write an integration-level unit test that only focuses on mocking out the backend (using `$httpBackend`) and checking the entire flow.

Let's try our hand at Option 2 with the following code snippet:

```
// File: chapter7/serverAppWithService.js

angular.module('serverApp2', [])
  .controller('MainCtrl', ['NoteService', function(NoteService) {
    var self = this;
    self.items = [];
    self.errorMessage = '';

    NoteService.query().then(function(response) {
      self.items = response.data;
    }, function(errResponse) {
      self.errorMessage = errResponse.data.msg;
    });
  }])
  .factory('NoteService', ['$http', function($http) {
    return {
      query: function() {
        return $http.get('/api/note');
      }
    };
  }]);
```

This example is almost the same as the one in the previous section, except that the `$http` call has been extracted into the `NoteService` service. Functionally, it behaves exactly the same way.

Now let's look at how we might test this, while also getting an idea for the error condition test:

```
// File: chapter7/serverAppWithServiceSpec.js
describe('Server App Integration', function() {
  beforeEach(module('serverApp2'));

  var ctrl, mockBackend;

  beforeEach(inject(function($controller, $httpBackend) {

    mockBackend = $httpBackend;
    mockBackend.expectGET('/api/note')
        .respond(404, {msg: 'Not Found'});
    ctrl = $controller('MainCtrl');
    // At this point, a server request will have been made
  }));

  it('should handle error while loading items', function() {
    // Initially, before the server responds,
    // the items should be empty
    expect(ctrl.items).toEqual([]);

    // Simulate a server response
```

```
    mockBackend.flush();

    // No items from server, only an error
    // So items should still be empty
    expect(ctrl.items).toEqual([]);
    // and check the error message
    expect(ctrl.errorMessage).toEqual('Not Found');
  });

  afterEach(function() {
    // Ensure that all expects set on the $httpBackend
    // were actually called
    mockBackend.verifyNoOutstandingExpectation();

    // Ensure that all requests to the server
    // have actually responded (using flush())
    mockBackend.verifyNoOutstandingRequest();
  });
});
```

In this test, very little has changed even though our code has added a new service and extracted it out. For the unit test, we are only ensuring that when the controller loads, it makes a server call to */api/notes*. We don't care whether it is through `NoteService` or directly. This makes it much more of an integration test, where it is independent of the underlying implementation.

Also, we are now changing the server to respond with a 404. Under such a condition, we expect the `items` array to still be empty, but now the `errorMessage` variable should be updated in the controller with the server's response. We added an `expect` to make sure this happens. The `afterEach` blocks remains as it was.

Conclusion

We expanded our understanding of unit testing controllers, and walked through how to unit test controllers that depend on built-in AngularJS services (like `$location` and `$window`). After that, we created our own services and learned how to unit test those as well. In both cases, it was as simple as asking for the service to be injected into our test, and then interacting with that as need be.

We then covered unit testing XHRs, using the mocked-out `$httpBackend` service that the *angular-mocks.js* file provides. We saw how to set expectations, and handle the asynchronous behavior of the server calls using the `flush()` method. We also covered how to handle and test both the error and the success cases.

In the next chapter, we will look at AngularJS filters. We will see how to apply common built-in AngularJS filters, as well as create new filters to perform our own formatting tasks.

CHAPTER 8

Working with Filters

In the previous few chapters, we have explored two of the four cornerstones of AngularJS applications: controllers and services. With controllers, we looked at how to get the data we want out into the UI, and how to handle simple styling and presentation logic. We used services to create common business logic, and a layer that would be common across all our controllers.

In this chapter, we work with AngularJS filters. By the end of the chapter, we will get a sense of how and when to use AngularJS filters, as well as how to create a very simple but useful custom AngularJS filter. We end the chapter with a section on best practices and how to get the most out of AngularJS filters.

What Are AngularJS Filters?

AngularJS filters are used to process data and format values to present to the user. They are applied on expressions in our HTML, or directly on data in our controllers and services. Mostly, they are used as that final level of formatting to convert data from the way it is stored to a user-readable format. Some common examples where we would use filters are to take a timestamp and make it human-readable, or to add the currency symbol to a number.

Another feature of AngularJS filters, when they are used in the view, is that they give us dynamic, on-the-fly data that doesn't need to be stored. When we apply filters in the HTML, the filtered values are shown to the user but do not modify the original value on which they are applied.

Let's look at some common AngularJS filters that come with the core AngularJS codebase and how we might use them under various scenarios.

Using AngularJS Filters

AngularJS has some built-in filters to work with dates, numbers, strings, and arrays. A common use case for filters is to use them directly in the view as a last level for formatting of the data that the user sees. Let's look at the example to see some common filters in action:

```
<!-- File: chapter8/filter-example-1.html -->
<html>
<head>
  <title>Filters in Action</title>
</head>
<body ng-app="filtersApp">

<div ng-controller="FilterCtrl as ctrl">
  <div>
    Amount as a number: {{ctrl.amount | number}}
  </div>
  <div>
    Total Cost as a currency: {{ctrl.totalCost | currency}}
  </div>
  <div>
    Total Cost in INR: {{ctrl.totalCost | currency:'INR '}}
  </div>
  <div>
    Shouting the name: {{ctrl.name | uppercase}}
  </div>
  <div>
    Whispering the name: {{ctrl.name | lowercase}}
  </div>
  <div>
    Start Time: {{ctrl.startTime | date:'medium'}}
  </div>
</div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>

<script type="text/javascript">
  angular.module('filtersApp', [])
    .controller('FilterCtrl', [function() {
      this.amount = 1024;
      this.totalCost = 4906;
      this.name = 'Shyam Seshadri';
      this.startTime = new Date().getTime();
    }]);
</script>
</body>
</html>
```

This example code translates into Figure 8-1.

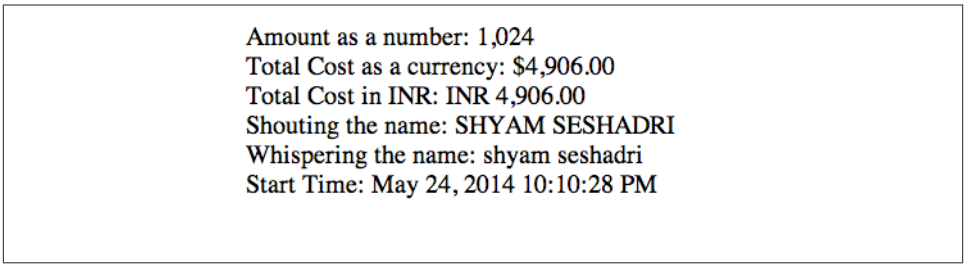

Figure 8-1. AngularJS filter output

We will go over the AngularJS filters one by one, but before that, let's talk about their usage and syntax. The general syntax to use filters is to use the Unix syntax of piping the result of one expression to another. That is:

```
{{expression | filter}}
```

The filter will take the value of the expression (a string, number, or array) and convert it into some other form. For example, the `currency` filter used in the previous code snippet takes the `totalCost` number and converts it into a string, with commas, decimals, and the currency symbol added. The `uppercase` and `lowercase` filters take the string name and convert it into uppercase and lowercase, respectively.

Do note that when we say something like:

```
{{ctrl.name | lowercase}}
```

we are formatting data on the fly. This means that the value of `ctrl.name` does not change, whereas the user still sees the final lowercase result.

We can also chain multiple filters together by piping one filter after another. The syntax would be:

```
{{expression | filter1 | filter2}}
```

For example, say we want to take our `name` variable from the previous controller, convert it to lowercase, and display only the first five letters. We could accomplish that like this:

```
{{ctrl.name | lowercase | limitTo:5}}
```

Each filter takes the value from the previous expression and applies its logic on it. In this example, the name would first be lowercased, and then the lowercase name would be provided to the `limitTo` filter. The `limitTo` filter would just return the first five characters from the string, thus returning "shyam" to the HTML. As you can see, we can pass arguments to filters as well, which we use to tell the `limitTo` filter how many characters to limit the string to.

Common AngularJS Filters

Let's go over each of the filters we mentioned in passing, as well as some additional ones with some examples to see how and when to use each one. We'll talk about the available filters before giving a comprehensive example that demos all of them in a single app:

`currency`
: The `currency` filter formats a given number as currency with the commas, decimals, and currency symbol added as needed. The filter takes an optional currency symbol as the second argument; if none exists, it takes the default symbol for the current browser.

`number`
: The `number` filter takes a number and converts it to a human-readable string with comma separation. The number filter also takes an optional decimal size that tells it how many digits to keep after the decimal point.

`lowercase`
: A very simple string filter that takes any string and converts all the characters to lowercase.

`uppercase`
: A very simple string filter that takes any string and converts all the characters to uppercase.

`json`
: The `json` filter is a great tool for debugging, or for any time we need to display the contents of a JSON object or an array in the UI. It takes a JSON object or array (or even primitives) and displays it as a string in the UI.

`date`
: The `date` filter is a customizable and powerful filter that takes a date object or a long timestamp and displays it as a human-readable string in the UI. It can take a user-defined format or one of the built-in *short*, *medium*, or *long* formats. The detailed documentation for various formatting options is available in the Date Filter API page.

The following example demonstrates these filters used in combination with strings and numbers:

```
<!-- File: chapter8/filter-number-string.html -->
<html>
<head>
  <title>Filters in Action</title>
</head>
<body ng-app="filtersApp">

<ul ng-controller="FilterCtrl as ctrl">
```

```
    <li>
      Amount - {{ctrl.amount}}
    </li>
    <li>
      Amount - Default Currency: {{ctrl.amount | currency}}
    </li>
    <li>
      <!-- Using the English pound sign -->
      Amount - INR Currency: {{ctrl.amount | currency:'&#163 '}}
    </li>

    <li>
      Amount - Number: {{ctrl.amount | number}}
    </li>
    <li>
      Amount - No. with 4 decimals: {{ctrl.amount | number:4}}
    </li>

    <li>
      Name with no filters: {{ctrl.name}}
    </li>
    <li>
      Name - lowercase filter: {{ctrl.name | lowercase}}
    </li>
    <li>
      Name - uppercase filter: {{ctrl.name | uppercase}}
    </li>

    <li>
      The JSON Filter: {{ctrl.obj | json}}
    </li>

    <li>
      Timestamp: {{ctrl.startTime}}
    </li>
    <li>
      Default Date filter: {{ctrl.startTime | date}}
    </li>
    <li>
      Medium Date filter: {{ctrl.startTime | date:'medium'}}
    </li>
    <li>
      Custom Date filter: {{ctrl.startTime | date:'M/dd, yyyy'}}
    </li>
  </ul>

  <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
  </script>

  <script type="text/javascript">
    angular.module('filtersApp', [])
```

```
    .controller('FilterCtrl', [function() {
      this.amount = 1024;
      this.name = 'Shyam Seshadri';
      this.obj = {test: 'value', num: 123};
      this.startTime = new Date().getTime();
    }]);
</script>
</body>
</html>
```

In this code snippet, four values are defined in the controller:

- A number, `amount`
- A string, `name`
- A JSON object, `obj`
- A timestamp, `startTime`

Then, in the HTML (shown in Figure 8-2) we have the following bindings and filters, in order:

1. The `amount` variable itself.
2. The `amount` variable, using the default `currency` filter. This uses the current browser to grab the currency symbol.
3. The `amount` variable using the `currency` filter with a defined symbol (in this case, the English pound sign).
4. The `amount` variable using the default `number` filter, which is like the `currency` filter but without decimals and the currency symbol by default.
5. The `amount` variable using the `number` filter, and forcing four digits after the decimal point.
6. The `name` variable itself.
7. The `name` variable using the `lowercase` filter, which ensures that the output becomes `"shyam seshadri"`.
8. The `name` variable using the `uppercase` filter, which ensures that the output becomes `"SHYAM SESHADRI"`.
9. The `obj` variable, printed as a string `{"test": "value", "num": 123}`
10. The `timestamp` used for the date filters.
11. The `timestamp` used with the default `date` filter (which prints something like "Jan 3, 2007": the medium date format).
12. The `timestamp` with the medium `date` filter, which prints the medium date in the filter, along with the time (something like "Jan 3, 2007 12:04:45 pm").

13. The `timestamp` with a custom `date` filter specified using a date format (something like 1/23, 2014).

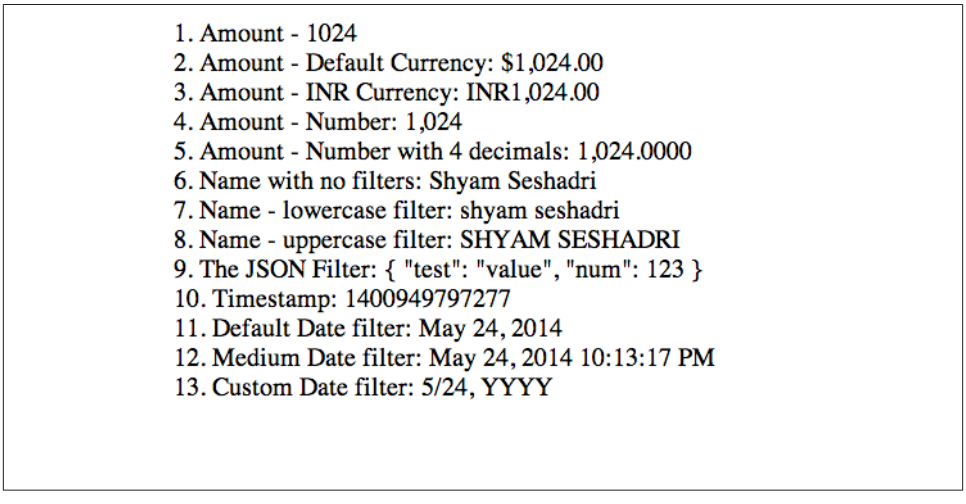

Figure 8-2. Screenshot of number and string filters

Next, let's look at the filters that work mostly with arrays and give ways to slice and dice and change the order as per our needs:

`limitTo`
: The `limitTo` is a simple AngularJS filter that takes either a string (as we saw in the example on chaining) or an array and returns a subset from the beginning or the end of the array, depending on the argument passed to it. If the `limitTo` is given only a number (say, 3), it returns only that many elements from the array or characters from the string (in this case, 3 again). If it is a negative number, it picks up those elements or characters from the end of the array.

`orderBy`
: One of the two more complicated filters (the other being `filter`, which we cover next), `orderBy` allows us to take an array and order it by a predicate expression (or a series of predicate expressons). It also takes a second optional Boolean argument, which decides whether or not the sorted array is reversed. The simplest form of a predicate expression is a string, which is the name of the field (the key of each object) to order the array by, with an optional + or – sign before the field name to decide whether to sort ascending or descending by the field. The predicate expression can also be passed a function, in which case the return value of the function will be used (with simple <, >, = comparisons) to decide the order. Finally, the predicate expression can be an array, in which case each element of the array is either a string

or a function. AngularJS will then sort it by the first element of the array, and keep cascading to the next element if it is equal.

`filter`

By far one of the most flexible filters in AngularJS is `filter` (confused yet?). While named slightly confusingly, the `filter` filter in AngularJS is used to filter an array based on predicates or functions, and decide which elements of an array are included. This filter is most commonly used along with the `ng-repeat` to do dynamic filtering of an array. The expression to filter the array can be one of the following:

`string`

If provided a string expression, AngularJS will look for the string in the keys of each object of the array, and if it is found, the element is included. The string can optionally be prefixed with an `!` to negate the match.

`object`

A pattern object can also be provided, in which case AngularJS takes each key of the object and makes sure that its value is present in the corresponding key of each object of the array. For example, an object expression like `{size: "M"}` would check each item of the array and ensure that the objects have a key called `size` and that they contain the letter "M" (not necessarily an exact match).

`function`

The most flexible and powerful of the options, the filter can take a function to implement arbitrary and custom filters. The function gets called with each item of the array, and uses the return value of the function to decide whether to include the item in the end result. Any item that returns a false gets dropped from the result.

Let's use an example to demonstrate how these might work:

```
<!-- File: chapter8/filter-arrays.html -->
<html>
<head>
  <title>Filters in Action</title>
</head>
<body ng-app="filtersApp">

<div ng-controller="FilterCtrl as ctrl">

  <button ng-click="ctrl.currentFilter = 'string'">
    Filter with String
  </button>
  <button ng-click="ctrl.currentFilter = 'object'">
    Filter with Object
  </button>
  <button ng-click="ctrl.currentFilter = 'function'">
```

```
    Filter with Function
  </button>
  Filter Text
  <input type="text"
         ng-model="ctrl.filterOptions['string']">
  Show Done Only
  <input type="checkbox"
         ng-model="ctrl.filterOptions['object'].done">
  <ul>
    <li ng-repeat="note in ctrl.notes |
                   filter:ctrl.filterOptions[ctrl.currentFilter] |
                   orderBy:ctrl.sortOrder |
                   limitTo:5">
       {{note.label}} - {{note.type}} - {{note.done}}
    </li>
  </ul>
</div>

<script
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
</script>

<script type="text/javascript">
  angular.module('filtersApp', [])
    .controller('FilterCtrl', [function() {
      this.notes = [
        {label: 'FC Todo', type: 'chore', done: false},
        {label: 'FT Todo', type: 'task', done: false},
        {label: 'FF Todo', type: 'fun', done: true},
        {label: 'SC Todo', type: 'chore', done: false},
        {label: 'ST Todo', type: 'task', done: true},
        {label: 'SF Todo', type: 'fun', done: true},
        {label: 'TC Todo', type: 'chore', done: false},
        {label: 'TT Todo', type: 'task', done: false},
        {label: 'TF Todo', type: 'fun', done: false}
      ];
      this.sortOrder = ['+type', '-label'];

      this.filterOptions = {
        "string": '',
        "object": {done: false, label: 'C'},
        "function": function(note) {
          return note.type === 'task' && note.done === false;
        }
      };

      this.currentFilter = 'string';
    }]);
</script>
</body>
</html>
```

In this example, we are using all the array-related filters in one example. We have an array of `notes`, with three keys (`label`, `type`, and `done`). We also define an array of sorting predicates, `sortOrder`, to tell AngularJS to first sort by the `type`, and if they are equal, to sort in reverse by `label`.

In the HTML, we are sorting by `sortOrder` and limiting the results to five elements. Before either of these, we are filtering the array by our `filterOptions`. By default, we are using the `string` filter, which is bound to the text box. If we type in anything, it will match against all the fields in each note and display only the items that match. Clicking the buttons will switch our filtering mode from `string` to `object` or `function`.

The `object` filter shows all `notes` that are not `done`, and which have the character `C` in the `label`. The checkbox in the UI allows us to toggle the `done` filter to show only the done notes, or only the notes that are not done.

The `function` filter only shows notes that are tasks and not done.

In a real application, we could bind checkboxes, select boxes, and text boxes to various fields of an object (using `ng-model`) and use the object to filter the list dynamically. The function filter could be expanded and made more complex based on the business logic. The beauty of the `filter` filter is that it is dynamic when used in the HTML directly, so the minute the underlying model changes, the entire list gets filtered automatically.

Using Filters in Controllers and Services

Filters are well and great in the HTML and UI, but what if we need to apply these transformations to our controller or service? Thankfully, AngularJS allows us to use the filters wherever we want or need through the power of Dependency Injection. So without ever needing to access the DOM or the UI, we can use the business logic of the filter right in our JavaScript code.

Any filter, whether built-in or our own, can be injected into any service or controller by affixing the word "Filter" at the end of the name of the filter, and asking it to be injected. For example, if we need the `currency` filter in our controller, we can do something like this:

```
angular.module('myModule', [])
    .controller('MyCtrl', ['currencyFilter',
        function(currencyFilter) {
}]);
```

Similarly, the `number` filter becomes `numberFilter` and `filter` of course becomes the convoluted `filterFilter`. Attaching the word "Filter" after any AngularJS filter allows us to inject it into our controllers or services.

Now, in the HTML, we use the pipe syntax to give it some input to work with, followed by optional arguments. When we get a handle on it in our controller or service, we get

a function. The first argument to the function is the value the filter needs to act upon: a string, number, or array. All additional parameters are the arguments we mentioned earlier, in order as needed.

So if we wanted to filter our array `self.notes` using the `filterFilter` with just a string, we could do something like:

```
self.filteredArray = filterFilter(self.notes, 'ch');
```

There are three main things to note:

- The first argument to the filter is the value it needs to act upon.
- Further arguments are the arguments that the filter needs (optional for some), in the order mentioned in the documentation.
- The return value of the filter is the final output that we need.

Creating AngularJS Filters

We saw how we can use some of the existing built-in AngularJS filters, but what if they are not enough? The date filter might be good, but we want our own formatting and behavior. We might want a filter for localization. In this section, we will go about creating our own filter.

We are going to write a very simple filter called `timeAgo`. In production, we might use something like MomentJS, but here, we are going to craft a very simplistic one ourselves. All we want to do is take a timestamp and display in the UI messages like "seconds ago," "minutes ago," "days ago," and "months ago." We are displaying only the message, without any numbers. So regardless of whether it was 5 or 15 seconds ago, the message would read "seconds ago." How might we go about doing this?

```
<!-- File: chapter8/custom-filters.html -->
<html>
<head>
  <title>Custom Filters in Action</title>
</head>
<body ng-app="filtersApp">

<div ng-controller="FilterCtrl as ctrl">
  <div>
    Start Time (Timestamp): {{ctrl.startTime}}
  </div>
  <div>
    Start Time (DateTime): {{ctrl.startTime | date:'medium'}}
  </div>
  <div>
    Start Time (Our filter): {{ctrl.startTime | timeAgo}}
  </div>
```

```
    <div>
      someTimeAgo : {{ctrl.someTimeAgo | date:'short'}}
    </div>
    <div>
      someTimeAgo (Our filter): {{ctrl.someTimeAgo | timeAgo}}
    </div>
  </div>

  <script
    src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.19/angular.js">
  </script>

  <script type="text/javascript">
    angular.module('filtersApp', [])
      .controller('FilterCtrl', [function() {
        this.startTime = new Date().getTime();
        this.someTimeAgo = new Date().getTime() -
            (1000 * 60 * 60 * 4);
      }])
      .filter('timeAgo', [function() {
        var ONE_MINUTE = 1000 * 60;
        var ONE_HOUR = ONE_MINUTE * 60;
        var ONE_DAY = ONE_HOUR * 24;
        var ONE_MONTH = ONE_DAY * 30;

        return function(ts) {
          var currentTime = new Date().getTime();
          var diff = currentTime - ts;
          if (diff < ONE_MINUTE) {
            return 'seconds ago';
          } else if (diff < ONE_HOUR) {
            return 'minutes ago';
          } else if (diff < ONE_DAY) {
            return 'hours ago';
          } else if (diff < ONE_MONTH) {
            return 'days ago';
          } else {
            return 'months ago';
          }
        };
      }]);
  </script>
  </body>
  </html>
```

In this example, we defined our own custom filter called `timeAgo`. We define a filter in a very similar manner to controllers and services, and we can also inject any services that our filter might depend on into it.

Every filter returns a function, which is what gets called for every usage of the filter. This function gets called with the value that the filter is being applied on. In this case, it is

the `timestamp` as a number. The filter can then act upon this value, and slice and dice it in whichever way it wants. We just take the difference of the timestamp and the current time, and then return a string based on the difference.

If we want to take optional arguments, like the `currency` filter or the `number` filter, we just have to add them as additional parameters to the function we return. Suppose we want to take a Boolean to say only show minutes, or `ignoreSeconds`, then we could change the return to something like this:

```
return function(ts, ignoreSeconds) {
```

This would then be passed in from the HTML as follows, with the optional argument set to `true`:

```
{{ctrl.startTime | timeAgo:true}}
```

If you need multiple arguments, keep adding them and pass them in the same order in the HTML:

```
return function(ts, arg1, arg2, arg3) {
```

and:

```
{{ctrl.startTime | timeAgo:arg1:arg2:arg3}}
```

We will see how to test these filters in Chapter 9.

Things to Remember About Filters

We saw how to use existing filters in AngularJS, as well as how to customize them and pass them arguments. We then saw how we could use them in controllers and services, before finally creating our own custom filter. Now in this section, we cover some best practices and things to remember about working with filters in AngularJS:

View filters are executed every digest cycle
: The first and foremost thing we should know and remember about AngularJS filters is that if we are using them directly in the view (which is quite often), they are re-evaluated every time a digest cycle happens (we cover this in depth in "The Digest Cycle" on page 206). Therefore, as the data we are working on grows, we need to be aware of the extra computation we might be performing when we extensively use filters across our UI.

Filters should be blazingly fast
: Because of the previous point, whenever we write our own filters, we should write them in such a way that makes their execution blazingly fast. Ideally, our filter functions should be capable of executing multiple times in milliseconds. So things like DOM manipulation, asynchronous calls, and other slow activities should be avoided when we write filters.