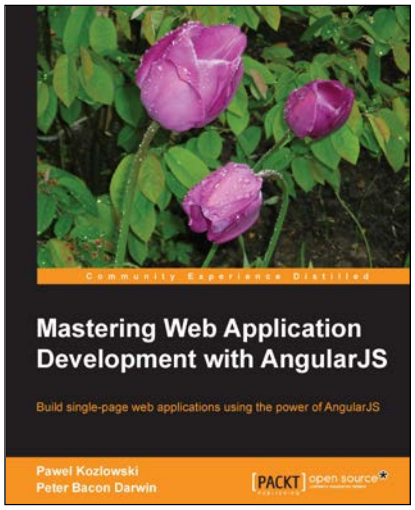
Mastering Web Application Development with AngularJS

Pawel Kozlowski

Peter Bacon Darwin



Chapter No. 5
"Creating Advanced Forms"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.5 "Creating Advanced Forms"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Pawel Kozlowski has over 15 years of professional experience in web development and was fortunate enough to work with variety of web technologies, languages, and platforms. He is not afraid of hacking both at client side and server side and always searches for the most productive tools and processes.

Pawel strongly believes in free, open source software. He has been very committed in the AngularJS project and also is very active in the AngularJS community. He also contributes to Angular UI – the companion suite to the AngularJS framework, where he focuses on the Twitter's Bootstrap directives for AngularJS.

When not coding, Pawel spreads a good word about AngularJS at various conferences and meetups.

Peter Bacon Darwin has been programming for over two decades. He worked with .NET from before it was released; he contributed to the development of IronRuby and was an IT consultant for Avanade and IMGROUP before quitting to share his time between freelance development and looking after his kids.

Peter is a notable figure in the AngularJS community. He has recently joined the AngularJS team at Google as an external contractor and is a founder member of the AngularUI project. He has spoken about AngularJS at Devoxx UK and numerous London meetups. He also runs training courses in AngularJS. His consultancy practice is now primarily focused on helping businesses make best use of AngularJS.

I would like to thank the team at Google for giving us AngularJS, in particular the ones who got it all going: Miško Hevery, Igor Minar, Brad Green, and Vojta Jína. They are a constant inspiration. My coauthor, Pawel, is the driving force behind this book. He conceived the structure, wrote most of the content, and is a great guy to work with. We have all benefited from the awesome active community that has built up around AngularJS in such a short time, especially the folks in the AngularUI project. Finally, I couldn't have completed the book without the love and support of my wife, Kelyn, and kids, Lily and Zachary.

Mastering Web Application Development with AngularJS

AngularJS is a relatively new JavaScript MVC framework but it is the real game changer. It has a novel approach to templating and bi-directional data binding which makes the framework very powerful and easy to use. People constantly report a dramatic reduction in the number of lines of code needed in applications using AngularJS as compared to other approaches.

AngularJS is an outstanding piece of engineering. With its strong emphasis on testing and code quality it promotes good practices for the entire JavaScript ecosystem. Given the quality and novelty of the technology, it is not surprising to see that many people are attracted to the framework, creating a very vibrant and supportive community around AngularJS, which contributes to its growing popularity.

As AngularJS becomes more and more popular, people will start to use it in complex projects. But you will soon face problems that are not solved in the standard documentation or in the simple examples found on the Web. AngularJS, as any other technology, has its own set of idioms, patterns, and best practices that have been uncovered by the community, based on their collective experiences.

This is where this book comes in – it aims to show how to write non-trivial AngularJS applications in a canonical way. Instead of describing how the framework works, this book focuses on how to use AngularJS to write a complex web application. It provides real answers to real questions being asked by the AngularJS community.

In short, this is a book written for application developers, by application developers, and based on real developers questions. In this book you will learn:

- How to build a complete, robust application using existing AngularJS services and directives.
- How to extend AngularJS (directives, services, filters) when there is no out-of-the-box solution
- How to set up a high quality AngularJS development project (code organization, build, testing, performance tuning)

What This Book Covers

Chapter 1, Angular Zen, serves as an introduction to AngularJS framework and the project. The first chapter outlines project's philosophy, its main concepts, and basic building blocks.

Chapter 2, Building and Testing, lays a foundation for a sample application used in this book. It introduces problem domain and covers topics such as testing and building best practices for the system.

Chapter 3, Communicating with a Back-end Server, teaches us how to fetch data from a remote back-end and feed those data effectively to the UI powered by AngularJS. This chapter has extensive coverage of the promise API.

Chapter 4, Displaying and Formatting Data, assumes that data to be displayed were already fetched from back-end and shows how to render those data in the UI. This chapter discusses the usage of AngularJS directives for UI rendering and filters for data formatting.

Chapter 5, Creating Advanced Forms, illustrates how to allow users to manipulate data through forms and various types of input fields. It covers various input types supported by AngularJS and contains deep dive into forms validation.

Chapter 6, Organizing Navigation, shows how to organize individual screens into an easy-to-navigate application. It starts by explaining role of URLs in single-page web applications and familiarizes a reader with key AngularJS services for managing URLs and navigation.

Chapter 7, Securing Your Application, goes into the details of securing single-page web applications written using AngularJS. It covers the concepts and techniques behind authenticating and authorizing users.

Chapter 8, Building Your Own Directives, serves as an introduction to one of the most exciting parts of the AngularJS: directives. It will guide the reader through a structure of sample directives as well as demonstrate testing approaches.

Chapter 9, Building Advanced Directives, is based on Chapter 8, Building Your Own Directives and covers more advanced topics. It is filled with a real-life directive examples clearly illustrating complex techniques.

Chapter 10, Building AngularJS Web Applications for an International Audience, deals with internationalization of AngularJS applications. Covered topics include approaches to translating templates as well as managing locale-dependent settings.

Chapter 11, Writing Robust AngularJS Web Applications focuses on non-functional, performance requirements for web applications. It peeks under the hood of AngularJS to familiarize readers with its performance characteristics. A good understanding of AngularJS internals will allow us to avoid common performance-related pitfalls.

Chapter 12, Packaging and Deploying AngularJS Web Applications will guide you through a process of preparing a finished application for production deployment. It illustrates how to optimize application load with a special focus on the landing page.

5
Creating Advanced Forms

AngularJS builds upon standard HTML forms and input elements. This means that you can continue to create your UI from the same HTML elements that you already understand using standard HTML design tools.

So far, in our SCRUM application, we've created some basic forms with input elements bound to model data and buttons for saving and deleting and so on. AngularJS takes care of wiring up both the element-model binding and the event-handler binding.

In this chapter, we will look at how AngularJS forms work in detail and then add validation and dynamic user interaction to our application's forms.

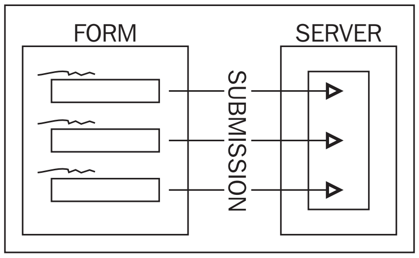
In this chapter we will cover:

- Model data binding and input directives
- Form Validation
- Nested and Repeated Forms
- Form Submission
- Resetting a Form

Comparing traditional forms with AngularJS forms

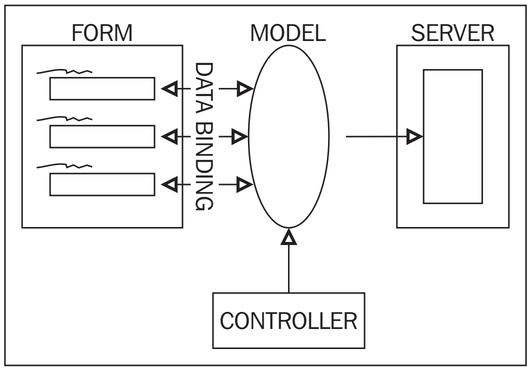
Before we begin to improve our application's forms we should understand how AngularJS forms work. In this section, we explain the differences between standard HTML input elements and AngularJS input directives. We will show how AngularJS modifies and expands the behavior of HTML input elements and how it manages updates to and from the model for you.

In a standard HTML form, the value of an input element is the value that will be sent to the server on submission of the form.



The trouble with the input elements holding the value of your submission is that you are stuck with having to work with the input values as they are shown to the user. This is often not what you want. For instance, a date input field may allow the user to enter a string in some predefined format, for example, "12 March 2013". But in your code you might want to work with the value as a JavaScript `Date` object. Constantly coding up these transformations is tedious and error-prone.

AngularJS decouples the model from the view. You let input directives worry about displaying the values and AngularJS worry about updating your model when the values change. This leaves you free to work with the model, via controllers for instance, without worrying about how the data is displayed or entered.



To achieve this separation, AngularJS enhances HTML forms with the `form` and `input` directives, validation directives, and controllers. These directives and controllers override the built-in behavior of HTML forms but, to the casual observer, forms in AngularJS look very similar to standard HTML forms.

First of all, the `ngModel` directive lets you define how inputs should bind to the model.

Introducing the ngModel directive

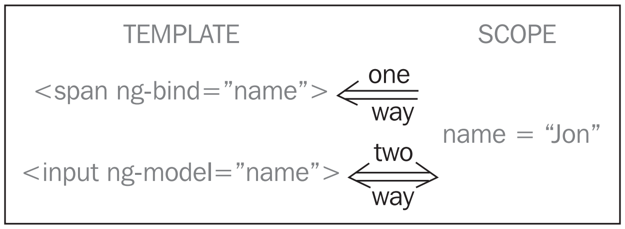
We have already seen how AngularJS creates data binding between fields on the `scope` object and HTML elements on the page. One can set up data binding by using curly braces, `{{}}`, or directives such as `ngBind`. But using such bindings is only one way. When binding to the value of an input directive, we use `ngModel`:

```
<div>Hello <span ng-bind="name"/></div>
<div>Hello <input ng-model="name"/></div>
```

Try it at `http://bit.ly/Zm55zM`.

In the first `div`, AngularJS binds `scope.name` of the current scope to the text of the `span`. This binding is one way: if the value of `scope.name` changes, the text of the `span` changes; but if we change the text of the `span`, the value of `scope.name` will not change.

In the second `div`, AngularJS binds `scope.name` of the current scope to the value of the `input` element. Here the data binding really is two way, since if we modify the value of the input box by typing in it, the value of the `scope.name` model is instantly updated. This update to `scope.name` is then seen in the one way binding to the `span`.



Why do we have a different directive to specify the binding on inputs? The `ngBind` directive only binds (one way) the value of its expression to the text of the element. With `ngModel`, data binding is two way, so changes to the value of the input are reflected back in the model.

In addition, AngularJS allows directives to transform and validate the `ngModel` directive values as the data binding synchronizes between the model and the input directive. You will see how this works in the `ngModelController` section.

Creating a User Information Form

In this section we will describe a simple User Information Form from our example SCRUM application. Throughout this chapter we will incrementally add functionality to this form to demonstrate the power of AngularJS forms. Here is our basic working form:

```
<h1>User Info</h1>
<label>E-mail</label>
<input type="email" ng-model="user.email">

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

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

<label>Website</label>
<input type="url" ng-model="user.website">

<label>Description</label>
<textarea ng-model="user.description"></textarea>

<label>Password</label>
<input type="password" ng-model="user.password">

<label>Password (repeat)</label>
<input type="password" ng-model="repeatPassword">

<label>Roles</label>
<label class="checkbox">
  <input type="checkbox" ng-model="user.admin"> Is Administrator
</label>

<pre ng-bind="user | json"></pre>
```

Try it at `http://bit.ly/10ZomqS`.

While it appears that we simply have a list of standard HTML inputs, these are actually AngularJS `input` directives. Each `input` has been given an `ngModel` directive that defines what current scope to bind the value of the `input` element. In this case, each `input` is bound to a field on the `user` object, which itself is attached to the current scope. In a controller we could log the value of a model field like so:

```
$log($scope.user.firstName);
```

Notice that we have not used a `form` element or put `name` or `id` attributes on any of the `input` elements. For simple forms with no validation this is all we need. AngularJS ensures that the values of the input elements are synchronized with the values in the model. We are then free to work with the user model in a controller, for example, without worrying about how they are represented in the view.

We have also bound a `pre` element with a JSON representation of the user model. This is so that we can see what is being synchronized to the model by AngularJS.

Understanding the input directives

In this section we describe the AngularJS input directives that are provided out of the box. Using input directives is very natural to people used to HTML forms because AngularJS builds on top of HTML.

You can use all the standard HTML input types in your forms. The input directives work with the `ngModel` directive to support additional functionality, such as validation or binding to the model. The AngularJS `input` directive checks the `type` attribute to identify what kind of functionality to add to the input element.

Adding the required validation

All the basic input directives support the use of the `required` (or `ngRequired`) attribute. Adding this attribute to your input element will tell AngularJS that the `ngModel` value is invalid if it is `null`, `undefined`, or `""` (the empty string). See the following section on Field Validation for more about this.

Using text-based inputs (text, textarea, e-mail, URL, number)

The basic input directive, `type="text"` or `textarea`, accepts any string for its value. When you change the text in the input, the model is instantly updated with the value.

The other text-based input directives, such as e-mail, URL, or number, act similarly except that they only allow the model to update if the value in the input box matches an appropriate regular expression. If you type into the e-mail input, the e-mail field in the model is blank until the input box contains a valid e-mail string. This means that your model never contains an invalid e-mail address. This is one of the benefits of decoupling the model from the view.

In addition to these validations all the text-based directives allow you to specify minimum and maximum lengths for the text as well as an arbitrary regular expression that must match. This is done with the `ngMinLength`, `ngMaxLength`, and `ngPattern` directives:

```
<input type="password" ng-model="user.password"
  ng-minlength="3" ng-maxlength="10"
  ng-pattern="/^.*(?=.*\d)(?=.*[a-zA-Z]).*$/">
```

Try it at `http://bit.ly/153L87Q`.

Here, the `user.password` model field must have between 3 and 10 characters, inclusive, and it must match a regular expression that requires it to include at least one letter and one number.

Note that these built-in validation features do not stop the user from entering an invalid string. The input directive just clears the model field if the string is invalid.

Using checkbox inputs

Checkboxes simply indicate a boolean input. In our form, the input directive assigns `true` or `false` to the model field that is specified by `ngModel`. You can see this happening in our User Info Form for the "Is Administrator" field.

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

The value of `user.admin` is set to `true` if the checkbox is checked and `false` otherwise. Conversely, the checkbox will be ticked if the value of `user.admin` is `true`.

You can also specify different strings for `true` and `false` values to be used in the model. For example, we could have used `admin` and `basic` in a role field.

```
<input type="checkbox" ng-model="user.role" ng-true-value="admin" ng-
false-value="basic">
```

Try it at `http://bit.ly/Yidt37`.

In this case, the `user.role` model, would contain either `admin` or `basic` depending on whether the checkbox was ticked or not.

Using radio inputs

Radio buttons provide a fixed group of choices for a field. AngularJS makes this really simple to implement: Just bind all the radio buttons in a group to the same model field. The standard HTML `value` attribute is then used to specify what value to put in the model when the radio is selected:

```
<label><input type="radio" ng-model="user.sex" value="male"> Male</
label>
<label><input type="radio" ng-model="user.sex" value="female">
Female</label>
```

Try it at `http://bit.ly/14hYNsN`.

Using select inputs

The `select` input directive allows you to create a drop-down list, from which the user can select one or more items. AngularJS lets you specify options for the drop down statically or from an array on the scope.

Providing simple string options

If you have a static list of options from which to select you can simply provide them as `option` elements below the `select` element:

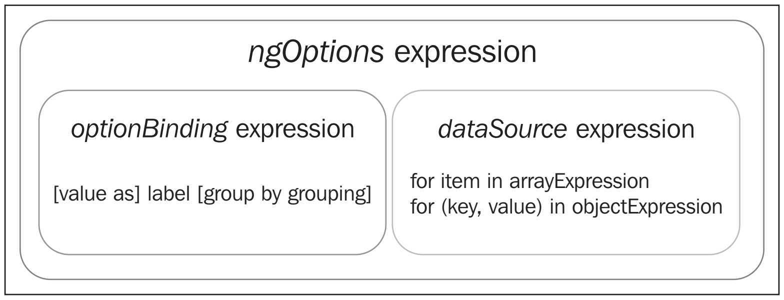
```
<select ng-model="sex">
  <option value="m" ng-selected="sex=='m'">Male</option>
  <option value="f" ng-selected="sex=='f'">Female</option>
</select>
```

Be aware that since the `value` attribute can only take a string, the value to which you bind can only be a string.

> If you want to bind to values that are not strings or you want your list of options to be created dynamically from data, then use `ngOptions` as follows.

Providing dynamic options with the ngOptions directive

AngularJS provides an additional syntax for dynamically defining a complex list of options for a `select` directive. If you want to bind the value of a `select` directive to an object, rather than a simple string, then use `ngOptions`. This attribute accepts a **comprehension expression** that defines what options are to be displayed. The form of this expression is:


ngOptions expression
optionBinding expression
[value as] label [group by grouping]
dataSource expression
for item in arrayExpression
for (key, value) in objectExpression

The `dataSource expression` describes the source of the information about the options to be displayed. It describes elements in an array or properties on an object. One select option will be generated for each item in the `dataSource expression`.

The `optionBinding` expression describes what should be extracted from each data source item and how that item should be bound to the `select` option.

Common examples of ngOptions

Before we explain the details of how to define these **comprehension expressions**, here are some typical examples.

Using array data sources

Select a user object with `user.email` as the label:

```
ng-options="user.email for user in users"
```

Select a user object with a computed label (the function would be defined on the scope):

```
ng-options="getFullName(user) for user in users"
```

Select a user's e-mail rather than the whole user object, with their full name as the label:

```
ng-options="user.email as getFullName(user) for user in users
```

Select a user object with the list grouped by sex:

```
ng-options="getFullName(user) group by user.sex for user in users"
```

Try it at `http://bit.ly/1157jqa`.

Using object data sources

Let's provide two objects that relate country names to codes:

```
$scope.countriesByCode = {
  'AF' : 'AFGHANISTAN',
  'AX' : 'ÅLAND ISLANDS',
  ...
};

$scope.countriesByName = {
  'AFGHANISTAN' : 'AF',
  'ÅLAND ISLANDS' : 'AX',
  ...
};
```

To select a country code by country name, ordered by country code:

```
ng-options="code as name for (code, name) in countriesByCode"
```

To select a country code by country name, ordered by country name:

```
ng-options="code as name for (name, code) in countriesByName"
```

Try it at `http://bit.ly/153LKdE`.

Now that we have seen some examples, we can show the full specification of these expressions.

Understanding the dataSource expression

If the data source will be an array then the `arrayExpression` should evaluate to an array. The directive will iterate over each of the items in the array, assigning the current item in the array to the `value` variable.

> The list of select options will be displayed in the same order as the items appear in the array.

If the data source will be an object then the `objectExpression` should evaluate to an object. The directive will iterate over each property of the object, assigning the value of the property to the `value` variable and the key of the member to the `key` variable.

The list of select options will be ordered alphabetically by the value of the key.

Understanding the optionBinding expression

The `optionBinding` expression defines how to get the label and value for each option and how to group the options from the items provided by the `dataSource expression`. This expression can take advantage of all the AngularJS expression syntax, including the use of filters. The general syntax is:

```
value as label group by grouping
```

If the `value` expression is not provided then the data item itself will be used as the value to assign to the model when this item is selected. If you provide a grouping expression, it should evaluate to the name of the group for the given option.

Using empty options with the select directive

What should the `select` directive do when the bound model value doesn't match with any of the values in the option list? In this case, the `select` directive will show an empty option at the top of the list of options.

The empty option will be selected whenever the model does not match any of the options. If the user manually selects the empty option then the model will be set to `null`. It will not be set to `undefined`.

You can define an empty option by adding an `option` element as a child of the `select` element that has an empty string for its value:

```
<select ng-model="..." ng-options="...">
  <option value="">-- No Selection --</option>
</select>
```

Try it at `http://bit.ly/ZeNpZX`.

Here, we defined an empty option, which will display the `-- No Selection --` label.

If you define your own empty option then it will always be shown in the list of options and can be selected by the user.

If you do not define your own empty option in the declaration of the `select` directive it will generate its own.

[If the directive generates the empty option, it will be shown only when the model does not match any items in the list. So the user will not be able to manually set the `select` value to `null/undefined`.]

It is possible to hide the empty option by defining your own and setting its style to `display: none`.

```
<option style="display:none" value=""></option>
```

Try it at `http://bit.ly/ZeNpZX`.

In this case the `select` directive will use our empty option but the browser will not show it. Now, if the model does not match any options the `select` directive will be blank and invalid but there will not be a blank option shown in the list.

Understanding select and object equivalence

The `select` directive matches the model value to the `option` value using the object equivalence operator (`===`). This means that if your option values are objects and not simply values (like numbers and strings) you must use a reference to the actual option value for your model value. Otherwise the `select` directive will think that the objects are different and will not match it to the option.

In a controller we might set up the options and selected items as an array of objects:

```
app.controller('MainCtrl', function($scope) {
  $scope.sourceList = [
    {'id': '10005', 'name': "Anne"},
    {'id': '10006', 'name': "Brian"},
    {'id': '10007', 'name': "Charlie"}
  ];
  $scope.selectedItemExact = $scope.sourceList[0];
  $scope.selectedItemSimilar = {'id': '10005', 'name': "Anne"};
});
```

Here, `selectedItemExact` actually references the first item in the `sourceList`, while `selectedItemSimilar` is a different object, even though the fields are identical:

```
<select
  ng-model="selectedItemExact"
  ng-options=" item.name for item in sourceList">
</select>
<select
  ng-model="selectedItemSimilar"
  ng-options="item.name for item in sourceList">
</select>
```

Try it at `http://bit.ly/Zrachk`.

Here, we create two `select` directives that are bound to these values. The one bound to `selectedItemSimilar` will not have an option selected. Therefore, you should always bind the value of the select to an item in the `ng-options` array. You may have to search the array for the appropriate option.

Selecting multiple options

If you want to select multiple items, you simply apply the `multiple` attribute to the `select` directive. The `ngModel` bound to this directive is then an array containing a reference to the value of each selected option.

AngularJS provides the `ngMultiple` directive, which takes an expression to decide whether to allow multiple selections. Currently, the `select` directive does not watch changes as to whether it accepts multiple selections, so the `ngMultiple` directive has limited use.

Working with traditional HTML hidden input fields

In AngularJS, we store all our model data in the scope so that there is rarely any need to use hidden input fields. Therefore, AngularJS has no hidden input directive. There are two cases where you might use hidden input fields: embedding values from the server and supporting traditional HTML form submission.

Embedding values from the server

You use a server-side templating engine to create the HTML and you pass data from the server to AngularJS via the template. In this case, it is enough to put an `ng-init` directive into the HTML that is generated by the server, which will add values to the scope:

```
<form ng-init="user.hash='13513516'">
```

Here the HTML sent from the server contains a form element that includes an `ng-init` directive that will initialize `user`, `hash` on the scope of the form.

Submitting a traditional HTML form

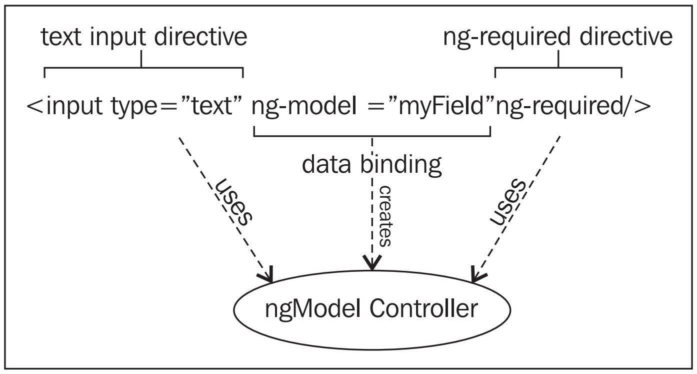
Traditionally, you might have wanted to submit values to the server that are not in the view, that is, not a visible input control. This would have been achieved by adding hidden fields to your form. In AngularJS, we work from a model that is decoupled from the form, so we do not need these hidden fields. We simply add such values to the scope and then simulate the form submission using the `$http` service. See *Chapter 3, Communicating with a Back-end Server* for how to do this.

Looking inside ngModel data binding

Until now we have seen that `ngModel` creates a binding between the model and the value in an input field. In this section we look deeper into what else this directive provides and how it works.

Understanding ngModelController

Each `ngModel` directive creates an instance of `ngModelController`. This controller is made available to all the directives on the `input` element.

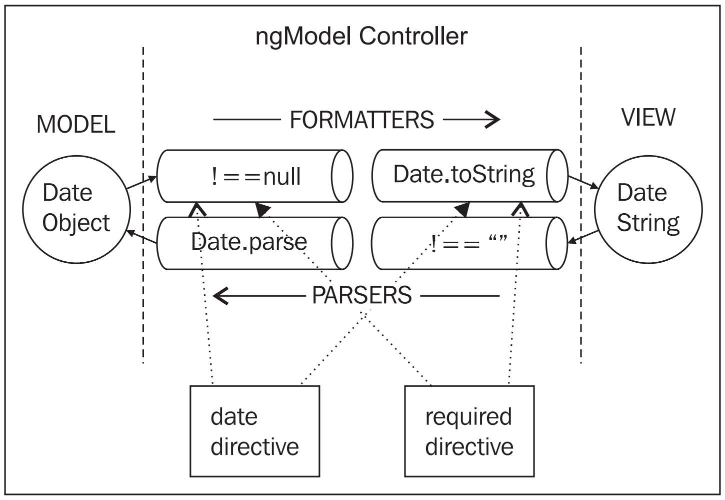


The `ngModelController` is responsible for managing the data binding between the value stored in the model (specified by `ngModel`) and the value displayed by the `input` element.

The `ngModelController` also tracks whether the view value is valid and whether it has been modified by the input element.

Transforming the value between the model and the view

The `ngModelController` has a transformation pipeline that it applies each time the data binding is updated. This consists of two arrays: `$formatters` that transforms from model into view and `$parsers` that transforms from view to model. Each directive on the `input` element can add in their own formatters and parsers to this pipeline in order to modify what happens to the data binding as shown in the following image:



Here two directives are adding to the transformation pipeline. The `date` directive is parsing and formatting dates. The `ng-required` directive is checking that the value is not missing.

Tracking whether the value has changed

Along with transforming the value between the model and the view, the `ngModelController` tracks whether the value has changed since it was initialized and whether the value is valid.

When it is first initialized the `ngModelController` marks the value as pristine, that is, it has not been modified. It exposes this as an `ng-pristine` CSS class on the input element. When the view changes, say through typing in an input box, the value is marked as dirty. It replaces the `ng-pristine` CSS class with the `ng-dirty` CSS class.

By providing CSS styles for these classes, we can change the appearance of the input element based on whether the user has entered or modified the data:

```
.ng-pristine { border: solid black 1px; }
.ng-dirty { border: solid black 3px; }
```

Here we make the border of the element thicker if the user has made changes to the input.

Tracking input field validity

Directives on an input element can also tell the `ngModelController` whether they believe the value is `valid` or `invalid`. This is normally done by hooking into the transformation pipeline and checking the value rather than transforming it. The `ngModelController` tracks the validity and applies the `ng-valid` or `ng-invalid` CSS classes accordingly. We can provide further styles to change the appearance of the element based on these classes:

```
.ng-valid.ng-dirty { border: solid green 3px; }
.ng-invalid.ng-dirty { border: solid red 3px; }
```

Here, we are using a combination of `pristine` and `invalid` to ensure that only fields that have been changed by user input are styled: thick red border when `invalid` and thick green border when `valid`.

In the next section, Validating forms, we will see how we can work with the concepts of pristine, dirty, valid, and invalid programmatically.

Validating AngularJS forms

In this section we explain how to use validation directives and how they work with `ngFormController` to provide a full validation framework.

Understanding ngFormController

Each `form` (or `ngForm)` directive creates an instance of `ngFormController`. The `ngFormController` object manages whether the form is valid or invalid and whether it is pristine or dirty. Importantly, it works with `ngModelController` to track each `ngModel` field within the form.

When an `ngModelController` is created, it registers itself with the first `ngFormController` it comes across as it traverses up its list of parent elements. This way, the `ngFormController` knows what input directives it should track. It can check whether these fields are valid/invalid or pristine/dirty and set whether the form is valid/invalid or pristine/dirty accordingly.

Using the name attribute to attach forms to the scope

You can make the `ngFormController` appear on the local scope by giving the form a name. Any input elements within the form that also have names will have their `ngModelController` object attached as a property to this `ngFormController` object.

The following table shows how the scope contains the controllers associated with each of the elements in the form:

HTML	Scope	Controller
	`model1, model2, …`	
`<form name="form1">`	`form1 : {`	`ngFormController`
	`$valid, $invalid,`	
	`$pristine, $dirty, …`	
`<input`	`field1: {`	`ngModelController`
`name="field1"`	`$valid, $invalid,`	
`ng-model="model1"`	`$pristine, $dirty, …`	
`/>`	`},`	
`<input`	`field2: {`	`ngModelController`
`name="field2"`	`$valid, $invalid,`	
`ng-model="model2"`	`$pristine, $dirty, …`	
`/>`	`}`	
`</form>`	`},`	

Adding dynamic behavior to the User Information Form

Our form allows us to enter values into fields and we can change the appearance of the input elements based on the values entered. But for a more responsive user experience, we would like to show and hide validation messages and change the state of buttons on our form depending upon the state of the form fields.

Having the `ngFormController` and `ngModelControllers` objects on our scope allows us to work with the state of the form programmatically. We can use values like `$invalid` and `$dirty` to change what is enabled or visible to our user.

Showing validation errors

We can show error messages for inputs and for the form as a whole if something is not valid. In the template:

```
<form name="userInfoForm">
  <div class="control-group"
       ng-class="getCssClasses(userInfoForm.email)">

    <label>E-mail</label>
    <input type="email" ng-model="user.email"
           name="email" required>

    <span ng-show="showError(userInfoForm.email, 'email')" ...>
       You must enter a valid email
    </span>

    <span ng-show="showError(userInfoForm.email, 'required')" ...>
       This field is required
    </span>
  </div>
  ...
</form>
```

In the controller:

```
app.controller('MainCtrl', function($scope) {
  $scope.getCssClasses = function(ngModelContoller) {
    return {
      error: ngModelContoller.$invalid && ngModelContoller.$dirty,
      success: ngModelContoller.$valid && ngModelContoller.$dirty
    };
  };
  $scope.showError = function(ngModelController, error) {
    return ngModelController.$error[error];
  };
});
```

Try it at `http://bit.ly/XwLUFZ`.

This example shows the e-mail input from our User Form. We are using Twitter Bootstrap CSS to style the form, hence the `control-group` and `inline-help` CSS classes. We have also created two helper functions in the controller.

The `ng-class` directive will update the CSS classes on `div` that contains the label, the input, and the help text. It calls the `getCssClasses()` method, passing in an object and an error name.

> The object parameter is actually the `ngModelController`, which has been exposed on the `ngFormController`, which in turn is exposed on the `scope.userInfoForm.email` scope.

The `getCssClasses()` method returns an object that defines which CSS classes should be added. The key of each object refers to the name of a CSS class. The value of each member is `true` if the class is to be added. In this case `getCssClasses()` will return `error` if the model is dirty and invalid and `success` if the model is dirty and valid.

Disabling the save button

We can disable the save button when the form is not in a state to be saved.

```
<form name="userInfoForm">
  ...
  <button ng-disabled="!canSave()">Save</button>
</form>
```

In our view, we add a `Save` button with an `ngDisabled` directive. This directive will disable the button whenever its expression evaluates to true. In this case it is negating the result of calling the `canSave()` method. We provide the `canSave()` method on the current scope. We will do this in our main controller:

```
app.controller('MainCtrl', function($scope) {
  $scope.canSave = function() {
    return $scope.userInfoForm.$dirty &&
           $scope.userInfoForm.$valid;
  };
});
```

Try it at `http://bit.ly/123zIhw`.

The `canSave()` method checks whether the `userInfoForm` has the `$dirty` and `$valid` flags set. If so, the form is ready to save.

Disabling native browser validation

Modern browsers naturally try to validate the input values in a form. Normally this occurs when the form is submitted. For instance, if you have a required attribute on an input box, the browser will complain independently of AngularJS, if the field does not contain a value when you try to submit the form.

Since we are providing all the validation through AngularJS directives and controllers, we do not want the browser to attempt its own native validation. We can turn off this by applying the HTML5 `novalidate` attribute to the form element:

```
<form name="novalidateForm" novalidate>
```

Try it at `http://bit.ly/1110hS4`.

This form is called `novalidateForm` and the `novalidate` attribute will tell the browser not to attempt the validation on any of the inputs in the form.

Nesting forms in other forms

Unlike standard HTML forms, AngularJS forms can be nested inside each other. Since form tags inside other form tags are invalid HTML, AngularJS provides the `ngForm` directive for nesting forms.

Each form that provides a name will be added to its parent form, or directly to the scope if it has no parent form.

Using subforms as reusable components

A nested form acts like a composite field that exposes its own validation information based on the fields that it contains. Such forms can be used to reuse as subforms by including them in container forms. Here we group two input boxes together to create a password and password confirmation widget:

```
<script type="text/ng-template" id="password-form">
  <ng-form name="passwordForm">
    <div ng-show="user.password != user.password2">
      Passwords do not match
    </div>
    <label>Password</label>
    <input ng-model="user.password" type="password" required>
```

```
      <label>Confirm Password</label>
      <input ng-model="user.password2" type="password" required>
    </ng-form>
  </script>

  <form name="form1" novalidate>
    <legend>User Form</legend>
    <label>Name</label>
    <input ng-model="user.name" required>
    <ng-include src="'password-form'"></ng-include>
  </form>
```

Try it at `http://bit.ly/10QWwyu`.

We define our subform in a partial template. In this case it is inline in a script block but it could be in a separate file also. Next we have our container form, `form1`,which includes the subform by using the `ngInclude` directive.

The subform has its own validity state and related CSS classes. Also, notice that because the subform has a name attribute, it appears as a property on the container form.

Repeating subforms

Sometimes, we have fields in a form that needs to be repeated by an arbitrary number of times based on the data in the model. This is a common situation where you want to provide a single form that can display a one-to-many relationship in the data.

In our SCRUM app, we would like to allow users to have zero or more website URLs in their User Info profile. We can use an `ngRepeat` directive to set this up:

```
    <form ng-controller="MainCtrl">
    <h1>User Info</h1>
    <label>Websites</label>
    <div ng-repeat="website in user.websites">
      <input type="url" ng-model="website.url">
      <button ng-click="remove($index)">X</button>
    </div>
    <button ng-click="add()">Add Website</button>
  </form>
```

The controller initializes the model and provides the helper functions, `remove()` and `add()`:

```
app.controller('MainCtrl', function($scope) {
  $scope.user = {
    websites: [
      {url: 'http://www.bloggs.com'},
      {url: 'http://www.jo-b.com'}
    ]
  };
  $scope.remove = function(index) {
    $scope.user.websites.splice(index, 1);
  };
  $scope.add = function() {
    $scope.user.websites.push({ url: ''});
  };
});
```

Try it at `http://bit.ly/XHLEWQ`.

In the template, we have an `ngRepeat` directive that iterates over the websites in the user's profile. Each input directive in the repeat block is data bound to the appropriate `website.url` in the `user.websites` model. The helper functions take care of adding and removing items to and from the array and AngularJS data binding does the rest.

It is tempting for each website item in the website's array to be a simple string containing the URL. This will not work since, in JavaScript, strings are passed by value and so the reference between the string in the `ngRepeat` block and the string in the array will be lost when you modify the value of the input box.

Validating repeated inputs

The problem with this approach comes when you want to do work with validation on these repeated fields. We need each input to have a unique name within the form in order to access that field's validity, `$valid`, `$invalid`, `$pristine`, `$dirty`, and so on. Unfortunately, AngularJS does not allow you to dynamically generate the name attribute for `input` directives. The name must be a fixed string.

We solve this problem by using nested forms. Each exposes itself on the current scope, so if we place a nested form inside each repeated block that contains the repeated input directives, we will have access on that scope to the field's validity.

Template:

```
<form novalidate ng-controller="MainCtrl" name="userForm">
  <label>Websites</label>
  <div ng-show="userForm.$invalid">The User Form is invalid.</div>
  <div ng-repeat="website in user.websites" ng-form="websiteForm">
    <input type="url" name="website"
           ng-model="website.url" required>
    <button ng-click="remove($index)">X</button>
    <span ng-show="showError(websiteForm.website, 'url')">
       Pleae must enter a valid url</span>
    <span ng-show="showError(websiteForm.website, 'required')">
       This field is required</span>
  </div>
  <button ng-click="addWebsite()">Add Website</button>
</form>
```

Controller:

```
app.controller('MainCtrl', function($scope) {
  $scope.showError = function(ngModelController, error) {
    return ngModelController.$error[error];
  };
  $scope.user = {
    websites: [
      {url: 'http://www.bloggs.com'},
      {url: 'http://www.jo-b.com'}
    ]
  };
});
```

Try it at `http://bit.ly/14i1sTp`.

Here, we are applying the `ngForm` directive to `div`, to create a nested form, which is repeated for each website in the array of `websites` on the scope. Each of the nested forms is called `websiteForm` and each input in the form is called `website`. This means that we are able to access the validity of the `ngModel` for each website from within the `ngRepeat` scope.

We make use of this to show an error message when the input is invalid. The two `ng-show` directives will show their error messages when the `showError` function returns `true`. The `showError` function checks the passed in `ngModelController` to see if it has the relevant validation entry in the `$error` field. We can pass `websiteForm.website` to this function since this refers to the `ngModelController` object for our website input box.

Outside the `ngForm` we cannot reference the `websiteForm` (`ngFormController`) object on the scope or the `websiteForm.website` (`ngModelController`) object since they do not exist in this scope. We can, however, access the containing `userForm` (`ngFormController)` object. This form's validity is based upon the validity of all its child inputs and forms. If one of the `websiteForms` is invalid, so is the `userForm`. The div at the top of the form displays an overall error message only if `userForms.$valid` is true.

Handling traditional HTML form submission

In this section we take a look at how AngularJS handles submission of forms. Single Page AJAX Applications, for which AngularJS is perfect, don't tend to follow the same process of direct submission to the server as traditional web application do. But sometimes your application must support this. Here we show the various submission scenarios that you may wish to implement when submitting form data back to a server.

Submitting forms directly to the server

If you include an `action` attribute on a form in an AngularJS application, then the form will submit as normal to the URL defined in the action:

```
<form method="get" action="http://www.google.com/search">
  <input name="q">
</form>
```

Try it at `http://bit.ly/115cQgq`.

[Be aware that the Plnkr preview will block the redirection to Google.]

Handling form submission events

If you don't include an `action` attribute, then AngularJS assumes that we are going to handle the submission on the client side by calling a function on the scope. In this case, AngularJS will prevent the form trying to directly submit to the server.

We can trigger this client-side function by using the `ngClick` directive on a `button` or the `ngSubmit` directive on the `form`.

[You should not use both the `ngSubmit` and `ngClick` directives on the same `form` because the browser will trigger both directives and you will get double submission.]

Using ngSubmit to handle form submission

To use `ngSubmit` on a form, you provide an expression that will be evaluated when the form is submitted. The form submission will happen when the user hits *Enter* in one of the inputs or clicks on one of the buttons:

```
<form ng-submit="showAlert(q)">
  <input ng-model="q">
</form>
```

Try it at `http://bit.ly/ZQBLYj`.

Here, hitting *Enter* while in the input will call the `showAlert` method.

[You should use `ngSubmit` only on a form that has only one input and not more than one button, such as our search form in the example.]

Using ngClick to handle form submission

To use `ngClick`, on a `button` or `input[type=submit]`, you provide an expression that will be evaluated when the button is clicked:

```
<form>
  <input ng-model="q">
  <button ng-click="showAlert(q)">Search</button>
</form>
```

Try it at `http://bit.ly/153OvLS`.

Here, clicking on the button or hitting *Enter* in the input field will call the `showAlert` method.

Resetting the User Info form

In our User Info form, we would like to cancel the changes and reset the form back to its original state. We do this by holding a copy of the original model with which we can overwrite any changes that the user has made.

Template:

```
<form name="userInfoForm">
  ...
  <button ng-click="revert()" ng-disabled="!canRevert()">Revert
Changes</button>
</form>
```

Controller:

```
app.controller('MainCtrl', function($scope) {
  ...
  $scope.user = {
    ...
  };
  $scope.passwordRepeat = $scope.user.password;

  var original = angular.copy($scope.user);

  $scope.revert = function() {
    $scope.user = angular.copy(original);
    $scope.passwordRepeat = $scope.user.password;
    $scope.userInfoForm.$setPristine();
  };

  $scope.canRevert = function() {
    return !angular.equals($scope.user, original);
  };

  $scope.canSave = function() {
    return $scope.userInfoForm.$valid &&
      !angular.equals($scope.user, original);
  };
});
```

Try it at `http://bit.ly/17vHLWX`.

Here, we have a button to revert the model back to its original state. Clicking on this button calls `revert()` on the scope. The button is disabled if `canRevert()` returns `false`.

In the controller, you can see that we use `angular.copy()` to make a copy of the model and place it in a local variable. The `revert()` method copies this original back over to the working `user` model and sets the form back to a pristine state so that all the CSS classes are no longer set to `ng-dirty`.

Summary

In this chapter we have seen how AngularJS extends standard HTML form controls to provide a more flexible and powerful system for getting input from the user. It enables a separation of the model from the view through `ngModel` and provides mechanisms to track changes and validity of input values through validation directives and the `ngFormController` object.

In the next chapter we will look at how to best manage navigation round our application. We will see how AngularJS supports deep linking to map URLs directly into aspects of our application and how to use `ngView` to automatically display relevant content to the user based on the current URL.