

Angular

IN ACTION

Jeremy Wilken

Sample Chapter





Angular in Action

by Jeremy Wilken

Chapter 9

Copyright 2018 Manning Publications

brief contents

- 1 ■ Angular: a modern web platform 1
- 2 ■ Building your first Angular app 25
- 3 ■ App essentials 54
- 4 ■ Component basics 76
- 5 ■ Advanced components 104
- 6 ■ Services 128
- 7 ■ Routing 159
- 8 ■ Building custom directives and pipes 188
- 9 ■ Forms 208
- 10 ■ Testing your application 242
- 11 ■ Angular in production 275

This chapter covers

- Creating forms using Angular's forms libraries
- Deciding between using Reactive or Template forms
- Validating forms with custom logic
- Accessing data and watching input changes
- Submitting form data and handling errors gracefully
- Creating custom form controls

Just about every application uses forms in some way, if only to do something simple like log in or manage settings. HTML comes with a number of form elements by default, such as inputs, selects, and buttons, and Angular provides a way to use these native elements and add some power to them. We've used forms in several previous examples, but in this chapter we'll dig into them much more completely.

Angular provides two approaches to building forms: *reactive forms* and *template forms*. I'll discuss the differences at length shortly, though they're largely in whether

you define the form in your controller or in the template. You're not limited to choosing only one or the other form, but typically applications will try to remain consistent with one of them.

With template forms, we'll see how to describe your forms primarily using the `NgModel` directive, which is used to define the form structure. With reactive forms, you declare the form structure yourself in the controller, and the template renders it out.

As complex as forms can become, the basics are fairly standard in all areas. There are form controls (fields that hold values like inputs, selects, and so on), and there are form buttons (like Save, Cancel, or Reset). The same holds true when working with forms in Angular—the basics remain consistent regardless of how complex the form becomes.

There are often situations where a form requires the use of additional third-party components to help, such as a date picker or range slider. Browsers may implement some newer form controls, but they're rarely standard, and other browsers might not support them at all. Although we're not going to focus on creating custom components that act like form controls, there are many great libraries that provide you additional features, or you can certainly build your own custom form controls by reviewing the documentation. I personally avoid creating these unless absolutely necessary, which it rarely is.

9.1 *Setting up the chapter example*

We're going to build a new application that helps us manage invoices and customers. Imagine you're a freelancer or small business owner and you have customers to manage. This application would be a good tool for keeping track of sending invoices and making sure you get paid (which is pretty important, right?).

The forms themselves are intentionally not complex, but they do demonstrate most of the needs for forms succinctly. You can take the examples you see in this chapter and translate them into larger, more complex forms without having to learn additional concepts. The only difference tends to be the scale.

The application is also designed for the mobile form factor, which is a nice little twist from our previous examples. It uses the Covalent UI library, from Teradata, which extends the concepts and Angular Material Design library. If you weren't aware, mobile browsers tend to have the best support for the latest HTML5 input types, such as search or number fields, which we will use for our example. I recommend using Chrome for this chapter.

Chrome has a useful device emulator in the developer tools, as shown in figure 9.1, and I suggest that you use it while you're building and using this application. It allows you to emulate the dimensions of various mobile devices and get a sense of how your application would look on those sizes. It doesn't really emulate the device in a true way, but it does provide an easy way to preview.

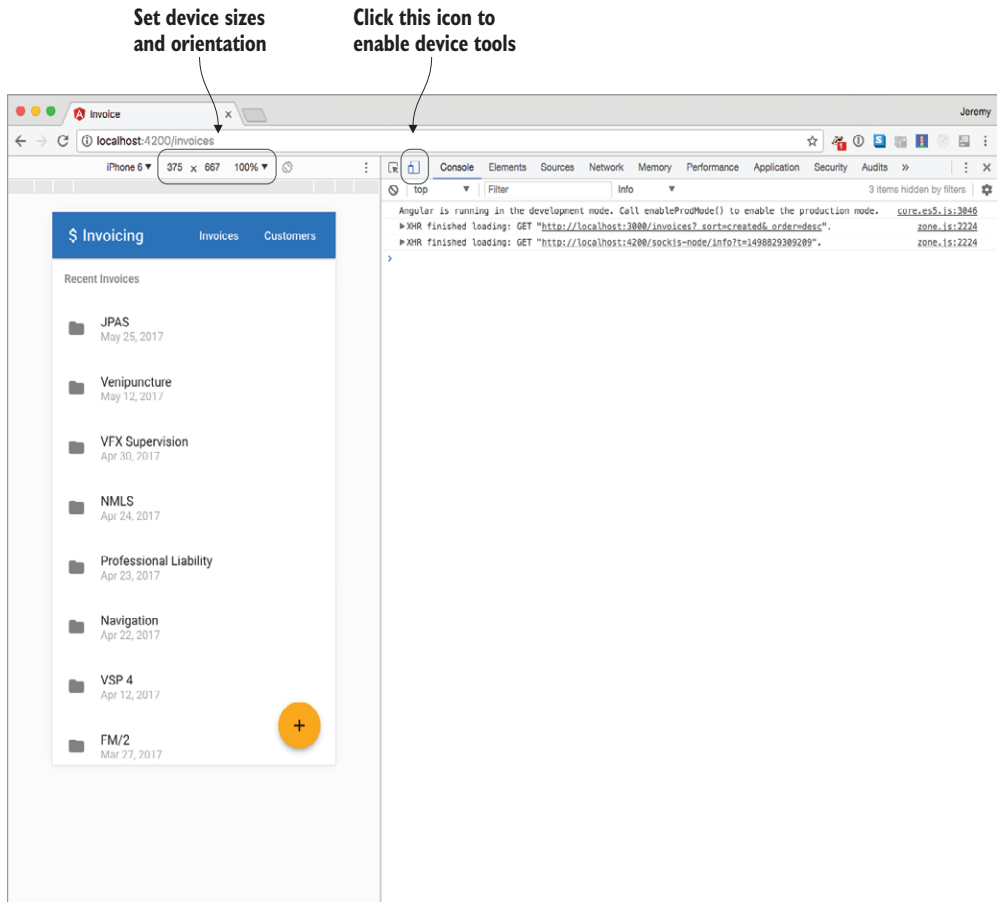


Figure 9.1 Use device tools in Chrome developer tools to simulate mobile devices.

Like other examples, this one is available on GitHub, and you can get the source code by cloning the repo. Make sure to check out the right tag when we start so you can code along, or look at the latest master for the final version:

```
git clone https://github.com/angular-in-action/invoice
cd invoice
git checkout start
```

Otherwise, you can download the archived files from <https://github.com/angular-in-action/invoice/archive/start.zip> and unzip the files.

When you start the application, you'll notice a number of services and components are there already. I've provided the majority of the code ahead of time so we can focus on the key features for forms. Even the form components are there, with standard HTML forms in place. They don't currently do anything when you try to save them, which is what we'll be updating and implementing in this chapter.

You'll need `npm install` for all the dependencies, and then you run `ng serve` to start the local development server. That isn't all, though. This app has a local API server that we need to run also. You'll need to open another terminal session and run the following:

```
npm run api
```

This will start up a local server that provides our app data. As you save and edit records, the data will persist into a local file called `db.json`, which is important for our app.

There may be a few warnings in the browser console when you run the example—you can safely ignore those. They refer to features that aren't necessary for the chapter example.

Now, before we get into the forms, let's review the rest of the app.

9.1.1 Review the app before starting

There are six routable views in this application. Let's talk briefly about some of the ones that don't contain forms. We'll focus only on two of the routes in the chapter; we'll build one of the forms with template-driven forms and the other with a reactive form. Let's take a look at several of the screens of the application.

Figure 9.2 shows some of the screens for the application, such as the list, detail, and form views. There are two list views, one for customers and one for invoices. These are both fairly simple in that they load a list from the API and render it. They also include a button that will take you to a form to create a new record. You can see these two in the Invoices and Customers components.

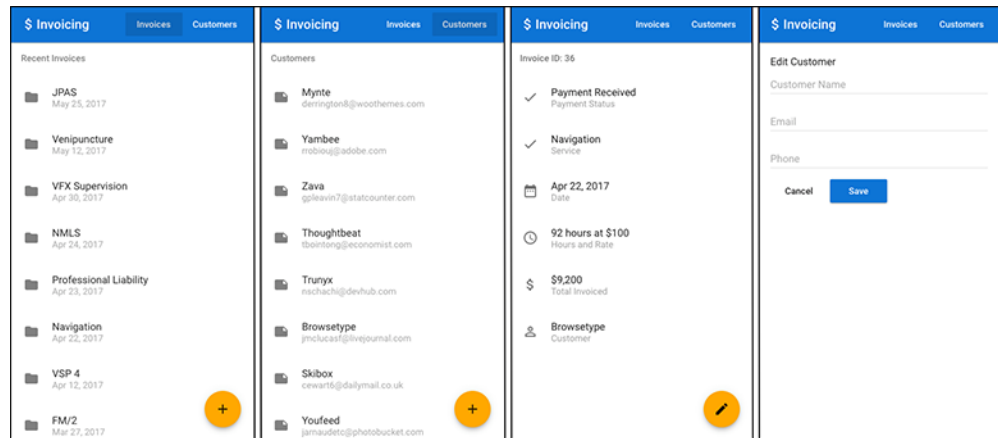


Figure 9.2 Invoicing application screens, emulated in a mobile device size. From left to right: list of invoices, list of customers, invoice detail view, and customer form.

The two detail views for a customer and invoice are also quite similar in that they simply show the relevant data for a given record. There is a button that allows you to edit that record as well. You can preview these in the Invoice and Customer components.

Finally, the two views we'll work on are the form views. The customer or invoice form will let you create or edit an existing record, and the form fields needed are already

provided with standard HTML. We'll be updating these forms and the controllers to handle the save, delete, and cancel events. These can be found in the `InvoiceForm` and `CustomerForm` components.

Inside of the components you'll see a few new things. The `TdLoading` directive is a feature from the Covalent library to display a loading indicator while data is being loaded. The `MdInput` directive will make an input Material Design-compliant. There are several other elements that start with `Md-`, which are all from the Material Design library and are UI components for structure or controls. It's best to review the Covalent and Material Design documentation for additional questions you may have about the use of these tools. Please note the specific version being used in the `package.json` file and make sure you're looking it up correctly.

There are also services for the customers and invoices APIs. You may want to review them as a way to extend one service to create another. Both the `Invoices` and `Customers` services extend the `Rest` service, which implements the basic API calls needed. The specialized instances (`Invoices` and `Customers`) provide a single property that's used by the `Rest` service to construct URLs.

All right, let's create the customer form using the template-driven approach.

9.2 **Template-driven forms**

We've already used template-driven forms in several of our examples, and the key marker is when you see the `NgModel` directive on a form control. AngularJS developers will be familiar with the patterns described in this section.

Template forms are named primarily because the form controls are defined in the template of the component. In this case, you can think of the template as having the final say about what is part of the form or not. For example, if you have an `input` element in the page that is part of the form that is wired up into the form controls, then it will also be defined in the controller.

The primary goal of a form is to be able to synchronize the data in the view with data in the controller so it can be submitted to be handled. Secondary goals are to perform tasks like validation, notify about errors, and handle other events like cancel.

Because the form is primarily defined in the template layer, it also means that validation errors are managed primarily through the template. We'll look at how to add validation and alert the user about invalid fields.

In figure 9.3 you can see the customer form that we'll be building in this section. The three fields will be part of the form data and will allow us to capture the input for processing.

To get started, we need to start working with our form controls and wire them up so that they bind the model between both the controller and the template using `NgModel`.

9.2.1 **Binding model data to inputs with NgModel**

Let's take a single form control to begin with and see what it takes to turn it into something Angular can use. In the `CustomerForm` component, you should see this input:

```
<input mdInput placeholder="Customer Name" value="">
```


The screenshot shows a web interface with a blue header bar containing a dollar sign icon, the text 'Invoicing', and two tabs: 'Invoices' and 'Customers'. Below the header is a form titled 'Edit Customer'. The form contains three input fields: 'Customer Name *' (with placeholder text 'Linklinks'), 'Email *' (with value 'mscoon0@hibu.com'), and 'Phone *' (with value '(970)426-1402'). At the bottom of the form are three buttons: 'Delete', 'Cancel', and 'Save'.

Figure 9.3 Customer form with three fields to bind data to

Right now it's just a normal form element (with the `MdInput` to make it Material Design), but by adding the `NgModel` directive we can turn it into an Angular form control. In the process, we can also remove the `value` attribute, as it's no longer needed:

```
<input name="customer" mdInput placeholder="Customer Name"
      [(ngModel)]="customer.name">
```

The `NgModel` directive is part of the Forms module and will ensure that the value of the form control is set based on the `customer` property value from the controller. But it also sets the value into the controller when it's changed in the view. If you look at the controller, there is no such property, and `NgModel` will create it for you.

You should recall the `[()]` syntax from earlier chapters, but to refresh your memory, it's a way of doing two-way data binding in Angular. That means the controller will now have a property called `customer`, and if the view or controller changes that value, the other will instantly be updated as well. AngularJS developers will know this concept well, and it exists in Angular as well.

Let's go ahead and wire up all of the form controls with `NgModel`. In the Customer-Form template, we'll need to modify the existing form controls, as you see bolded in the following listing. Open `src/app/customer-form/customer-form.component.html` and update it.

Listing 9.1 CustomerForm using NgModel

```
<md-card-content>
  <md-input-container>
    <input name="customer" mdInput placeholder="Customer Name"
          [(ngModel)]="customer.name">
  </md-input-container>
</md-input-container>
```

Form control using NgModel

```

<input name="email" mdInput type="email" placeholder="Email"
  [(ngModel)]="customer.email"> ← Form control using NgModel
</md-input-container>
<md-input-container>
  <input name="phone" mdInput type="tel" placeholder="Phone"
    [(ngModel)]="customer.phone"> ← Form control using NgModel
  </md-input-container>
  {{customer | json}} ← Displays the customer content in template temporarily
</md-card-content>

```

In these form controls, we now have them wired up to do two-way binding using `NgModel`. Notice that we're also setting the model values as part of the customer property, so the data is stored in one object. Except for a form that has only one control, it's highly recommended to always use models like this. It will help us later to have all the customer data stored on the same object instead of on different properties of the controller.

So far, this won't change anything significant about our form that you can see, but it will be adding new properties to the customer model as you change the input values behind the scenes. If you watch the customer interpolation binding, you will see that as values are changed in the form inputs, the model is updated.

There are a couple of notes to make about using `NgModel`. First, you always use it with the two-way binding syntax—it doesn't work otherwise. Second, inputs should always have a name when using `NgModel`, because it requires that information internally. Last, you notice the value attribute was omitted because `NgModel` will overwrite it and it's best to just leave it off.

Save these changes and then go to the customer's list, select one, and click the edit icon in the bottom right to view the form. You should ensure there are no errors by looking at the browser console as well, in case you typed something incorrectly.

This is great, because our primary object is largely complete. We simply add `NgModel`, and our form elements are now being tracked in both the template and controller as changes are made. The next step is to start validating these form fields. Using the power of `NgModel`, we can track the validity of a form field and report meaningful errors to the user.

9.2.2 Validating form controls with `NgModel`

HTML already provides some built-in form validations that can be put onto form elements, such as `required` or `minlength`. Angular works with these attributes and automatically will validate inputs based on them.

Let's take the example of our customer name input field. All we need to do is add the additional `required` attribute to validate the input to force validation for this field:

```

<input name="customer" mdInput placeholder="Customer Name"
  [(ngModel)]="customer.name" required>

```

When the form control has an invalid value, we can also inspect the state of a field and render out messages about what is incorrect, as shown in figure 9.4.

Figure 9.4 Customer form with validation errors

It's time to set up the validation for all the fields and also to look at how to access the state of those fields. Update the CustomerForm template snippet as you see bolded in the following listing.

Listing 9.2 CustomerForm validating fields with NgModel

```
<md-input-container>
  <input name="customer" mdInput placeholder="Customer Name"
    [(ngModel)]="customer.name" required #name="ngModel">
    <md-error *ngIf="name.touched && name.invalid">
      Name is required
    </md-error>
</md-input-container>
<md-input-container>
  <input name="email" mdInput type="email" placeholder="Email"
    [(ngModel)]="customer.email" required #email="ngModel">
    <md-error *ngIf="email.touched && email.invalid">
      A valid email is required
    </md-error>
</md-input-container>
<md-input-container>
  <input name="phone" mdInput type="tel" placeholder="Phone"
    [(ngModel)]="customer.phone" required #phone="ngModel" minlength="7">
    <md-error *ngIf="phone.touched && phone.errors?.required">
      Phone number is required
    </md-error>
    <md-error *ngIf="phone.touched && phone.errors?.minlength">
      Not a valid phone number
    </md-error>
</md-input-container>
```

← Adds validation attributes and template variable to form control

Uses form control validation to conditionally show error message

← Form control exposes what specific error is found.

The form controls now each have a required attribute and a local template variable. The phone number also has a minlength attribute, because we expect a phone number to be at least seven digits. We've used local template variables in the component chapter to access values from other controllers inside of the template, and that's precisely the same

thing here. For example, `#name="ngModel"` is a way to define the template variable name to be a reference to the `NgModel` result, which is the form control data. Remember, template variables are only valid within the template they're defined in, so you can't reach them from your controller.

This form control data is a `FormControl` type from Angular, which you can view in the API docs to see more about what it can do for you. It has a number of properties, such as `valid`, `invalid`, `pristine`, and `dirty`. These are Boolean values that you can easily use to determine whether something is true or false. See table 9.1 for the most useful form control properties.

Table 9.1 Form control validation properties

Property	Meaning
<code>valid</code>	The form control is valid for all validations.
<code>invalid</code>	The form control has at least one invalid validation.
<code>disabled</code>	The form control is disabled and can't be interacted with.
<code>enabled</code>	The form control is enabled and can be clicked or edited.
<code>errors</code>	An object that either contains keys with validations that are <code>invalid</code> , or null when all are <code>valid</code> .
<code>pristine</code>	The form control has not yet been changed by the user.
<code>dirty</code>	The form control has been changed by the user.
<code>touched</code>	The form control has been in focus, and then focus has left the field.
<code>untouched</code>	The form control has not been in focus.

The `MdError` element is from the Material Design library and shows a little validation error when the `NgIf` is true. For example, `*ngIf="email.touched && email.invalid"` will show the error when the form control is `invalid`, and the user has left focus on that field. (As a side note, if the value was loaded from a database but was `invalid`, the preceding validation would fail, so you should consider the needs of your application.) This is nice because the error doesn't appear immediately, but only when the user tries to leave the field with an invalid value. You can use different combinations of the properties in table 9.1 to determine when to show a validation error. When you're creating a new item, all the required fields will be `invalid`, but it won't show validation errors until the user has tried to edit them.

Notice how the validation message for the phone number has two different validations: `required` and `minlength`. We're then able to look at the control's error object to determine whether a specific validation failed and show the appropriate message. In this case, if the user leaves it blank, it will prompt it to say the field is required, but if the user only inputs four characters, it will show that it expects at least seven digits.

It's also useful to note that Angular will apply various CSS classes to a form control based on its validation state. They mirror the properties in table 9.1, but have the

ng- prefix. For example, an invalid form control will have the ng-invalid class applied. This is useful if you want to craft your own styling for valid or invalid controls without any special work. We're not doing that here, but you could certainly take advantage of them. Some Angular UI libraries may come with support for them out of the box.

Though this validation is helpful, it's still possible to submit the form with invalid values. We'll prevent this from happening in a moment, but first I want to wrap up validation by creating our own validation directive.

9.2.3 Custom validation with directives

The validation for our phone number is somewhat lacking. We really would want it to enforce not just the length but also that the content matches a known phone format. Unfortunately, even the tel input type doesn't do that for us, so we'll have to implement our own custom validation using a directive. Our best effort so far has been to enforce a minlength validation, but that only cares about the number of characters, not the actual value.

Although there is the pattern validation attribute in HTML, which allows you to declare a regular expression to validate the input, it's not very usable and doesn't work in all browsers.

We'll need to create two things to make this happen: a customer validator function and a directive that uses the validator function. Start by creating a new directory at src/app/validators; then create a file inside it named phone.validator.ts, and add the code from the following listing.

Listing 9.3 Phone validator

```

import { AbstractControl, ValidatorFn } from '@angular/forms';

const expression = /((\d{3}\d{3}) | (\d{3}-)\d{3}-\d{4})/;

export function PhoneValidator(): ValidatorFn {
  return (control: AbstractControl): { [key: string]: any } => {
    const valid = expression.test(control.value) && control.value.length <
      14;
    return valid ? null : { phone: true };
  };
}

```

Regular expression to validate typical phone number

Returns a function to handle validation

Defines a function that returns a ValidatorFn type

Validates the control value against expression

This is a bit terse, so let's look at it step by step. First, we're defining a regular expression that should validate the primary phone number formats. You could select a different expression if your needs require. Then we're exporting a function that will return a function. The ValidatorFn interface expects that this returned function will accept a control as a parameter and return either null or an object with validation errors.

Our PhoneValidator function will return the real validation function to use during the validation. It accepts a single argument, which is the form control. For the most part, you only care about the control.value property, which holds the current value of the form control. Then inside of the validation function, it tests the current value

against the expression and returns either null, to mean it's valid, or an object if it's invalid, with a property explaining what is invalid.

If it returns an object, it expects you to give it a property with a value. Here it's a Boolean, but it could be any value you want to expose. Normally, I find Boolean is suitable unless you want to also provide the error message as a string. You can access the value in the local template control.errors property.

To use this validator we need to create a directive. Using the Angular CLI, generate a new directive like so:

```
ng generate directive validators/phone
```

Now open src/app/validators/phone.directive.ts and add the code found in the following listing to it. This will take the validator function we created a moment ago and make it possible to apply it to an element as an attribute.

Listing 9.4 Phone validator directive

```
import { Directive } from '@angular/core';
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';
import { PhoneValidator } from './phone.validator';

@Directive({
  selector: '[phone][ngModel]',
  providers: [{ provide: NG_VALIDATORS, useExisting: PhoneDirective, multi: true }]
})
export class PhoneDirective implements Validator {
  private validator = PhoneValidator();

  validate(control: AbstractControl): { [key: string]: any } {
    return this.validator(control);
  }
}
```

Selector designed to apply to elements with phone and NgModel attributes

Way to define this directive as part of Angular's list of validators

Creates instance of validation function

Implements the Validator interface

Method that form controls will call to validate value

This is also a bit terse, but what we're doing is implementing the necessary pieces to wire up the directive to Angular's list of validators and implement the same interface. We start by defining the selector to expect to have both phone and NgModel attributes on the form control. This means if you just put phone as an attribute, it won't use this directive for validation, because NgModel is required.

The directive also has a providers array and uses a multiprovider, which allows a single token (like NG_VALIDATORS) to have multiple dependencies. NG_VALIDATORS contains a list of default validation dependencies, and this extends that list by adding one more of our own. This isn't very common, but it's required in this situation.

Our directive then exports a class, which implements the Validator interface. This expects that there will be a validate method defined in the class, which we have done. We also have a property that holds an instance of our validator function that we imported, and then inside of the validate method we call our custom validator and pass in the control.

There's a bit of juggling of the form control in this custom validation process, but when you look at these two files together, it should be clearer how they relate to one another. To implement this new validator directive, we need to update our phone form control, as you see in the following listing.

Listing 9.5 Updated phone form control

```
<md-input-container>
  <input name="phone" mdInput type="tel" placeholder="Phone"
    [(ngModel)]="customer.phone" required phone #phone="ngModel">
  <md-error *ngIf="phone.touched && phone.errors?.phone">
    Not a valid phone number
  </md-error>
</md-input-container>
```

Adds the phone directive

Looks for validation errors of phone type

The form control removes the `minlength` attribute and replaces it with the `phone` attribute. This makes the form control now aware of the phone validation, and when the number isn't a correct phone number we can tell by looking at the `errors.phone` property. Recall our validator function returns an object with `{phone: true}`, so this is where we see it returned to us. We also removed the additional error message for it being required, as our new validation covers that scenario as well.

To review, when we add the `phone` attribute, the `NgModel` will validate using the `Phone` validator directive. Internally, the `Phone` validator directive registers itself with the default list of validators that `NgModel` knows about by declaring the `multiprovider` (a special kind of provider that can be registered more than once) for `NG_VALIDATORS`. It then implements a `validate` method, which calls the `validator` function we created at the beginning. There are a few steps here, but that's the price we pay for the flexibility provided by Angular's platform.

Congrats! You've now got a custom validation directive that you can reuse on any form control, or you can create additional ones for different scenarios. Now we need to wrap up this form by handling events to either submit, cancel, or delete.

9.2.4 Handling submit or cancel events

We've got all the data and validation we would like on this form, so now it's time to handle the various events that might happen with it. The most important is to handle the submit event, but also we want to allow the user to cancel from saving the edits or delete the record if it exists.

The controller already implements all the methods we need to handle these scenarios, so we just need to write up our form to call them properly. You can review the methods in there and see how they work.

The first thing we should do is update our form element. Angular does another thing to forms that isn't visible by default. It automatically implements an `NgForm` on a form even if you don't declare a directive (unlike how you have to declare `NgModel`). When it does this, it essentially attaches an `NgForm` controller that then maintains the form controls in the form.

NgForm provides a couple of features we'll need; the first is that it can tell us if the entire form is valid (not just an individual field) and help us implement an event binding for submitting the form. Find the form element at the top of the CustomerForm template and update it to have these additional values shown in bold:

```
<form *ngIf="customer" #form="ngForm" (ngSubmit)="save()">
```

First we create another template variable and reference the NgForm controller. This is the same idea we used for our form controls with NgModel, except this local template variable will reference the entire form. Then we have an (ngSubmit) event handler to call the save method.

Now we just need to update our buttons at the bottom to call the correct methods. The following code in bold contains the pieces to add to the buttons in the card actions element near the bottom:

```
<md-card-actions>
  <button type="button" md-button (click)="delete()" *ngIf="customer.
    id">Delete</button>
  <button type="button" md-button (click)="cancel()">Cancel</button>
  <button type="submit" md-raised-button color="primary" [disabled]="form.
    invalid">Save</button>
</md-card-actions>
```

The first two buttons are standard buttons, so we just use the click event binding to call the appropriate method. The delete button is hidden if we're creating the record by checking whether there is an ID, which is only set after creation. The submit button doesn't have an event binding, because that's already being handled by ngSubmit. But we do bind to the disabled property and look at the form.invalid property to determine if the entire form is invalid.

That about wraps up template-driven forms. Everything about our form was described in the template, primarily by adding NgModel directives to our form controls. Using local template variables that referenced the NgModel of a control, we could inspect the validation errors for a field and show appropriate error messages. We also were able to build a custom validator for phone numbers that works like any default validation attribute. Finally, we handled the submit event and checked the validation of the overall form before enabling the submit button. Not too bad for a modest amount of code! The final version of the customer form can be seen here in the following listing.

Listing 9.6 Final customer form template

```
<div *tdLoading="'customer'">
  <form *ngIf="customer" #form="ngForm" (ngSubmit)="save()">
    <md-card>
      <md-card-header>Edit Customer</md-card-header>
      <md-card-content>
        <md-input-container>
          <input name="customer" mdInput placeholder="Customer Name"
            [(ngModel)]="customer.name" required #name="ngModel">
          <md-error *ngIf="name.touched && name.invalid">
            Name is required
        </md-input-container>
      </md-card-content>
    </md-card>
  </form>
</div>
```



```

    </md-error>
  </md-input-container>
  <md-input-container>
    <input name="email" mdInput type="email" placeholder="Email"
    [(ngModel)]="customer.email" required #email="ngModel">
    <md-error *ngIf="email.touched && email.invalid">
      A valid email is required
    </md-error>
  </md-input-container>
  <md-input-container>
    <input name="phone" mdInput type="tel" placeholder="Phone"
    [(ngModel)]="customer.phone" required phone #phone="ngModel">
    <md-error *ngIf="phone.touched && phone.errors?.required">
      Phone number is required
    </md-error>
    <md-error *ngIf="phone.touched && phone.errors?.phone">
      Not a valid phone number
    </md-error>
  </md-input-container>
</md-card-content>
<md-card-actions>
  <button type="button" md-button (click)="delete()" *ngIf="customer.
id">Delete</button>
  <button type="button" md-button (click)="cancel()">Cancel</button>
  <button type="submit" md-raised-button color="primary"
[disabled]="form.invalid">Save</button>
</md-card-actions>
</md-card>
</form>
</div>

```

Now it's time to implement the other form for creating or editing an invoice in the reactive form style. It will approach the form from the controller first and have less logic in the template to manage.

9.3 Reactive forms

The alternative to template-driven forms, *reactive* forms, is the other way to design your forms in Angular. The name *reactive* comes from the style of programming known as reactive, where you have immutable data structures and your views never mutate them directly. That means no two-way binding is allowed.

The basic idea is that your form has a copy of the original model that it uses while the user is editing the form, and upon saving, you trigger an action like saving it to the database and update the original model. Template-driven forms only have one shared model, and because values are being constantly synced between the two, there may be timing issues of values changing in multiple places.

One of my favorite aspects of reactive forms is that you can use an observable to watch a particular form control for changes. I might do this to handle a task like autocomplete, for example. It's been useful for me on several occasions, and template-driven forms don't have a good way to do this.

Reactive forms still have a template, because you need to define the markup associated with the form. The main difference in the template is you won't use `NgForm` or `NgModel` on any of the form controls; instead we'll use a different directive to link a particular form control in the template to the corresponding form control declared in the controller.

There are a few other differences in the way that reactive forms behave. Because template-driven forms employ two-way binding concepts, they're inherently asynchronous in their handling. During the rendering of a template-driven form, the `NgModel` directive is building the form up for you behind the scenes. This takes more than one change detection cycle, though, causing potential race conditions where you expect a form element to be registered but it hasn't yet. This doesn't happen with reactive forms, because you define the form in your controller, so it's not dependent on change detection cycles.

The challenges with timing of template-driven forms tend to only appear when you try to access form controls or the form itself too early, and require you to wait until the `AfterViewInit` lifecycle hook to ensure the view has fully rendered. The Angular documentation covers some details about the differences and virtues of each approach as well and is worth reviewing: <https://angular.io/guide/forms>.

Setting aside some of the internal mechanical differences, let's focus on what reactive forms look like. In a template-driven form the `NgModel` builds the form controls, but with reactive we need to define our form programmatically in the controller. When you settle on using one form approach, it's not easy or advisable to mix them in the same form, though you could in different forms.

In this section, we'll build the `InvoiceForm` component form, and you can see the result in figure 9.5. It has more fields, but visually isn't all that different from the last form. Let's start by building the entire form for our `InvoiceForm` component. We already have the markup ready to go, so we need to define it for Angular.

9.3.1 *Defining your form*

The first step is to define the form for our invoice, and this is done to ensure that the controller is aware of all of the aspects of the form. This will define a separate model that exists just for the form. When we load the invoice data from the API, we'll load it into the form, rather than directly bind the form to it like we saw with `NgModel`.

Angular provides a service called `FormBuilder`, which is a helpful tool to build a reactive form. We'll use this to build the description of our form. It lets us define each of the form controls and any validations we want to apply to them.

We'll be editing the `InvoiceForm` component in this section, so start by opening `src/app/invoice-form/invoice-form.component.ts` and update the constructor like you see in listing 9.7. This only includes the top portion of the file—to focus on the changing pieces, which are in bold.

Figure 9.5 Invoice form with more controls, built in reactive style

Listing 9.7 Using FormBuilder to define the form

```
export class InvoiceFormComponent implements OnInit {
  invoiceForm: FormGroup;
  invoice: Invoice;
  customer: Customer;
  customers: Customer[];
  total = 0;

  constructor(
    private loadingService: TdLoadingService,
    private invoicesService: InvoicesService,
    private router: Router,
    private dialogService: TdDialogService,
    private customersService: CustomersService,
    private formBuilder: FormBuilder,
    private route: ActivatedRoute) {

    this.invoiceForm = this.formBuilder.group({
      id: [''],
      service: ['', Validators.required],
      customerId: ['', Validators.required],
      rate: ['', Validators.required],
      hours: ['', Validators.required],
      date: ['', Validators.required],
      paid: ['']
    });
  }
}
```

Creates a property to hold the resulting form

Defines the form by creating a group

Defines a property that has a validation

Defines a property without validation

To begin, we set a property on the controller to hold our form. It's of the `FormGroup` type, which is an object designed to hold various form controls together. Then inside of the constructor, we'll use the `FormBuilder` service to build a group of controls.

It accepts an object that contains properties with the name of the control set to an array that holds at least one value. The first value is the value it should hold, which we're defaulting to empty for all of them. For some properties, we only define the default value. For other properties, we can add additional items to the array that must be validator functions. We'll create a custom one in a little bit, but for now we're assigning the required validation to each.

That's all we need to do to define our form. But it will always be a blank form, so when we're editing a record we need to load the data into the form. We do this in the `OnInit` lifecycle hook where we load the data. In the following listing, you can see the snippet for the data loading and add the bolded line that sets the form state based on the data.

Listing 9.8 Setting the form state

```
this.route.params.map((params: Params) => params.invoiceId).
  subscribe(invoiceId => {
    if (invoiceId) {
      this.invoicesService.get<Invoice>(invoiceId).subscribe(invoice => {
        this.invoiceForm.setValue(invoice);
        this.invoice = invoice;
        this.loadingService.resolve('invoice');
      });
    } else {
      this.invoice = new Invoice();
      this.loadingService.resolve('invoice');
    }
  });
```

Use `setValue` to update the form state.

The `invoiceForm` has a `setValue` method, which takes a data model and sets properties based on that. Otherwise, it's a new form, and the default values were already declared earlier in the controller when we defined the form. In the case where we're editing and have an existing invoice, it gets set into the form after it's been loaded from the API.

Now we need to update our template so the form controls are aware of this form and its data.

9.3.2 Implementing the template

The form controls in our template are currently unaware of our reactive form, and this step is about linking the form controls in the template and form controls defined in the controller. Form controls in a template exist like a normal HTML form by default. But for this all to work right, they need to know about the form and its current state so they can display properly.

The `InvoiceForm` template has a couple of UI components from Material Design: a date picker and a slide toggle. These act like normal form elements, and you can learn more about them in the documentation.

Much as we used `NgModel` to link a form control to the form, we'll use a different directive called `FormGroupName`. This will indicate which form control should be bound into that element, based on the name provided when we built the form.

Open `src/app/invoice-form/invoice-form.component.html` and make the additions to the form controls, as you see in bold in the following listing, to wire up the controls.

Listing 9.9 InvoiceForm template with form controls

```

<div *tdLoading="'invoice'">
  <form *ngIf="invoice" [formGroup]="invoiceForm">
    <md-card>
      <md-card-header>Edit Invoice</md-card-header>
      <md-card-content>
        <md-input-container>
          <input name="service" mdInput type="text" placeholder="Service"
formControlName="service">
        </md-input-container>
        <md-input-container>
          <input mdInput [mdDatepicker]="picker" placeholder="Choose a date"
formControlName="date">
          <button type="button" mdSuffix [mdDatepickerToggle]="picker"></
button>
        </md-input-container>
        <md-datepicker #picker></md-datepicker>
        <md-input-container>
          <input name="hours" mdInput type="number" placeholder="Hours"
formControlName="hours">
        </md-input-container>
        <md-input-container>
          <input name="rate" mdInput type="number" placeholder="Rate"
formControlName="rate">
        </md-input-container>
        <div>
          <md-select name="customerId" placeholder="Customer"
formControlName="customerId">
            <md-option [value]="customer.id" *ngFor="let customer of
customers">{{customer?.name}}</md-option>
          </md-select>
        </div>
        <div class="toggler">
          <md-slide-toggle formControlName="paid">Paid</md-slide-toggle>
        </div>
        <div class="total">
          Total: {{total | currency:'USD':true:'.2'}}
        </div>
      </md-card-content>
      <md-card-actions>
        <button type="button" md-button>Delete</button>
        <button type="button" md-button>Cancel</button>
        <button type="submit" md-raised-button color="primary">Save</button>
      </md-card-actions>
    </md-card>
  </form>
</div>

```

Defines the form group on the form element

Adds a `formControlName` that maps to the control name

Material Design date picker works with a normal input field.

Slide toggle has a Boolean state.

The first step is to use the `FormGroup` directive to bind the form we declared to the form element. If you miss this step, the form won't know about the model you defined. Then we just linked the form controls with the name used when we built the form, and at this point the form will now render properly. We'll have to work out the details of saving in a little bit, but otherwise it's a fully functional form.

Now I think it would be nice for us to display the invoice total in the page so users know the invoice total based on the rate and hours input. We can do this by observing form controls, so let's see how we can use that.

9.3.3 Watching changes

Unlike in template-driven forms, our reactive form controller has the source of truth for the form state, and it gives us the ability to observe a form or a single control for changes. This lets us run logic that might be useful, such as validation or saving progress.

In our case, we want to display the total invoice amount at the bottom, and that requires multiplying the hours and rate. Each form control exposes an observable that we can use to subscribe to changes, and we'll use it to get both hours and rate values.

The template already has a place for the total at the bottom, but it shows 0 all the time. Although we could try to do math directly in the interpolation binding, it gets a little bit messy and harder to test. We'd rather handle this in the controller.

Using the form, we can get a specific control using `invoiceForm.get('hours')`. You pass a string that's the name of the form control, and you get the instance of that control. This instance provides a number of properties and capabilities, one of which is the `valueChanges` observable.

Let's make this work by adding a little bit to the end of the `OnInit` method. You can see the snippet to add here in the following listing.

Listing 9.10 Observing state changes in the form

```
Observable.combineLatest(  
  this.invoiceForm.get('rate').valueChanges,  
  this.invoiceForm.get('hours').valueChanges  
)  
.subscribe([rate = 0, hours = 0]) => {  
  this.total = rate * hours;  
});
```

This snippet might be new to you, but we're using the `combineLatest` operator from RxJS. This operator takes two observables, which are references to the stream of value changes of the rate and hours controls, and merges them into one. We can then get the latest values from the stream and multiply them to get the current total.

Imagine you had more complex math here, such as adding in taxes, or perhaps there was another value to plug in. Doing math in the interpolation binding directly would quickly get out of hand, and this provides you direct access to run calculations when values change. This is also a pattern of reactive, because in this case you're reacting to changes in the form state and updating the total.

When you use `invoiceForm.get('rate')`, you're also able to access the same properties from table 9.1 (form control status properties). You can check whether the control is valid, pristine, touched, or what errors exist. This might be helpful for you to do additional validation or checks.

We can also implement our own validator functions as we did before and see how to plug them into the form.

9.3.4 Custom validators with reactive forms

Previously, when we implemented custom validation, we created both a validation function and a directive. With reactive forms, we only need to create the validation function and then add it into the form when we create it with FormBuilder.

We'll update our validation messages as well to use the validation rules we defined, as you see in figure 9.6.

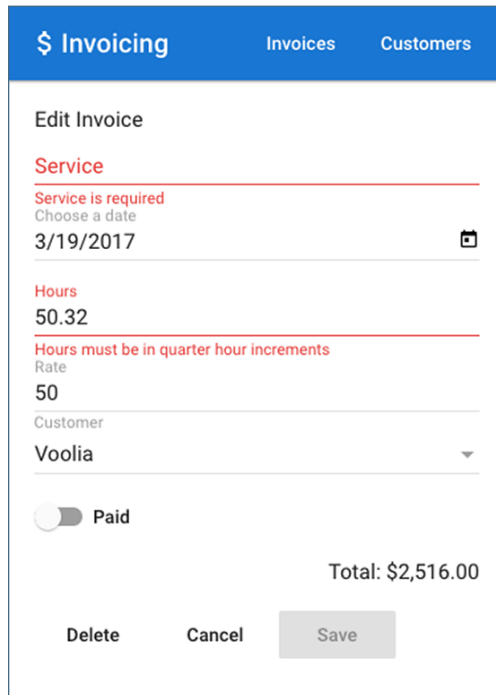
The screenshot shows a mobile application interface for editing an invoice. At the top is a blue header with a dollar sign icon, the title 'Invoicing', and two tabs: 'Invoices' and 'Customers'. Below the header, the title 'Edit Invoice' is displayed. The form contains several fields: 'Service' with a red error message 'Service is required' and a sub-message 'Choose a date'; a date field showing '3/19/2017' with a calendar icon; 'Hours' with a value of '50.32' and a red error message 'Hours must be in quarter hour increments'; a 'Rate' field with a value of '50'; a 'Customer' dropdown menu currently showing 'Voolia'; and a 'Paid' toggle switch which is currently turned off. At the bottom right, the 'Total' is calculated as '\$2,516.00'. At the very bottom are three buttons: 'Delete', 'Cancel', and 'Save'.

Figure 9.6 Validation rules in the invoice form

Imagine our invoicing application had the restriction that hours had to be always rounded to the quarter hour—like 1 hour, 1.25, 1.5, 1.75, or 2. It should not allow values like 1.1 or 1.7. This is fairly common when invoicing by time, and the way we enforce this is to validate the hours input and see if it's valid by quarter hour.

We'll build a validator function like we did previously, but we won't have to wrap it up in a directive. Start by making a new file at `src/app/validators/hours.validator.ts`, and add the code from the following listing to it.

Listing 9.11 Hour validator

Directly exports the validator function

```
import { AbstractControl, ValidatorFn } from '@angular/forms';
export function HoursValidator(control: AbstractControl) : { [key: string]:
  any } {
  return (Number.isInteger(control.value * 4)) ? null : { hours: true };
}
```

Determines if the number is a valid figure and returns validation

This is very succinct, but in contrast to the previous validator, we're directly exporting the validation function. When we created a custom validator earlier for a directive, we needed a function to return a validator function, whereas here we export the validator function directly. When the validator function runs, it multiplies the value by 4 and checks if it is an integer. That means any valid hourly increment will multiply an integer by 4 and return null for valid. Otherwise, it returns the object with the key and Boolean.

Now we need to make our form aware of this validation function, and that's done when we construct the form using FormBuilder. In the component controller, update the form definition like you see in the following listing. You'll need to import the HoursValidator function into the file.

Listing 9.12 Using HoursValidator

```
this.invoiceForm = this.formBuilder.group({
  id: [''],
  service: ['', Validators.required],
  customerId: ['', Validators.required],
  rate: ['', Validators.required],
  hours: ['', [Validators.required, HoursValidator]],
  date: ['', Validators.required],
  paid: ['']
});
```

Adds custom validator to the control

Because we're directly constructing the form, we just need to pass the custom validation function into the control. Notice how the hours control also now has an array for the second item in the array. That's because if you have multiple validators, they need to be grouped here. The form control takes the default value, synchronous validators, and asynchronous validators as a third array item.

We haven't looked at async validators, but the only difference is that they might take a moment to run. Imagine you needed a validator that checked whether a username was already taken; that probably requires making an API call. The only difference when you implement an async validator is that you need to return a promise or observable, and Angular handles it.

We'd also like to show validation errors in the template, so we'll need to add the same type of error messages we saw earlier. But the way we access the form elements to check their validity is slightly different.

Open the template again and update the fields with error messages, as you see in the following listing.

Listing 9.13 Validation messages

```

<md-card-content>
  <md-input-container>
    <input name="service" mdInput type="text" placeholder="Service"
      formControlName="service">
    <md-error *ngIf="invoiceForm.get('service').touched && invoiceForm.
      get('service').invalid"> ← Use invoiceForm to get the
      Service is required          form control reference.
    </md-error>
  </md-input-container>
  <md-input-container>
    <input mdInput [mdDatepicker]="picker" placeholder="Choose a date"
      formControlName="date">
    <button type="button" mdSuffix [mdDatepickerToggle]="picker"></button>
    <md-error *ngIf="invoiceForm.get('date').touched && invoiceForm.
      get('date').invalid">
      Date is required
    </md-error>
  </md-input-container>
  <md-datepicker #picker></md-datepicker>
  <md-input-container>
    <input name="hours" mdInput type="number" placeholder="Hours"
      formControlName="hours">
    <md-error *ngIf="invoiceForm.get('hours').touched && invoiceForm.
      get('hours').invalid">
      Hours must be in quarter hour increments
    </md-error>
  </md-input-container>
  <md-input-container>
    <input name="rate" mdInput type="number" placeholder="Rate"
      formControlName="rate">
    <md-error *ngIf="invoiceForm.get('rate').touched && invoiceForm.
      get('rate').invalid">
      Hourly rate is required
    </md-error>
  </md-input-container>
  <div>
    <md-select name="customerId" placeholder="Customer"
      formControlName="customerId">
      <md-option [value]="customer.id" *ngFor="let customer of
        customers">{{customer?.name}}</md-option>
    </md-select>
  </div>
  <div class="togglers">
    <md-slide-toggle formControlName="paid">Paid</md-slide-toggle>
  </div>
  <div class="total">
    Total: {{total | currency:'USD':true:'.2'}}
  </div>
</md-card-content>

```

At time of writing,
select did not
support validation.

Here we've added the same `MdError` to display errors, except we use `invoiceForm.get('rate')` to access the form control. The same properties from the earlier table are still available to you, but instead of having a local template variable to get a reference to it, we reference it from the form itself.

Now that we have the form validated as we would like, we need to be able to submit it. Let's see how that's done with reactive forms now.

9.3.5 Handling submit or cancel events

The final step is to submit the form when it's ready. The steps are almost identical, except we manage the data in a different way before we submit it to the service. The `NgSubmit` event binding is still available for us to capture submit events to handle, so we'll use that again.

Open the `InvoiceForm` component template again and update the form element like you see here in bold:

```
<form *ngIf="invoice" [formGroup]="invoiceForm" (ngSubmit)="save()">
```

While you have the template open, let's also wire up the buttons at the bottom. Add the bolded parts to your buttons:

```
<md-card-actions>
  <button type="button" md-button (click)="delete()" *ngIf="invoice.
    id">Delete</button>
  <button type="button" md-button (click)="cancel()">Cancel</button>
  <button type="submit" md-raised-button color="primary"
    [disabled]="invoiceForm.invalid">Save</button>
</md-card-actions>
```

Here we're implementing the click handlers on the delete and cancel buttons, and also disabling the save button unless the form is valid. Notice how we're using the `InvoiceForm` properties to determine the form state, similar to how we used `NgForm` with template-driven forms.

The last step is to update the save method in the controller so it gets its data from the form. Because the data was bound into the form when we loaded the component, we need to extract it back out before we save. Update the save method as you see here in bold:

```
save() {
  if (this.invoice.id) {
    this.invoicesService.update<Invoice>(this.invoice.id, this.invoiceForm.
      value).subscribe(response => {
      this.viewInvoice(response.id);
    });
  } else {
    this.invoicesService.create<Invoice>(this.invoiceForm.value).
      subscribe(response => {
        this.viewInvoice(response.id);
      });
  }
}
```

You can see that when we need to get the data back out of the form, we can look at the `invoiceForm.value` property. This gives us an object representing the same form model with the values for each field. We pass this into the service to either create or update a record and see our values being saved correctly.

We're now finished with our invoice form, and you can see both the controller and template in listings 9.14 and 9.15 to ensure you have everything correct.

Listing 9.14 InvoiceForm component controller

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router } from '@angular/router';
import { TdLoadingService, TdDialogService } from '@covalent/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { InvoicesService, Invoice, CustomersService, Customer } from '@aia/
  services';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/combineLatest';
import { HoursValidator } from '../validators/hours.validator';

@Component({
  selector: 'app-invoice-form',
  templateUrl: './invoice-form.component.html',
  styleUrls: ['./invoice-form.component.css']
})
export class InvoiceFormComponent implements OnInit {
  invoiceForm: FormGroup;
  invoice: Invoice;
  customer: Customer;
  customers: Customer[];
  total = 0;

  constructor(
    private loadingService: TdLoadingService,
    private invoicesService: InvoicesService,
    private router: Router,
    private dialogService: TdDialogService,
    private customersService: CustomersService,
    private formBuilder: FormBuilder,
    private route: ActivatedRoute) {

    this.invoiceForm = this.formBuilder.group({
      id: [''],
      service: ['', Validators.required],
      customerId: ['', Validators.required],
      rate: ['', Validators.required],
      hours: ['', [Validators.required, HoursValidator]],
      date: ['', Validators.required],
      paid: ['']
    });
  }

  ngOnInit() {
    this.loadingService.register('invoice');
    this.loadingService.register('customers');

    this.customersService.query().subscribe(customers => {
      this.customers = customers;
    });
  }
}
```

```

        this.loadingService.resolve('customers');
    });

    this.route.params.map((params: Params) => params.invoiceId).
    subscribe(invoiceId => {
        if (invoiceId) {
            this.invoicesService.get<Invoice>(invoiceId).subscribe(invoice => {
                this.invoiceForm.setValue(invoice);
                this.invoice = invoice;
                this.loadingService.resolve('invoice');
            });
        } else {
            this.invoice = new Invoice();
            this.loadingService.resolve('invoice');
        }
    });

    Observable.combineLatest(
        this.invoiceForm.get('rate').valueChanges,
        this.invoiceForm.get('hours').valueChanges
    ).subscribe(([rate = 0, hours = 0]) => {
        this.total = rate * hours;
    });
}

save() {
    if (this.invoice.id) {
        this.invoicesService.update<Invoice>(this.invoice.id, this.invoiceForm.
        value).subscribe(response => {
            this.viewInvoice(response.id);
        });
    } else {
        this.invoicesService.create<Invoice>(this.invoiceForm.value).
        subscribe(response => {
            this.viewInvoice(response.id);
        });
    }
}

delete() {
    this.dialogService.openConfirm({
        message: 'Are you sure you want to delete this invoice?',
        title: 'Confirm',
        acceptButton: 'Delete'
    }).afterClosed().subscribe((accept: boolean) => {
        if (accept) {
            this.loadingService.register('invoice');
            this.invoicesService.delete(this.invoice.id).subscribe(response => {
                this.loadingService.resolve('invoice');
                this.invoice.id = null;
                this.cancel();
            });
        }
    });
}

```

```

    });
  }

  cancel() {
    if (this.invoice.id) {
      this.router.navigate(['/invoices', this.invoice.id]);
    } else {
      this.router.navigateByUrl('/invoices');
    }
  }

  private viewInvoice(id: number) {
    this.router.navigate(['/invoices', id]);
  }
}

```

Listing 9.15 InvoiceForm component template

```

<div *tdLoading="'invoice'">
  <form *ngIf="invoice" [formGroup]="invoiceForm" (ngSubmit)="save()">
    <md-card>
      <md-card-header>Edit Invoice</md-card-header>
      <md-card-content>
        <md-input-container>
          <input name="service" mdInput type="text" placeholder="Service"
            formControlName="service">
          <md-error *ngIf="invoiceForm.get('service').touched && invoiceForm.
            get('service').invalid">
            Service is required
          </md-error>
        </md-input-container>
        <md-input-container>
          <input mdInput [mdDatepicker]="picker" placeholder="Choose a date"
            formControlName="date">
          <button type="button" mdSuffix [mdDatepickerToggle]="picker"></
            button>
          <md-error *ngIf="invoiceForm.get('date').touched && invoiceForm.
            get('date').invalid">
            Date is required
          </md-error>
        </md-input-container>
        <md-datepicker #picker></md-datepicker>
        <md-input-container>
          <input name="hours" mdInput type="number" placeholder="Hours"
            formControlName="hours">
          <md-error *ngIf="invoiceForm.get('hours').touched && invoiceForm.
            get('hours').invalid">
            Hours must be in quarter hour increments
          </md-error>
        </md-input-container>
      </md-card-content>
    </md-card>
  </form>
</div>

```

```

        <input name="rate" mdInput type="number" placeholder="Rate"
formControlName="rate">
        <md-error *ngIf="invoiceForm.get('rate').touched && invoiceForm.
get('rate').invalid">
            Hourly rate is required
        </md-error>
    </md-input-container>
</div>
    <md-select name="customerId" placeholder="Customer"
formControlName="customerId">
        <md-option [value]="customer.id" *ngFor="let customer of
customers">{{customer?.name}}</md-option>
    </md-select>
</div>
<div class="toggler">
    <md-slide-toggle formControlName="paid">Paid</md-slide-toggle>
</div>
<div class="total">
    Total: {{total | currency:'USD':true:'.2'}}
</div>
</md-card-content>
<md-card-actions>
    <button type="button" md-button (click)="delete()" *ngIf="invoice.
id">Delete</button>
    <button type="button" md-button (click)="cancel()">Cancel</button>
    <button type="submit" md-raised-button color="primary"
[disabled]="invoiceForm.invalid">Save</button>
</md-card-actions>
</md-card>
</form>
</div>

```

That covers the majority of what you need to know about both reactive and template-driven forms. There are certainly more minor features that exist for additional cases, but this foundation should get you building forms, and you can learn about other features as you go.

9.3.6 Which form approach is better?

That is a trick question, to my mind, though you probably want a bit more of an explanation. Rather than tell you to use one and never the other, I'll share from my experience why I use both.

Excluding the mechanical differences of the two form libraries, I find the most important aspect is how they approach defining the form. The patterns they employ can work in most situations.

Most of the time I suggest reactive forms. I like the guarantees reactive provides and the way you define the model and let the template react. I prefer my templates to reflect state, not create state. By that I mean how `NgModel` creates the controls for you behind

the scenes and binds data up to the controller. If you need an answer, I would recommend reactive forms, if you really pinned me down.

But you may have noticed this is the first time we've seen reactive forms in the book. Sometimes it's simpler to use `NgModel`, especially when it's a single form field. In simple scenarios, I find template-driven forms to be more approachable with low overhead, but when a form becomes more complex, then I recommend reactive forms.

I think the most important thing is to be consistent in your applications. Although you can mix and match as much as you like, there's a mental drawback to that when you write and test them.

Before I close out the chapter, let's see how to implement your own form controls in cases where your application needs controls that don't exist out of the box or in libraries.

9.4 Custom form controls

There are scenarios where your application requires a different form control that isn't defined in HTML or in a third-party library. All form controls have a few basic requirements, and Angular already implements them for the built-in HTML form elements.

Regardless of whether you use reactive or template-driven forms, there has to be some logic to write up the native HTML element (or custom component) with the forms library. There are essentially two places to track the current value of a form control: in the form and the control. Angular provides the `ControlValueAccessor` interface as a way to implement a custom control that works with forms, which we'll use in conjunction with the Angular Material library components to create our own custom control.

In our application, there are several candidates for creating custom form controls, but we'll be transforming the current hours input field from the invoice form into a custom form control. We'll implement some basic features that make it easier to use, but also encapsulate the internal logic of the control.

As you see in figure 9.7, the hours form field now has several buttons underneath that help you dial in the value by smaller increments. As you change the values, the form element will continue to validate and update the total invoice value at the bottom, as you would expect.

Our first step is to create a new component to house our custom control. To do this, use the CLI as you see here:

```
ng generate component hours-control
```

Once the component is created, open the `src/app/hours-control/hours-control.component.ts` file and replace the contents with what you see in listing 9.16. There's a lot happening in this file, so we'll look at the various pieces closely.

Figure 9.7 New hours custom control that connects with Angular forms

Listing 9.16 HoursControl controller

```
import { Component, forwardRef } from '@angular/core';
import { ControlValueAccessor, NG_VALIDATORS, NG_VALUE_ACCESSOR, FormControl
    } from '@angular/forms';
import { HoursValidator } from '../validators/hours.validator';

@Component({
  selector: 'app-hours-control',
  templateUrl: './hours-control.component.html',
  styleUrls: ['./hours-control.component.css'],
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => HoursControlComponent),
    multi: true
  }, {
    provide: NG_VALIDATORS,
    useExisting: forwardRef(() => HoursControlComponent),
    multi: true
  }]
})
export class HoursControlComponent implements ControlValueAccessor {
  hours = 0;
  validateFn = HoursValidator;
  onChange = (v: any) => {};
```

Declares providers

ControlValueAccessor interface is used by all form controls

Properties to house the value, validation function, and change event


```

update() {
  this.onChange(this.hours);
}

keypress($event) {
  if ($event.key === 'ArrowUp') {
    this.setValue(.25);
  } else if ($event.key === 'ArrowDown') {
    this.setValue(-.25);
  }
}

setValue(change: number) {
  this.hours += change;
  this.update();
}

validate(control: FormControl) {
  return this.validateFn(control);
}

writeValue(value: any) {
  if (value !== undefined) {
    this.hours = value;
  }
}

registerOnChange(fn) {
  this.onChange = fn;
}

registerOnTouched() {}

```

Changes binding to update the form control

Event handler for key press

Method to set value from button clicks

Validation handler

Handles writing a value into the control

Wires up change handler

← Empty method to satisfy interface

There's a lot happening here in a short amount of space, so let's break things down. The `HoursControl` component implements the `ControlValueAccessor` interface, which ensures that your form control is designed to work correctly with Angular forms. It requires that a control implements the three methods found at the end of the controller: `writeValue`, `registerOnChange`, and `registerOnTouched`.

The `writeValue` method is used by Angular to pass a value into the form control from the form itself. This is similar to binding a value into the component, though it works with the form controls like `NgModel`, and it passes the value from the form into the control.

The `registerOnChange` method accepts a function that the form library will pass in that your control needs to call whenever the value changes. It stores this function on the `onChange` property of the controller, and the default noop function is defined so the component compiles correctly. In other words, it gives you a method to call that passes the current form value up to the form.

The `registerOnTouch` method isn't implemented here, but it allows you to accept a method to handle touch events. This might be useful on controls that have some kind

of touch impact, such as a toggle switch. But there isn't much for us to implement for a form control that takes a number input.

In the component metadata, we see some providers are declared. Recall that we did something similar when we created our directive for validation. Here we have to declare two providers—the first is to register this component with `NG_VALUE_ACCESSOR`. This marks this component as a form control and registers it with dependency injection so Angular can access it later. The second is to register the component with `NG_VALIDATORS`. This control has validation internally, so we need to register the control on the validators provider for Angular to access later.

Because the control has a `validate` method, Angular can call this method to determine whether the control is valid or not. This is the same as with creating a `Validator` directive as we did earlier in listing 9.4. In this case, though, we import the `HoursValidator` function and reuse it inside the component.

The rest of the methods are there to handle the internal actions of the control. The `update` method is responsible for calling the change event handler, which will alert the form that the control's internal state value has changed. The `keypress` method is just a nice feature that allows us to bind to the `keyup` event, and if the user pressed up or down arrows, it will increment or decrement the current value by 0.25. Finally, the `setValue` method is called by the row of buttons to add or subtract from the current value.

In summary, this component really has three roles. First, it implements an internal model to track the current value of the control (the number of hours) and allows that value to be manipulated by buttons or keypresses. Second, it provides validation and ensures the number provided is to a quarter of an hour. Third, it wires up the necessary methods for Angular forms to be made aware of the current state of the control.

Next we need to look at the template, so let's go ahead and implement it so we can see everything together. Open `src/app/hours-control/hours-control.component.html` and replace its contents with the code from the following listing.

Listing 9.17 HoursControl template

<p>Shows or hides error messages</p>	<pre><md-input-container> <input name="hours" mdInput type="number" placeholder="Hours" [(ngModel)]="hours" hours (keyup)="keypress(\$event)" #control="ngModel" (change)="update()"> <md-error *ngIf="control.touched && control.invalid"> Hours must be a number in increments of .25 </md-error> </md-input-container> <div layout="row"> <button type="button" md-button flex (click)="setValue(1)">+ 1</button> <button type="button" md-button flex (click)="setValue(.25)">+ .25</button> <button type="button" md-button flex (click)="setValue(-.25)">- .25</button> <button type="button" md-button flex (click)="setValue(-1)">- 1</button> </div></pre>	<p>Binds hours using NgModel, adds directive to validate, and event bindings</p>
---	---	---

In our template, we encapsulate the entire form control that we want to provide, which includes the buttons and the original input box. Because we're still accepting text input, we use a standard input element. But we're also setting up the `NgModel`, a validation directive (which we'll create next), and two event bindings for `keyup` and `change`.

This control has built-in error validation and uses the same Angular Material patterns we saw earlier in the chapter. It shows a message if the control is invalid, and if the control has been focused on. It's nice that the validation messaging is built in, because it doesn't need to be implemented later. If you have the same controls in many places with the same validation, this might be useful. If you want to ensure that this control is more reusable, with different validation types, this might not be ideal.

The last set of elements is the buttons that add or subtract from the current state. When they're clicked, the internal hours model is updated, and the form is alerted to the change as well.

There's a bit of CSS that we need to add for the control to look correct, so open `src/app/hours-control/hours-control.css` and add the code from the following listing.

Listing 9.18 HoursControl stylings

```
:host {  
  width: 100%;  
  display: block;  
}  
md-input-container {  
  width: 100%;  
}  
button {  
  padding: 0;  
  min-width: 25%;  
}
```

This makes sure that a few pieces of the control play nicely with the UI library, since we've changed the way it usually expects elements to be laid out.

You probably noticed that we have a validation directive for hours on the input, but we haven't created a directive version of this validator yet. That's simple to do, and we need to do so before we use this control. Create a new directive by running the following command:

```
ng generate directive validators/hours
```

Then open the directive file at `src/app/validators/hours.directive.ts` and replace its contents with the code in the following listing.

Listing 9.19 Hours validation directive

```
import { Directive } from '@angular/core';  
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';  
import { HoursValidator } from './hours.validator';  
  
@Directive({
```

```

selector: '[hours] [ngModel]',
providers: [{ provide: NG_VALIDATORS, useExisting: HoursDirective, multi:
  true }]
})
export class HoursDirective implements Validator {
  private validator = HoursValidator;

  validate(control: AbstractControl): { [key: string]: any } {
    return this.validator(control);
  }
}

```

Ensures selector applies to elements with the hour and NgModel attributes

This directive looks almost identical to the one we created earlier, except it references the `HoursValidator` function. I recommend reviewing the details from listing 9.3 for specifics if you have any questions.

Now we have everything we need to use our new control. This control is meant to be used in the `InvoiceForm` component, so open the template found at `src/app/invoice-form/invoice-form.component.html` and replace the existing hours input element with our newly created form control, as you see here in bold in the snippet of the whole template:

```

<md-datepicker #picker></md-datepicker>
<app-hours-control formControlName="hours"></app-hours-control>
<md-input-container>
  <input name="rate" mdInput type="number" placeholder="Rate"
    formControlName="rate">
</md-input-container>

```

Because this is a custom form control, we can use it with reactive forms or template-driven forms without issue. Congratulations! You've created your own control and can now make as many as you want.

But wait—there are a couple of caveats to building your own controls, and to this particular example. Custom controls seem like a great idea, but they can also be a lot of work to build properly. For example, does your custom control work well on mobile or touch devices? Does it have proper support for screen readers and other accessibility requirements? Does it work in multiple applications or is it too custom for your application? These are important questions to ask, and also to verify whether your controls work for the largest set of users. One of the major reasons I advocate using an existing UI library is that the good libraries will have solved these issues ahead of time for you.

Before going off to build a custom control, see if you can think clearly about the user experience and determine whether an existing control could be used instead of a new one. Users tend to struggle more with custom form elements that they haven't seen before, so it can be very practical to adjust the application slightly so it can use already existing controls before you make a new one.

Because this chapter is using a specific UI library, I've implemented the form controls in a way that fits with that library. Therefore, it's limited to being used only with Angular Material, which may limit the use of your control. On the other hand, if you can expect to always use Angular Material (or the UI library of choice), then the custom control may be saving you a lot of repetition.

At the time of writing, the Angular Material library doesn't support creating your own form controls that work nicely with the input container. (See <https://github.com/angular/material2/issues/4672>.) This is why I ultimately encapsulated the entire form control and surrounding markup. It makes the example more verbose than you might need, so you should consider how to simplify your form controls if possible.

This example uses a standard `input` element inside, which is why `NgModel` was used, but in many custom form controls you may not have an input, so you wouldn't use `NgModel`. In those cases, you simply make sure that as the control state changes (such as a toggler that goes from true to false), you call the change handler so the form knows the state changes.

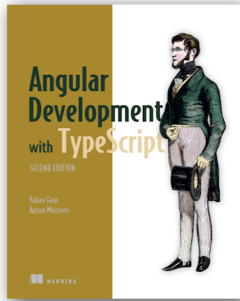
That wraps up forms, both reactive and template-driven, as well as creating your own controls. Forms are very important to most applications, and you should now have the tools to craft feature-rich and usable forms.

Summary

We've built two forms, in both the reactive and template-driven styles in this chapter. Along the way we also managed to learn about most of what forms have to offer. Here's a brief summary of the key takeaways:

- Template-driven forms define the form using `NgModel` on form controls.
- You can apply normal HTML validation attributes, and `NgModel` will automatically try to validate based on those rules.
- Custom validation is possible through a custom validator function and directive, which gets registered with the built-in list of validators.
- The `NgForm` directive, though it can be transparent, exposes features to help manage submit events and overall form validation inspection.
- Reactive forms are different in that you define the form model in the controller and link form controls using `FormControlName`.
- You can observe the changes of a form control with reactive forms and run logic every time a new value is emitted.
- Reactive forms declare validation in the controller form definition, and creating custom validations is easier because they don't require a directive.
- Ultimately, both form patterns are available to you. I tend to use reactive forms, especially as the form gets more complex.
- Creating a new form control requires implementing the `ControlValueAccessor` methods and registering it with the controls provider.

RELATED MANNING TITLES



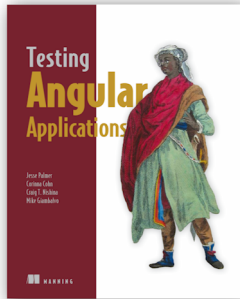
Angular Development with Typescript, Second Edition

by Yakov Fain and Anton Moiseev

ISBN: 9781617295348

475 pages, \$49.99

June 2018



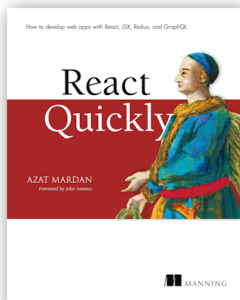
Testing Angular Applications

by Jesse Palmer, Corinna Cohn, Michael Giambalvo, Craig Nishina

ISBN: 9781617293641

235 pages, \$44.99

April 2018



React Quickly

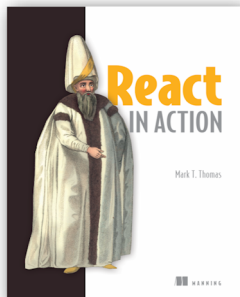
Painless web apps with React, JSX, Redux, and GraphQL

by Azat Mardan

ISBN: 9781617293344

528 pages, \$49.99

August 2017



React in Action

by Mark Tielens Thomas

ISBN: 9781617293856

300 pages, \$44.99

March 2018

For ordering information go to www.manning.com

Angular IN ACTION

Jeremy Wilken



Angular makes it easy to deliver amazing web apps. This powerful JavaScript platform provides the tooling to manage your project, libraries to help handle most common tasks, and a rich ecosystem full of third-party capabilities to add as needed. Built with developer productivity in mind, Angular boosts your efficiency with a modern component architecture, well-constructed APIs, and a rich community.

Angular in Action teaches you everything you need to build production-ready Angular applications. You'll start coding immediately, as you move from the basics to advanced techniques like testing, dependency injection, and performance tuning. Along the way, you'll take advantage of TypeScript and ES2015 features to write clear, well-architected code. Thoroughly practical and packed with tricks and tips, this hands-on tutorial is perfect for web devs ready to build web applications that can handle whatever you throw at them.

What's Inside

- Spinning up your first Angular application
- A complete tour of Angular's features
- Comprehensive example projects
- Testing and debugging
- Managing large applications

Written for web developers comfortable with JavaScript, HTML, and CSS.

Jeremy Wilken is a Google Developer Expert in Angular, Web Technologies, and Google Assistant. He has many years of experience building web applications and libraries for eBay, Teradata, and VMware.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/angular-in-action

“A comprehensive introduction to the world of Angular with great code samples to help readers get started.”

—Tanya Wilke, Sanlam

“Detailed and clear explanations; lots of useful real-world examples.”

—Harsh Raval, Zymr Systems

“You can never have enough Angular references ... this one will float to the top.”

—Michael A. Angelo
US Department of Agriculture
Forestry Services

“The bible for Angular!”

—Phily Austria
Faraday Future



\$44.99 / Can \$59.99 [INCLUDING eBook]

ISBN-13: 978-1-61729-331-3
ISBN-10: 1-61729-331-8



9 781617 293313



5 4 4 9 9