**Chapter 10. Introducing the Forms API**

*This chapter covers*

* Understanding the Angular Forms API
* Working with template-driven forms
* Working with reactive forms

HTML provides basic features for displaying forms, validating entered values, and submitting data to the server. But HTML forms may not be good enough for real-world applications, which need a way to programmatically process the entered data, apply custom validation rules, display user-friendly error messages, transform the format of the entered data, and choose the way data is submitted to the server. For business applications, one of the most important considerations when choosing a web framework is how well it handles forms.

Angular offers rich support for handling forms. It goes beyond regular data binding by treating form fields as first-class citizens and providing fine-grained control over form data. In this chapter, we’ll introduce you two Forms APIs: template-driven and reactive.

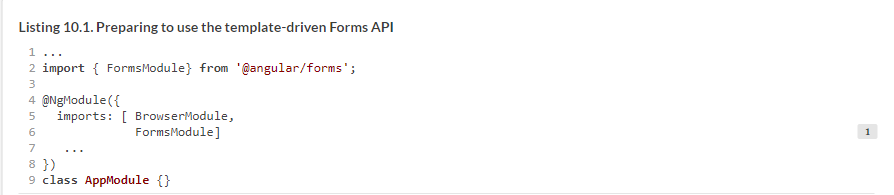
### 10.1. Two Forms APIs

Every Angular-powered form has an underlying model object that stores the form’s data. There are two approaches to working with forms in Angular: *template-driven* and *reactive*. These two approaches are exposed as two different APIs (sets of directives and TypeScript classes).

With the *template-driven* API, forms are fully programmed in the component’s template using directives, and the model object is created implicitly by Angular. The template defines the structure of the form, the format of its fields, and the validation rules. Because you’re limited to HTML syntax while defining the form, the template-driven approach suits only simple forms.

With the reactive API, you explicitly create the model object in TypeScript code and then link the HTML template elements to that model’s properties using special directives. You construct the form model object explicitly using the FormControl, FormGroup, and FormArray classes. In the template-driven approach, you don’t access these classes directly, whereas in the reactive approach, you explicitly create instances of these classes. For non-trivial forms, the reactive approach is a better option.

Both template-driven and reactive APIs need to be explicitly enabled before you start using them. To enable reactive forms, add ReactiveFormsModule from @angular/forms to the imports list of NgModule. For template-driven forms, import FormsModule, as shown in the following listing.



It’s time to discuss both APIs in greater detail.

### 10.2. Template-driven forms

With the template-driven API, you can use only directives in a component’s templates. These directives are included in the FormsModule: NgModel, NgModelGroup, and NgForm. We’ll briefly look at these directives and then apply the template-driven approach to the sample registration form.

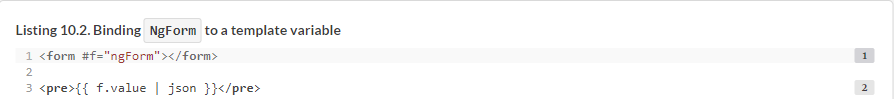
#### 10.2.1. Forms directives

This section briefly describes the three main directives from FormsModule: NgModel, NgModelGroup, and NgForm. We’ll show you how they can be used in the template and highlight their most important features.

##### NgForm

NgForm is the directive that represents the entire form. It’s automatically attached to every <form> element. NgForm implicitly creates an instance of the FormGroup class that represents the model and stores the form’s data (more on FormGroup later in this chapter). NgForm automatically discovers all child HTML elements marked with the NgModel directive and adds their values to the form model object.

You can bind an implicitly created NgForm object to a local template variable so you can access values of the NgForm object inside the template, as shown in the following listing.



The local template variable f points at the instance of NgForm attached to the <form>. Then you can use the f variable to access instance members of the NgForm object. One of them is value, which represents a JavaScript object containing current values of all form fields. You can pass it through the standard json pipe to display the form’s value on the page.

NgForm intercepts the standard HTML form’s submit event and prevents automatic form submission. Instead, it emits the custom ngSubmit event:

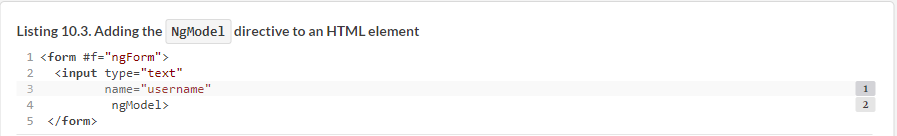


This subscribes to the ngSubmit event using event-binding syntax. The onSubmit handler is a method with an arbitrary name defined in the component, and it’s invoked when the ngSubmit event is emitted. To pass all the form’s values as an argument to this method, use a local template variable (for example, f) to access NgForm’s value property.

##### NgModel

[Section 2.6.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/22#!/book/angular-development-with-typescript-second-edition/chapter-2/ch02lev2sec3) of [chapter 2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/22#!/book/angular-development-with-typescript-second-edition/chapter-2/ch02) discusses how the NgModel directive can be used for two-way data binding. But in the Forms API, NgModel plays a different role: it marks the HTML element that should become a part of the form model.

In the context of the Forms API, NgModel represents a single field on the form. If an HTML element includes ngModel, Angular implicitly creates an instance of the FormControl class that represents the model and stores the fields’ data (more on FormControl later in this chapter). Note that the Forms API doesn’t require a value assigned to ngModel, nor any kind of brackets around this attribute, as you can see in the following listing.



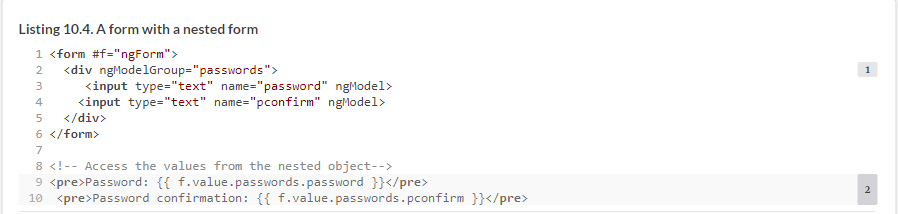
The NgForm.value property points at the JavaScript object that holds the values of all form fields. The value of the field’s name attribute becomes the property name of the corresponding property in the JavaScript object in NgForm.value.

##### NOTE

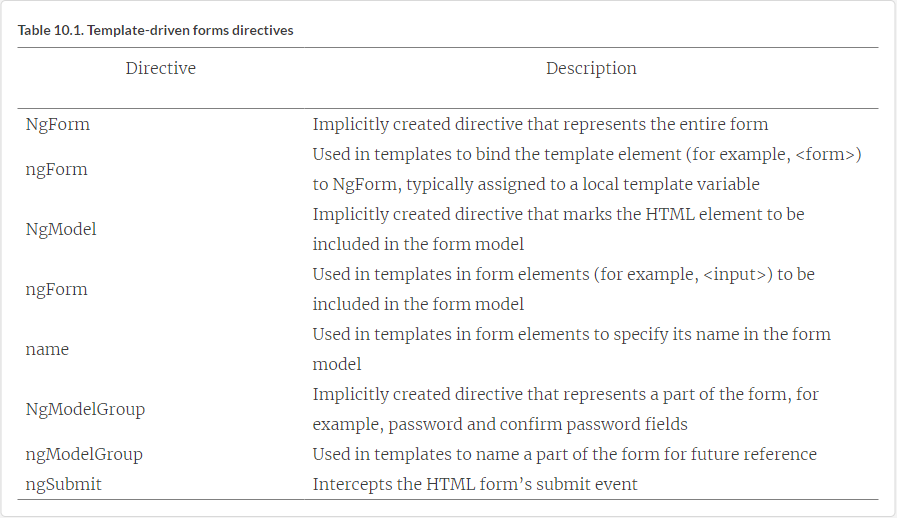
Although the names of the classes that implement form directives are capitalized, their names should start with a lowercase letter in templates (for example, NgForm versus ngForm).

##### NgModelGroup

NgModelGroup represents a part of the form and allows you to group form fields together. Like NgForm, it implicitly creates an instance of the FormGroup class. NgModelGroup creates a nested object inside the object stored in NgForm.value. All the child fields of NgModelGroup become properties of the nested object, as you can see in the following listing.



[Table 10.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/32#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10table01) contains a summary of directives used in template-driven forms.



#### 10.2.2. Applying the template-driven API to HTML forms

Let’s create a simple user registration form, applying the template-driven Forms API. You’ll also add validation logic and enable programmatic handling of the ngSubmit event. You’ll start by creating the template, and then you’ll work on the TypeScript part. First, modify the standard HTML <form> element to match the following listing.



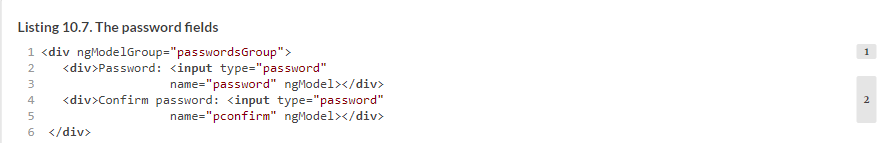
A local template variable f points at the NgForm object attached to the <form> element in the DOM. You need this variable to access the form’s properties (such as value and valid), and to check whether the form has errors in a specific field.

The ngSubmit event is emitted by NgForm. You don’t want to listen to the standard submit event because NgForm intercepts the submit event and stops its propagation. This prevents the form from being automatically submitted to the server, resulting in a page reload. Instead, NgForm emits its own ngSubmit event.

The onSubmit() method will handle the ngSubmit event, and you’ll add this method to the component’s class. It takes one argument—the form’s value—which is a plain JavaScript object that keeps the values of all the fields on the form. Next, add the username and ssn fields (SSN is a unique ID that every US resident has).



Now you’ll add the fields to enter and confirm the password. Because these fields are related and represent the same value, it’s natural to combine them into a group. Wrapping both passwords into a single object is useful for implementing a validator that checks whether both passwords are the same, as you can see in the following listing (you’ll see how to do it in section 11.3.1 in [chapter 11](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/42#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11)).



The Submit button remains the same as in the plain HTML version of the form:



Now that you’re done with the template, you’ll use it in a component, as shown in the following listing.

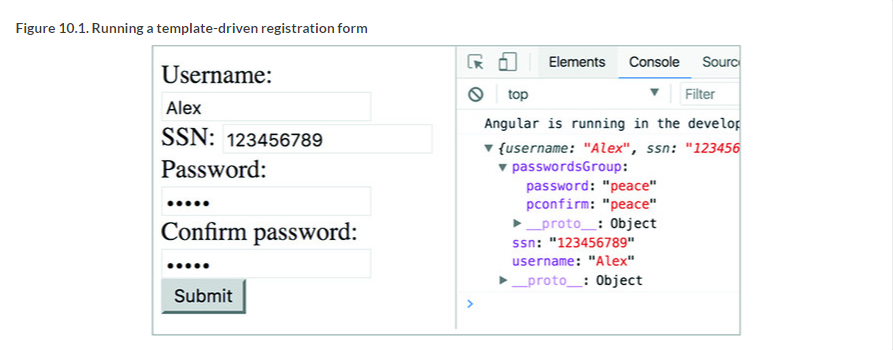


The onSubmit() event handler takes a single argument: the form model’s value, an object containing the field’s values. As you can see, the handler doesn’t use an Angular-specific API. Depending on the validity flag on the model, you can decide whether to post the formData to the server. In this example, you print it to the console.

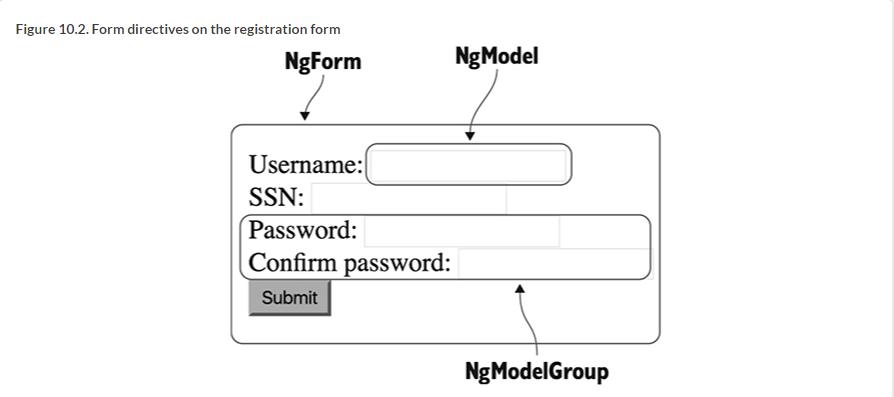
To see this app in action, run npm install in the directory form-samples, and then run the following command:



Fill out the form and click the Submit button. The value of the model object will be printed in the browser’s console, as shown in [figure 10.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/51#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10fig01).



[Figure 10.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/54#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10fig02) displays a sample registration form with the form directives applied to it. Each form directive is circled so you can see what makes up the form. The complete running application that illustrates how to use form directives is located in the template-driven directory.



##### NOTE

Source code for this chapter can be found at <https://github.com/Farata/angulartypescript> and [www.manning.com/books/angular-development-with-typescript-second-edition](http://www.manning.com/books/angular-development-with-typescript-second-edition).

### 10.3. Reactive forms

Creating a reactive form requires more steps than creating a template-driven one. In short, you need to perform the following steps:

**1**.  Import ReactiveFormsModule in the NgModule() where your component is declared.

**2**.  In your TypeScript code, create an instance of the model object FormGroup to store the form’s values.

**3**.  Create an HTML form template, adding reactive directives.

**4**.  Use the instance of the FormGroup to access the form’s values.

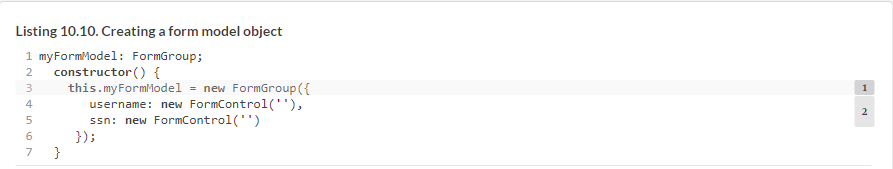
Adding ReactiveFormsModule to the @NgModule() decorator is a trivial operation.



Now let’s talk about how to create a form model.

#### 10.3.1. Form model

The form model is a data structure that holds the form’s data. It can be constructed from FormControl, FormGroup, and FormArray classes. For example, the following listing declares a class property of type FormGroup and initializes it with a new object that will contain instances of the form controls for your form.

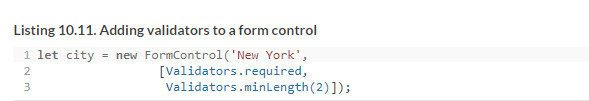


##### FormControl

FormControl is an atomic form unit. Most often, it corresponds to a single <input> element, but it can also represent a more complex UI component like a calendar or a slider. A FormControl instance stores the current value of the HTML element it corresponds to, the element’s validity status, and whether it’s been modified. Here’s how you can create a control passing its initial value as the first argument of the constructor:



You can also create a FormControl attaching one or more built-in or custom validators. [Chapter 11](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/72#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11) covers form validation, but the following code listing shows how to attach two built-in Angular validators to a form control.

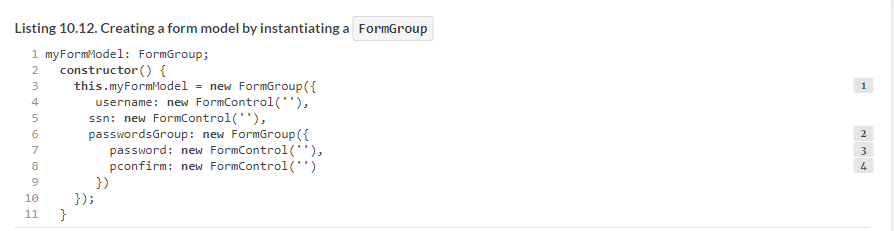


##### NOTE

You can add a formControl directive to a template without wrapping it inside an NgForm directive—for example, it can be used with a standalone <input> element. You can find such an example in [chapter 6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/75#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06) in [section 6.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/75#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06lev1sec3).

##### FormGroup

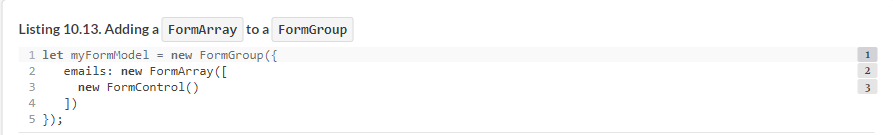
FormGroup is a collection of FormControl objects and represents either the entire form or its part. FormGroup aggregates the values and validity of each FormControl in the group. If one of the controls in a group is invalid, the entire group becomes invalid. The following listing shows the use of FormGroup to represent the form or part of it.



In section 10.3.6, you’ll see a simplified syntax for creating form models with nesting.

##### FormArray

When you need to programmatically add (or remove) controls to a form, use FormArray. It’s similar to FormGroup but has a length variable. Whereas FormGroup represents an entire form or a fixed subset of a form’s fields, FormArray usually represents a collection of form controls that can grow or shrink. For example, you could use FormArray to allow users to enter an arbitrary number of emails. The following listing shows a model that would back such a form.



In [section 10.3.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/86#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10lev2sec6), we’ll show you an app that allows the user add more email controls during runtime, to allow users to enter multiple emails.

#### 10.3.2. Reactive directives

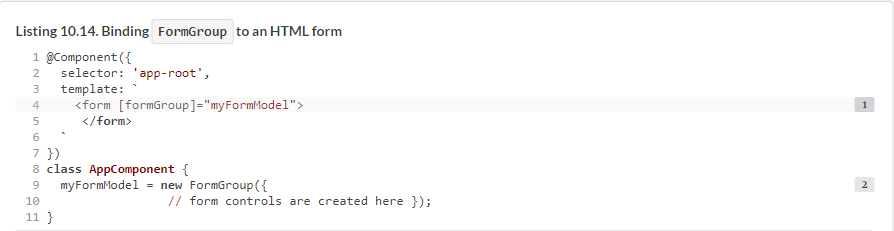
The reactive approach also requires you to use directives in component templates, but these directives are different compared to ones from the template-driven API. The reactive directives come with ReactiveFormsModule and are prefixed with form—for example, formGroup (note the small f).

You can’t create a local template variable in the template that binds to a reactive directive, and it’s not needed. In template-driven forms, the model is created implicitly, and local template variables would give you access to the model or its properties. In reactive forms, you explicitly create a model in TypeScript and don’t need to access the model in the component template.

The reactive directives formGroup and formControl bind a DOM element to the model object using property-binding syntax with square brackets. The directives that link a DOM element to a TypeScript model’s properties by name are formGroupName, formControlName, and formArrayName. They can only be used inside the HTML element marked with the formGroup directive. Let’s look at the form directives.

##### formGroup

The formGroup directive binds an instance of the FormGroup class that represents the entire form model to a top-level form’s DOM element, usually a <form>. In the component template, use formGroup with a lowercase f, and in TypeScript, create an instance of the FormGroup class with a capital F. All directives attached to the child DOM elements will be in the scope of formGroup and can link model instances by name. To use the formGroup directive in a template, you need to first create an instance of FormGroup in the TypeScript code of a component, as shown in the following listing.



##### formGroupName

The formGroupName directive can be used to link nested groups in a form within templates. Use formGroupName in the scope of a parent formGroup directive to link its child FormGroup instances. The next listing shows how you’d define a form model that uses formGroupName.



##### formControlName

formControlName must be used in the scope of the formGroup directive. It links an individual FormControl instance to a DOM element. Let’s continue adding code to the example of the dateRange model from the previous section. The component and form model remain the same. You only need to add HTML elements with the formControlName directive to complete the template.



As in the formGroupName directive, you specify the name of a FormControl you want to link to the DOM element. Again, these are the names you chose when defining the form model.

##### formControl

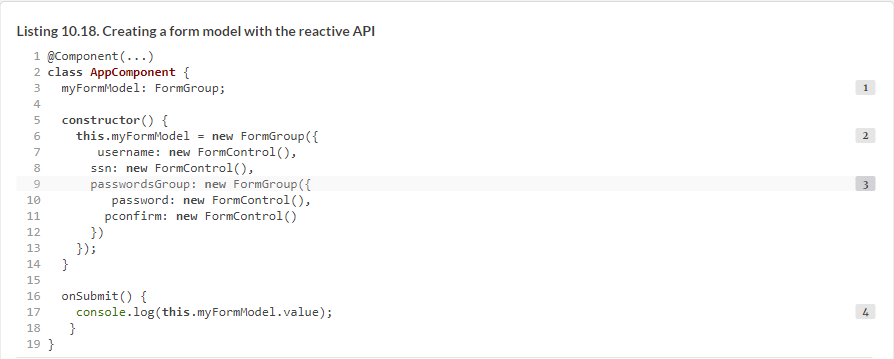
The formControl directive is used with individual form controls or single-control forms, when you don’t want to create a form model with FormGroup but still want to use Forms API features like validation and the reactive behavior provided by the FormControl.valueChanges property. You saw it in the weather app in [section 6.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/97#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06lev1sec4) of [chapter 6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/97#!/book/angular-development-with-typescript-second-edition/chapter-6/ch06). The following listing shows the essence of that example from the Forms API perspective.



You could use ngModel (as [section 2.6.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/103#!/book/angular-development-with-typescript-second-edition/chapter-2/ch02lev2sec3) of [chapter 2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/103#!/book/angular-development-with-typescript-second-edition/chapter-2/ch02)) to sync the value entered by the user with the component’s property; but because you’re using the Forms API, you can use its reactive features. In [listing 10.18](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/103#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10ex18), you apply two RxJS operators to the observable returned by the valueChanges property to improve the user experience.

#### 10.3.3. Applying the reactive API to HTML forms

Let’s refactor the user registration form from [section 10.2.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/103#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10lev2sec2) to use the reactive Forms API. The following listing uses the reactive Forms API, starting by creating a model object in TypeScript.



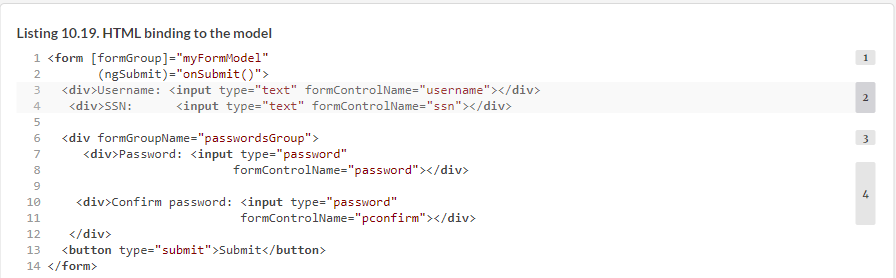
The myFormModel property holds a reference to the FormGroup instance. You’ll bind this property to the formGroup directive in the component template. The myFormModel property is initialized by instantiating a model class. The names you give to form controls in the parent FormGroup will be used in the component’s template to link the model to the DOM elements with the formControlName and formGroupName directives.

The passwordsGroup property represents a nested FormGroup that encapsulates the password and confirm password fields. It will be convenient to manage their values as a single object for validation.

##### NOTE

In the reactive API, the onSubmit() method doesn’t need arguments because you access the form values using your component’s myFormModel property.

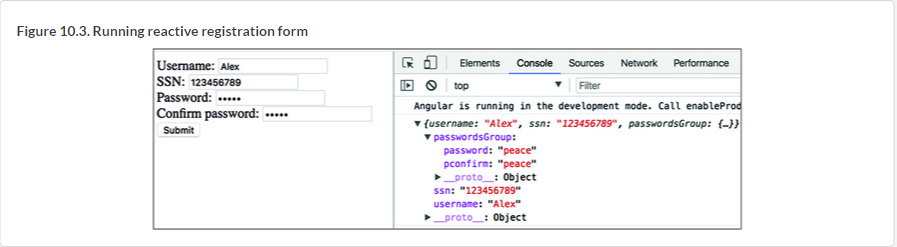
Now that the model is defined, you can write the HTML markup that binds to your model object.



The behavior of this reactive version of the registration form is identical to the template-driven version, but the internal implementation differs. To see this app in action, open the form-samples directory in your IDE, and run the following command:



Fill out the form, and click Submit. The object with the entered values will be printed in the browser’s console, as shown in [figure 10.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/113#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10fig03).

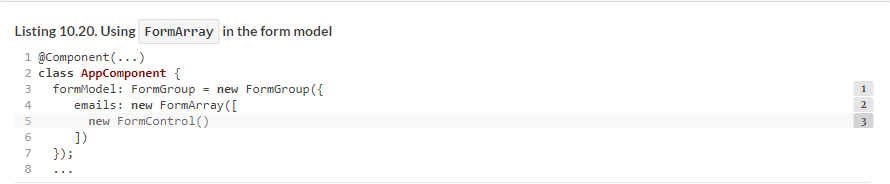


This was a rather simple form with predefined controls, but what if you want to be able to dynamically add form controls during runtime?

#### 10.3.4. Dynamically adding controls to a form

When you know in advance all the controls in a particular form, you can associate each template form element with a corresponding property of the FormGroup instance using the formControlName directive. But if you want to be able to dynamically add/remove controls, you need a different way to link the control names with the model properties. By using FormArray instead of FormGroup, you can specify an array index as a name of the corresponding template element.

Let’s look at an example that allows users to have a form with an arbitrary number of email controls. First, you’ll define the model that will include a FormArray called emails, which will initially have just one form control for entering an email.



In the template, you’ll create a <ul> HTML element and will link it to the emails array of the model using the formArrayName directive. Then, you’ll iterate through this array with \*ngFor, rendering an <li> element for each form control from this array. Your template will also have an Add Email button, and if the user clicks it, you’ll add a new FormControl to the emails array, as shown in the next listing.



In Angular templates, the \*ngFor directive gives you access to a special index variable that stores the current index while iterating through a collection. The let i notation in the \*ngFor loop allows you to automatically bind the value index to the local template variable i available within the loop.

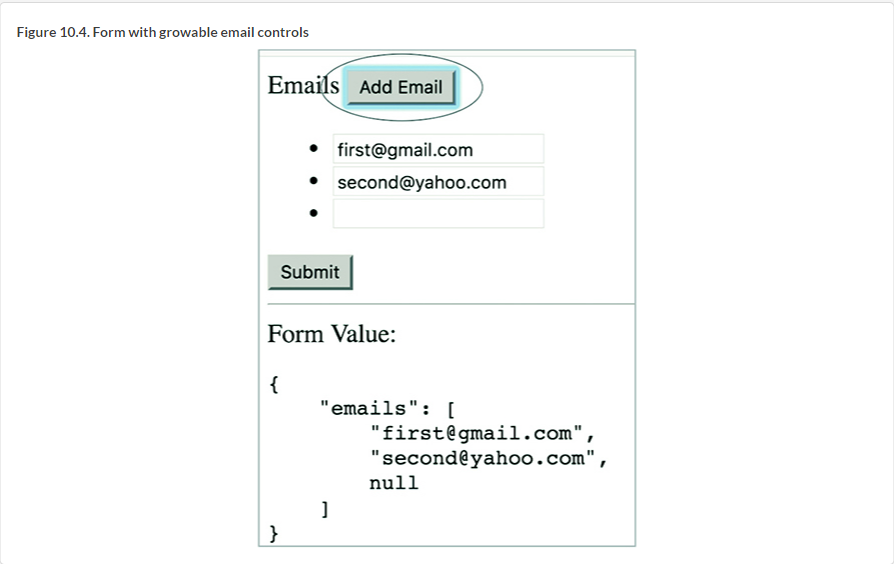
The formControlName directive links the FormControl in FormArray to the currently rendered DOM element; but instead of specifying a name, it uses the current value of the variable i. When the user clicks the Add Email button, your component adds a new FormControl instance to the FormArray:



In the dynamic-form-controls directory, you can find the complete code for the app that dynamically adds email form controls on each click of the Add Email button. To see this example in action, run the following command:

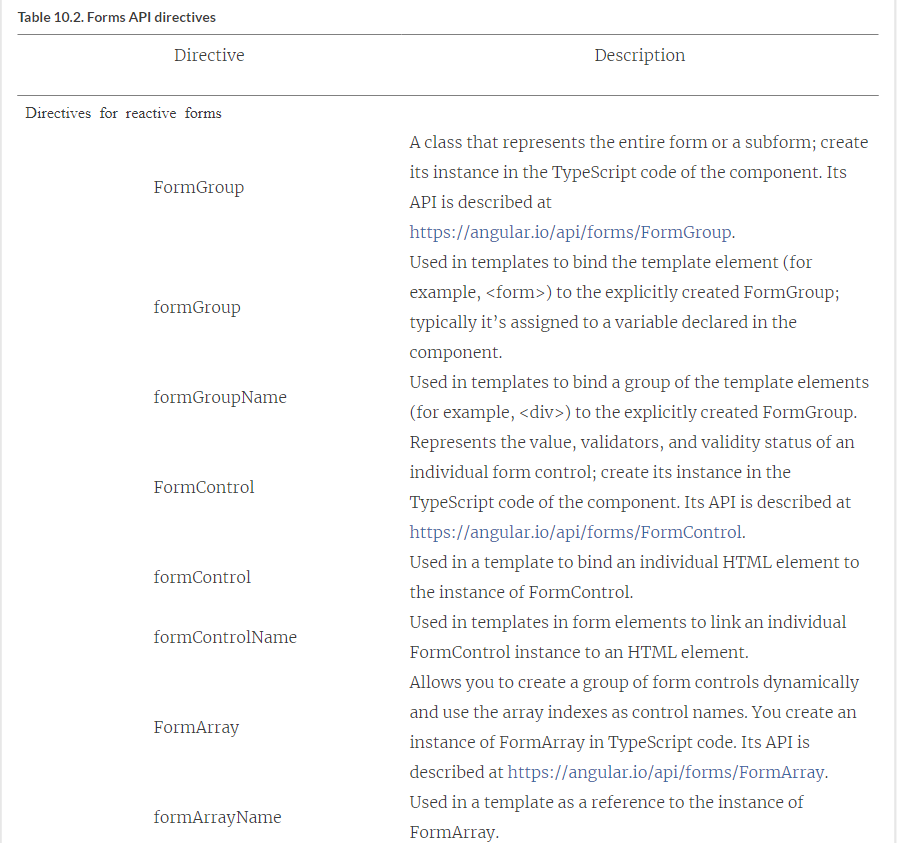


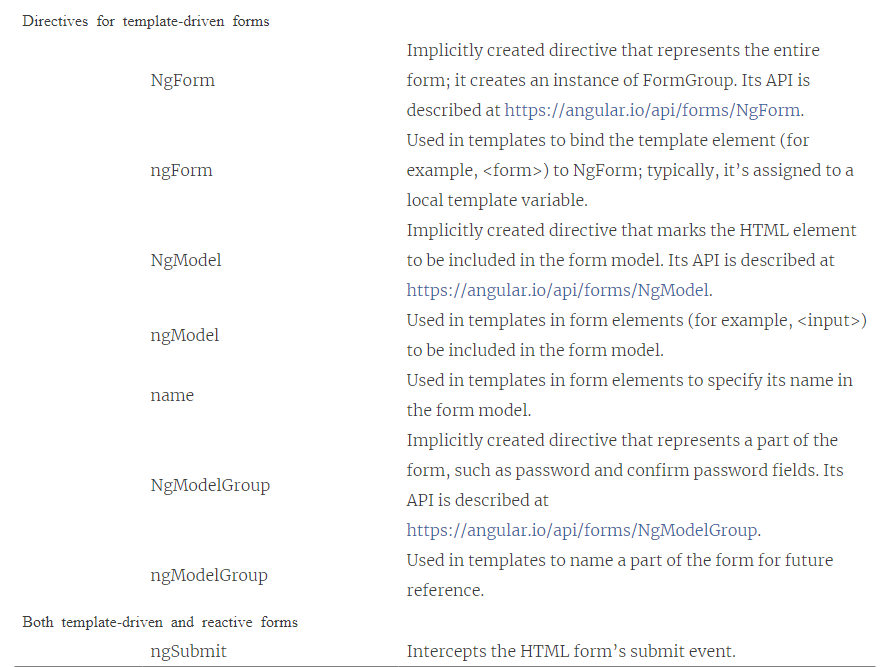
[Figure 10.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/127#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10fig04) shows what this form will look like after the user clicks Add Email. The second email field was added dynamically by adding a new FormControl instance to the FormArray named emails, and each control from this array was rendered on the page.



### 10.4. Forms API directives summary

You’ve used+ many different Forms API directives in both template-driven and reactive forms. [Table 10.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/131#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10table02) lists what they are for.





Note that the name of any directive used in the component template starts with a lowercase letter. The name of the underlying class that implements the directive starts with a capital letter. In template-driven forms, you don’t need to explicitly create instances of these classes, but in reactive forms, you instantiate them in the TypeScript code as needed.

All the code samples in this chapter illustrate use cases of the user entering the data in a form, but often you need to populate a form with existing data.

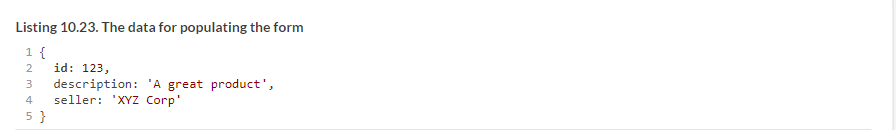
### 10.5. Updating form data

In some scenarios, a form needs to be populated without the user’s interaction. For example, you may need to create a form for editing product data retrieved from the server or another source. Another example is implementing a master-detail relationship—for example, selecting a product in a list should show its details in a form.

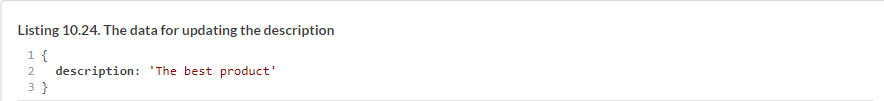
The Angular Forms API offers several functions for updating a form model including reset(), setValue(), and patchValue(). The reset() function reinitializes the form model and resets the flags on the model, like touched, dirty, and others. The setValue() function is used for updating all values in a form model. The patchValue() function is used when you need to update the selected properties of a form model. Let’s create a simple app that will have a form with the model shown in the following listing.



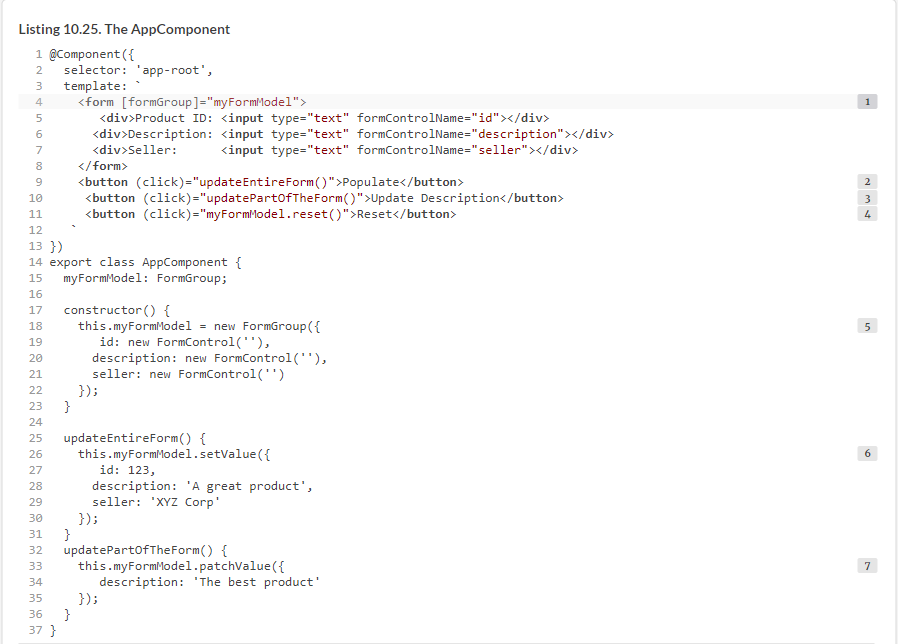
Your app will also have three buttons: Populate, Update Description, and Reset. Accordingly, the Populate button uses setValue() to populate the object that has values for each control defined in the form model.



The Update Description button uses patchValue() for the partial form update (just the description) from the object in the next listing.



The Reset button removes all data from the form and resets all flags on the form model. The code of your app is shown in the following listing.



The code of this app is located in the populate directory. To see it in action, run the following command:



##### NOTE

You can’t use setValue() in a form that uses FormArray. For such forms, you need to use patchValue() and then invoke the setControl() method on the form model to reset FormArray.

If a form has multiple controls, your code may contain lots of new operators creating new instances of form elements. Is there a way to avoid polluting your code with new statements?

### 10.6. Using FormBuilder

The injectable service FormBuilder simplifies the creation of form models. It doesn’t provide any unique features compared to the direct use of the FormControl, FormGroup, and FormArray classes, but its API is terser and saves you from the repetitive instantiation of objects.

Let’s refactor the code in the user registration form from [section 10.3.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/147#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10lev2sec5). The template will remain exactly the same, but the following listing uses FormBuilder to construct the form model.



The FormBuilder.group() method accepts an object with extra configuration parameters as the last argument. You can use it to specify group-level validators there if needed.

As you can see, configuring a form model with FormBuilder is less verbose and is based on the configuration object rather than requiring explicit instantiation of the control’s classes.

To see this app in action, run the command ng serve --app formbuilder -o. Now that you know how to work with form models and templates, you may be wondering how to ensure that the values entered in the form are valid. That’s subject of the next chapter.

### Summary

* Angular offers two APIs for working with forms: template-driven and reactive.
* The template-driven approach is easier and quicker to configure, but it has limited features.
* The reactive approach gives you more control over forms, which can be created or modified during runtime.