**Chapter 11. Validating forms**

*This chapter covers*

* Using built-in form validators
* Creating custom validators
* Handling sync and async validation

The user fills out a form and clicks Submit, expecting that the app will process the data in some way. In web applications, the data is usually sent to the server. Often, the user receives some data back (for example, search results), but sometimes, the data is just saved in the server’s storage (for example, creating a new order). In any case, the data should be valid so the server’s software can do its job properly.

For example, an app can’t log in a user unless they’ve provided a user ID and a password in the login form. Both fields are required—otherwise, the form isn’t valid. You shouldn’t even allow submitting this form until the user has filled out all required fields. A user registration form may be considered invalid if the password doesn’t contain at least 8 characters, including a number, an uppercase letter, and a special character.

In this chapter, we’ll show you how to validate forms in Angular using built-in validators and how to create custom forms. At the end of the chapter, you’ll develop a new version of ngAuction that will include three fields. The entered values will be validated first, and only afterward will they be submitted for finding products that meet entered criteria.

We’ll start exploring built-in validators by using a reactive form and then move to a template-driven one.

### 11.1. Using built-in validators

The Angular Forms API includes the Validators class, with static functions such as required(), minLength(), maxLength(), pattern(), email(), and others. These built-in validators can be used in templates by specifying the directives required, minLength, maxLength, pattern, and email, respectively. The pattern validator enables you to specify a regular expression.

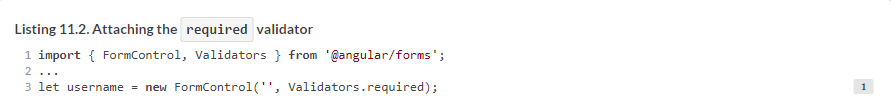
Validators are functions that conform to the interface in the following listing.



If a validator function returns null, that means no errors. Otherwise, it’ll return a ValidationErrors object of type {[key: string]: any}, where the property names (error names) are strings, and values (error descriptions) can be of any type.

A validator function should declare a single argument of type AbstractControl (or its descendants) and return an object literal or null. There, you implement business logic for validating user input. AbstractControl is the superclass for FormControl, FormGroup, and FormArray; hence, validators can be created for all model classes.

With the reactive Forms API, you can either provide validators while creating a form or form control or attach validators dynamically during runtime. The next listing shows an example that attaches the required validator to the form control represented by the variable username.



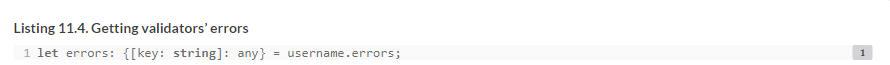
Here, the first parameter of the constructor is the initial value of the control, and the second is the validator function. You can also attach more than one validator to a form control.



To query the form or form control’s validity, use the valid property, which can have one of two values, true or false:

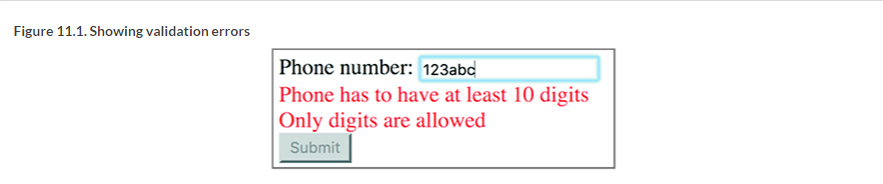


The preceding line checks whether the value entered in the form control passes or fails all the validation rules attached to this control. If any of the rules fails, you’ll get error objects generated by the validator functions, as in the next listing.

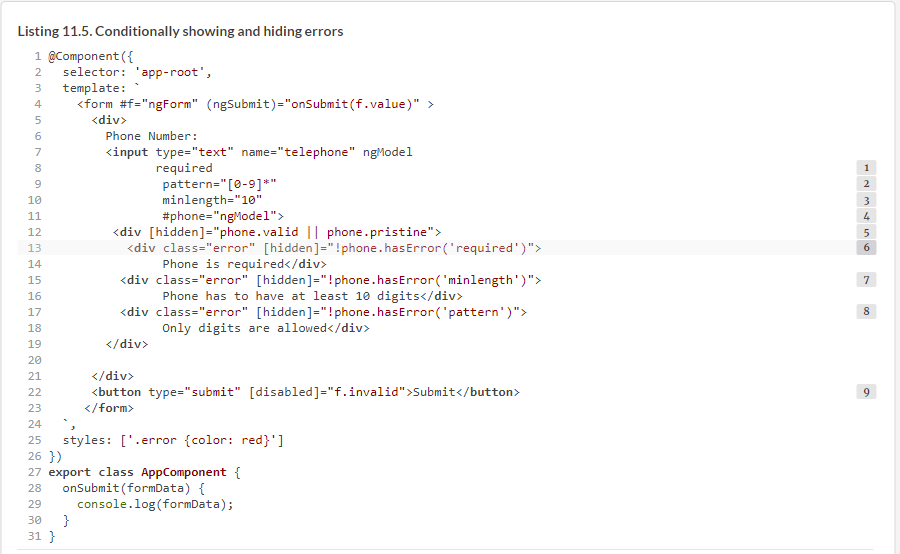


With the method hasError(), you can check whether a form or control has specific errors and conditionally show or hide corresponding error messages.

Now let’s see how to apply built-in validators in a template-driven form. You’ll create an app that illustrates how to show or hide error messages for required, minLength, and pattern validators. The UI of this app may look like [figure 11.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/20#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig01) if the user enters an invalid phone number.



The app component’s template will include the <div> element with error messages located under the <input>, and the <div> will be hidden if the phone number is valid or if its value wasn’t entered (*pristine*), as shown in the following listing. The Submit button will stay disabled until the user enters a value that passes all validators.

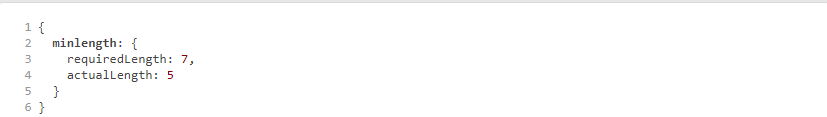


To see this app in action, run npm install in the project folder named form-validation, and then run the following command:



##### THE VALIDATIONERRORS OBJECT

The error returned by a validator is represented by a JavaScript object that has a property whose name briefly describes an error. The property value can be of any type and may provide additional error details. For example, the standard Validators.minLength() validator returns an error object as shown here:



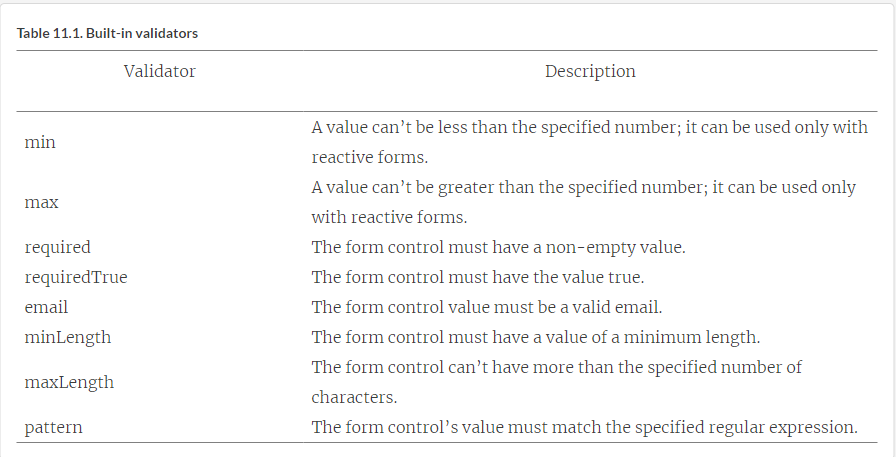
This object has a property named minlength, which means that the minimum length is invalid. The value of this property is also an object with two fields: requiredLength and actualLength. These error details can be used to display a user-friendly error message. Not all validators provide error details. Sometimes, the property just indicates that an error has occurred. In this case, the property is initialized with the value true.

The following snippet shows an example of the built-in Validators.required() error object:



In [section 11.6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/33#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11lev1sec6), you’ll find an example of how to extract the error description from the ValidationErrors object.

[Table 11.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/33#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11table01) provides a brief description of Angular built-in validators offered by the Validators class.



In the code sample that validated the phone number, the validators were checking the value after each character entered by the user. Is it possible to control when validation starts?

### 11.2. Controlling when validation starts

Prior to Angular 5, the validators performed their work each time a value in the form control changed. Now, you can use the updateOn property, which gives you better control over the validation process. When you attach a validator, you can specify when validation should start. The updateOn property can take one of these values:

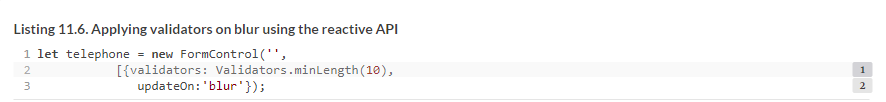
* **change—**This is the default mode, with validators checking a value as soon as it changes. You saw this behavior in the previous section when validating a phone number.
* **blur—**Checks validity of a value when the control loses focus.
* **submit—**Checks validity when the user submits the form.

To try out these options with a template-driven form, add [ngModelOptions]= "{updateOn:'blur'}" to the telephone input field in [listing 11.5](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/38#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11ex05), and the user’s input will be validated only when you move the focus from this control. To start validation when the Submit button is clicked or the Enter key is pressed, use the option [ngModelOptions]="{updateOn:'submit'}".

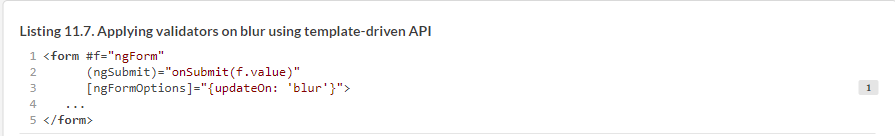
##### NOTE

If you use the sample from [listing 11.5](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/41#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11ex05) with the option updateOn: 'submit', remove the code that conditionally disables the Submit button or use the Enter key to test the validation.

In case of the reactive API, you would set the update mode for a form as follows.



You can also specify the update mode on the form level using the property ngFormOptions, as shown in the following listing.



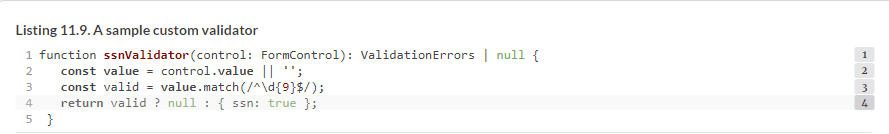
Built-in validators are good for basic validation, but what if you need to apply application-specific logic to decide whether the entered value is valid?

### 11.3. Custom validators in reactive forms

You can create custom validators in Angular. Similar to built-in validators, custom validators should comply with the interface in the following listing.



You need to declare a function that accepts an instance of one of the control types—FormControl, FormGroup, or FormArray—and returns the ValidationErrors object or null. The next listing shows an example of a custom validator that checks whether the control’s value is a valid social security number (SSN).



You can attach custom validators to form controls the same way you attach the built-in ones, as you can see in the following listing.



You’ll see a window with an input field that requires you to enter nine digits to make the error message go away.

Your ssnValidator returns an error object that indicates there’s something wrong with the SSN value: { ssn: true }. You added the error text “SSN is invalid” to the HTML template. The ValidationErrors object can contain a more specific description of the error, for example {ssn: {description: 'SSN is invalid'}}, and you can get the error description using the getError() method. The following listing shows a modified version of ssnValidator and the template.



##### NOTE

In [listing 11.10](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/62#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11ex10), you use Angular’s *safe navigation operator*, which is represented by a question mark and can be used in the component template. The question mark after the invocation of getError() means “Don’t try to access the property description if the object returned by getError() is undefined or null,” meaning when the entered value is valid. If you didn’t use the safe navigation operator, this code would produce the runtime error “cannot read property description of null” for valid SSN values.

If you run this app, the browser shows an empty input field and the message “SSN is invalid,” but the user didn’t have the chance to enter any value. Before showing a validation error message, always check whether the form control is dirty (has been modified). The <span> element should look like the following listing.



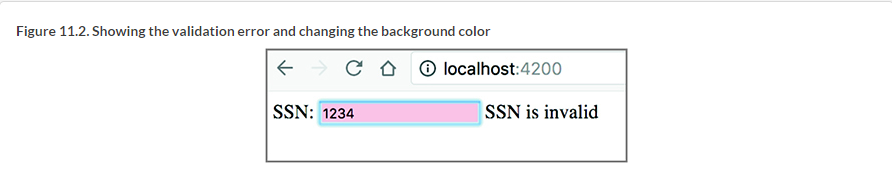
Now let’s add some styling to this form. Angular’s Forms API offers a number of CSS classes that work hand in hand with their respective flags on the form: .ng-valid, .ng-invalid, .ng-pending, .ng-pristine, .ng-dirty, .ng-untouched, and .ng-touched. In the code sample, if the value is invalid and dirty, you want to change the background of the input field to be light pink, as shown in the following listing.



The application that illustrates the use of ssnValidator in a reactive form is located in the reactive-validator directory, and you can run it as follows:



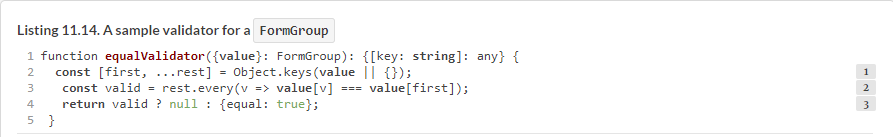
[Figure 11.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/69#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig02) shows how the browser will render this app if the value isn’t valid.



Now that you know how to create a custom validator for a single form control, let’s consider another scenario: validating a group of form controls.

### 11.4. Validating a group of controls

You can validate a group of form controls by attaching validator functions to a FormGroup instead of an individual FormControl. The following listing creates an equalValidator that ensures that the password and password confirmation fields on the sample user registration form have the same value.



The signature of the preceding function conforms to the ValidatorFn interface: the first parameter is of type FormGroup, a subclass of AbstractControl, and the return type is an object literal. Note that you use object destructuring in the function argument to extract the value property from the instance of the FormGroup object.

You also use array destructuring combined with rest parameters in the first line of the function so you can iterate through the properties of FormGroup.value. You get the names of all properties in the value object and save them in two variables, first and rest. first is the name of a property that will be used as the reference value—values of all other properties must be equal to it to make validation pass. rest holds the names of all the other properties.

Finally, the validator function returns either null, if the values in the group are the same, or an error object, otherwise. Let’s apply the ssnValidator and equalValidator in the sample user registration form. The following listing shows the code of the modified AppComponent class.



To display validation errors when the user enters invalid values, you’ll add a <span> element next to each form control in the template. Depending on the return of the hasError() method, the error text will be either shown or hidden, as in the following listing.

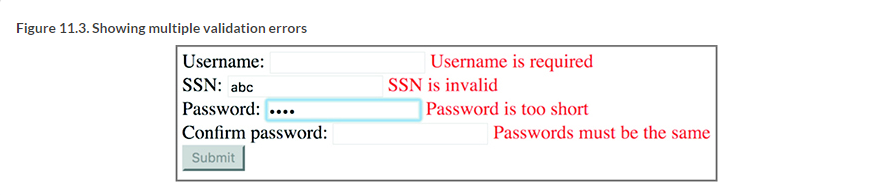


Note how you access the form model’s hasError() method. It takes two parameters: the name of the validation error you want to check and the control name from the form model. In the case of username, it’s a direct child of the top-level FormGroup that represents the form model, so you specify the name of the control. But the password field is a child of the nested FormGroup, so the path to the control is specified as an array of strings: ['passwordsGroup', 'password']. The first element is the name of the nested group, and the second is the name of the password field itself.

You can find the code for this app in the group-validators directory. To see this app in action, run the following command:



[Figure 11.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/82#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig03) illustrates the error messages produced by invalid values in the individual fields Username, SSN, and Password, as well as invalid values in the form group—provided passwords don’t match.



### 11.5. Checking a form control’s status and validity

You already used such control properties as valid, invalid, and errors for checking field status. In this section, we’ll look at a number of other properties that help improve the user experience.

#### 11.5.1. touched and untouched form controls

In addition to checking a control’s validity, you can also use the touched and untouched properties to check whether a form control was visited by the user. If the user puts the focus into a form control using the keyboard or mouse and then moves the focus out, this control becomes touched; while the focus remains in the control, it’s still untouched. This can be useful when displaying error messages—if the value in a form control is invalid, but it was never visited by the user, you can choose not to highlight it with red, because the user didn’t even try to enter a value. The following listing shows an example.



##### NOTE

All the properties discussed in [section 11.5.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/92#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11lev2sec1) are available for the model classes FormControl, FormGroup, and FormArray as well as for the template-driven directives NgModel, NgModelGroup, and NgForm.

Note the CSS class binding example on the last line. It conditionally applies the hasError CSS class to the element if the expression on the right side is true. If you used only c.invalid, the border would be highlighted as soon as the page was rendered; but that can confuse users, especially if the page has a lot of fields. Instead, you add one more condition: the field must be touched. Now the field is highlighted only after a user visits and leaves this field.

#### 11.5.2. pristine and dirty fields

Another useful pair of properties are pristine and dirty. pristine means the user never interacted with the form control. dirty indicates that the initial value of the form control was modified, regardless of where the focus is. These properties can be used to display or hide validation errors.

##### NOTE

All the properties in [section 11.5.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/95#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11lev2sec2) have corresponding CSS classes (ng-touched and ng-untouched, ng-dirty and ng-pristine, ng-valid and ng-invalid) that are automatically added to HTML elements when the respective property is true. These can be useful to style elements in a certain state.

#### 11.5.3. Pending fields

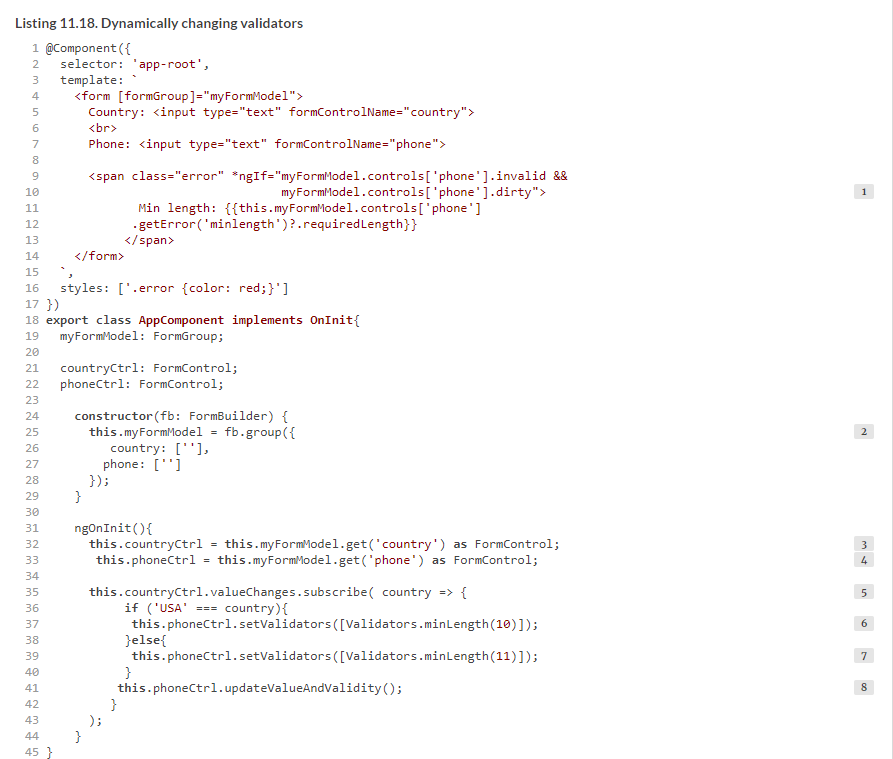
If you have async validators configured for a control, the pending property may come in handy. It indicates whether the validity status is currently unknown. This happens when an async validator is still in progress and you need to wait for the results. This property can be used for displaying a progress indicator.

For reactive forms, the type of the statusChanges property is Observable, and it emits one of three values: VALID, INVALID, and PENDING.

### 11.6. Changing validators dynamically in reactive forms

Using the reactive Forms API, you can change the validators attached to a form or one of its controls during runtime. You may need to implement a scenario where, depending on user input in one control, validation rules for another control should be changed. You can do that using the setValidators() and updateValueAndValidity() functions.

Imagine a form that has two controls: country and phone. If the user enters USA in the country field, you want to allow entering the phone number without the country code, and the phone has to have at least 10 characters. For other countries, the country code is required, and the phone has to have at least 11 characters. In other words, you need to dynamically set the validator for the phone based on the input in the country field. The following listing shows how to implement this: you subscribe to the valueChanges property of the country field and assign the validator to the phone field based on the selected country.



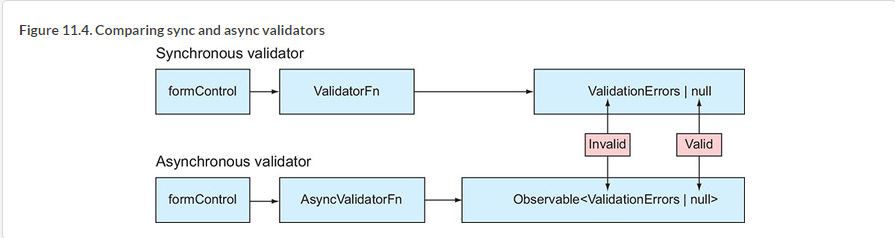
To see this app in action, run the following command:



So far, you’ve been performing validation on the client side, but what if you want to do server-side validation of form values?

### 11.7. Asynchronous validators

Asynchronous validators can be used to check form values by making requests to a remote server. Like synchronous validators, async validators are functions. The main difference is that async validators should return either an Observable or a Promise object. [Figure 11.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/108#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig04) compares the interfaces that synchronous and asynchronous validators should implement. It shows the validators for a formControl, but the same applies to any subclass of the AbstractControl.

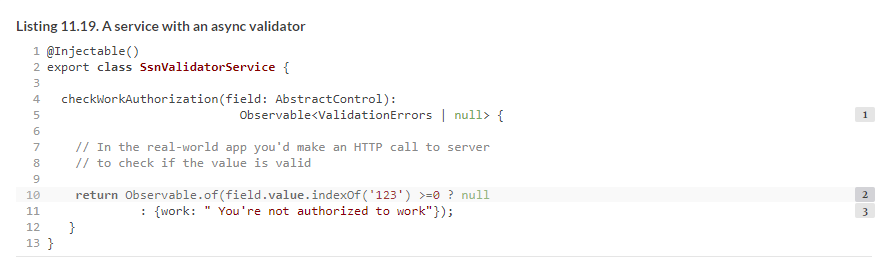


##### TIP

If a form control has both sync and async validators, the latter will be invoked only after the value(s) pass all synchronous validators.

The code that comes with this chapter includes a directory called async-validator that uses both sync and async validators to validate the SSN. For synchronous validation, you’ll reuse the ssnValidator() function that you created in [section 11.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/112#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11lev1sec3). That validator checked that the user entered nine digits in the form control.

Now you also want to invoke a service that will check whether the entered SSN authorizes the user to work in the USA. By your rules, if a person’s SSN has a sequence 123 in it, they can work in the USA. The following listing creates an Angular service that includes such an asynchronous validator.



The following listing creates a component that, in addition to the synchronous ssnValidator(), attaches to the form control the async validator, checkWorkAuthorization().



Async validators are passed as the third argument to constructors of model classes. If you need to have several synchronous or asynchronous validators, specify an array as the second and/or third argument.

In general, the HTML <template> element is used to specify the content that’s not rendered by the browser on page load but can be rendered by JavaScript later on. The Angular <ng-template> directive serves the same purpose. In your component, the content of <ng-template> is “SSN is valid,” and it’s not rendered on page load. The Angular directive \*ngIf will render it if the entered SSN is valid, using the template variable validSSN as a reference.

When assigning the asynchronous validator checkWorkAuthorization(), you want to make sure that this method runs in the context of the service ssnValidatorService. That’s why you used the JavaScript function bind(). To see this application in action, run the following command:



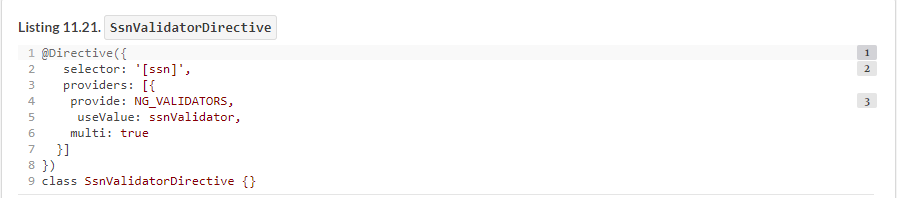
Try entering the SSN with and without the 123 sequence to see different validation messages.

##### NOTE

The source code for this example includes one more async validator, checkWorkAuthorizationV2(), that can’t be attached to the form control because it doesn’t conform to the interface shown in [figure 11.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/121#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig04). We added that validator just to show that you can invoke any function for validating the form values.

### 11.8. Custom validators in template-driven forms

With template-driven forms, you can use only directives to specify validators, so wrapping validator functions into directives is required. The following listing creates a directive that wraps the synchronous SSN validator from [section 11.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/121#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11lev1sec3).



The square brackets around the ssn selector denote that the directive can be used as an attribute. This is convenient, because you can add this attribute to any <input> element or to an Angular component represented as a custom HTML element.

In [listing 11.20](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/129#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11ex20), you register the validator function using the predefined NG\_VALIDATORS Angular token. This token is, in turn, injected by the NgModel directive, and NgModel gets the list of all validators attached to the HTML element. Then, NgModel passes validators to the FormControl instance it implicitly creates. The same mechanism is responsible for running validators; directives are just a different way to configure them. The multi property lets you associate multiple values with the same token. When the token is injected into the NgModel directive, NgModel gets a list of values instead of a single value. This enables you to pass multiple validators.

Here’s how you can use the SsnValidatorDirective:



You can find the complete running application that illustrates directive validators in the template-validator directory. To see this app in action, run the following command:



[Chapter 10](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/133#!/book/angular-development-with-typescript-second-edition/chapter-10/ch10) covered the basics of the Forms API. In this chapter, we’ve explained how to validate form data. Now it’s time to modify ngAuction and add a search form so users can search for products.

##### NOTE

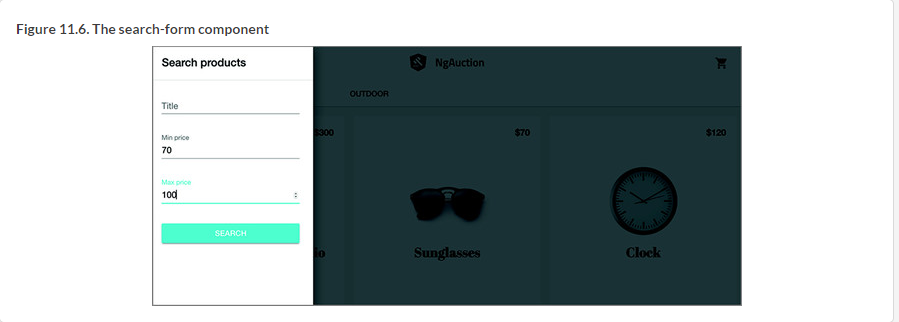
Source code for this chapter can be found at <https://github.com/Farata/angulartypescript> and [www.manning.com/books/angular-development-with-typescript-second-edition](http://www.manning.com/books/angular-development-with-typescript-second-edition).

### 11.9. Adding a search form to ngAuction

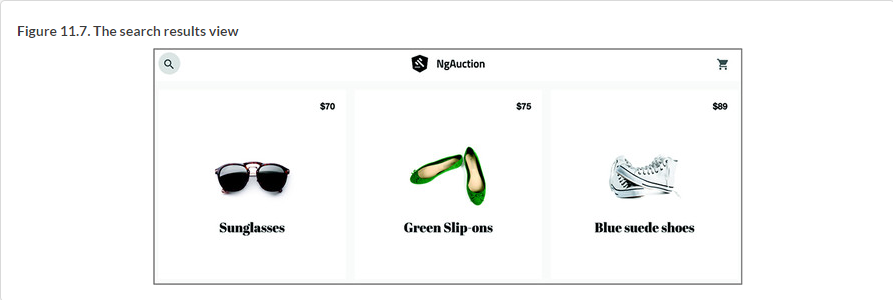
You made quite a few changes to the new version of ngAuction. The main addition is the new search component where you use the Angular Forms API. You’ll also add tabs with product categories, so the top portion of ngAuction will look like [figure 11.5](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/133#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig05).



To return to the landing page from any other app view, the user should click the ngAuction logo. When the user clicks the search icon, the search-form component will slide from the left, where the user can enter search criteria, as shown in [figure 11.6](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/138#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig06).



After the user clicks the Search button, the app will invoke ProductService .search(); the search-form component will slide back off the screen; and the user will see the products that meet the search criteria rendered by the search-results component. Note that there are no tabs with categories in the search result view displayed in [figure 11.7](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/141#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig07). That’s because products from different categories can meet the search criteria—for example, a price between $70 and $100.



In this section, we won’t be providing detailed instructions for implementing all the code changes, because that would take lots of pages to describe. We’ll do a code review of the new search-form component and search-results component. Then we’ll highlight other important changes made throughout the code of ngAuction.

#### 11.9.1. The search-form component

You created the search-form component in the shared directory of the project just in case the search functionality will be required in other parts of this app. The template of the search-form component contains a form with three input fields: Title, Min price, and Max price.

Each of these fields, along with corresponding validation error messages (<mat-error>), is wrapped into an Angular Material <mat-form-field>, and the value in the placeholder attribute (or the field label, if present) becomes a floating label, as you can see in the following listing.

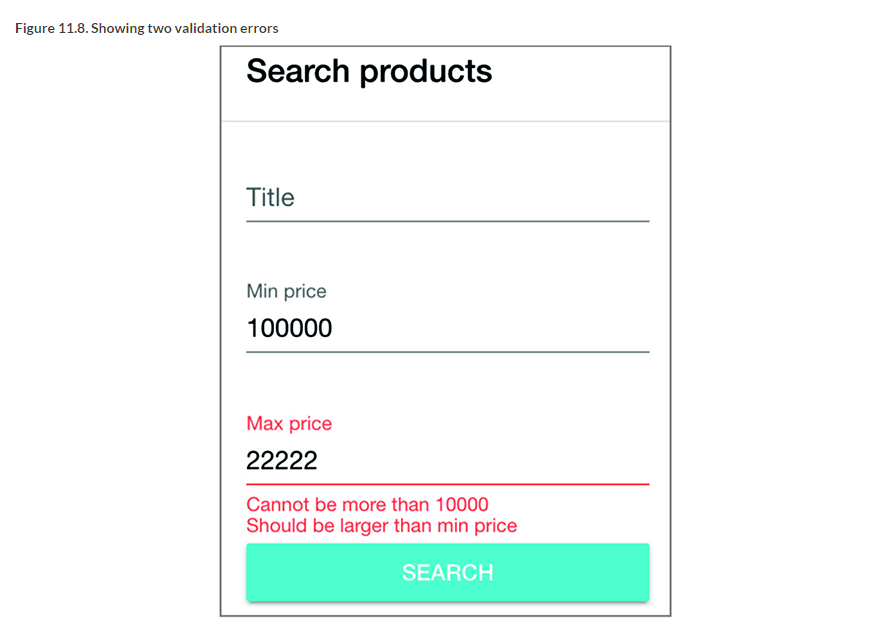


In the TypeScript code, you’ll attach a validator to require at least two characters in the Title field. The <mat-error> will display the error message if the entered value won’t pass the validator. The Min price field has a validator that doesn’t allow negative numbers.

But the Max price field has three validators and three corresponding error messages: the first one is shown if the value is negative, the second is shown if the entered price is greater than 10,000, and the third one is shown if the entered Max value is less than the Min value.

You’ll create a custom validator named minLessThanMaxValidator in the TypeScript code of the search-form component. Because this validator needs the values from two fields, you’ll attach it to the entire form and not to the individual form control. Accordingly, for this validator in <mat-error>, you invoke hasError() not in the form control but in the form.

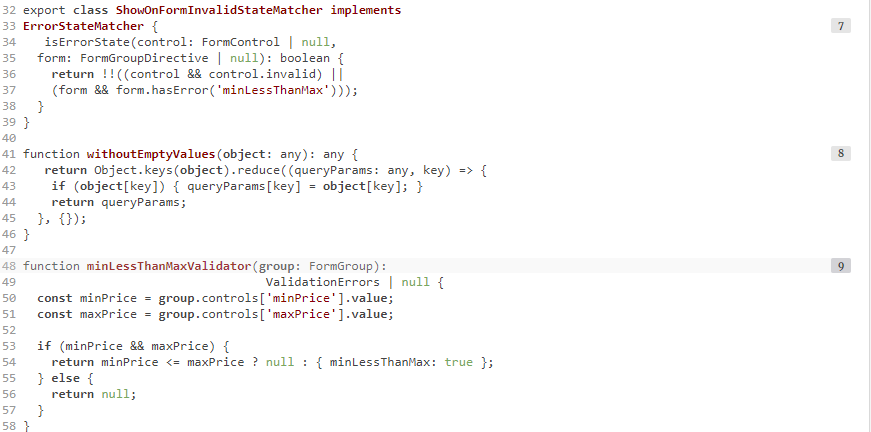
By default, validation errors are shown when the value is invalid and the user interacts with the control. The Max price field is special because one of its validators should kick in when the value in the Min price field is also entered. To specify *when* this validator should check the values, you’ll implement the ErrorStateMatcher interface in the TypeScript code of the search-form component. If the entered value doesn’t pass one or more validators, the respective error messages will be displayed, as shown in [figure 11.8](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/150#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig08).



The search-form.component.ts file includes the decorated class SearchFormComponent and the custom validator minLessThanMaxValidator. The form model is created using FormBuilder.

When the user clicks the Search button, the router navigates to the search-results component, passing the search criteria as query parameters (see [section 3.5.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/155#!/book/angular-development-with-typescript-second-edition/chapter-3/ch03lev2sec4) in [chapter 3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/155#!/book/angular-development-with-typescript-second-edition/chapter-3/ch03)). The search-results component implements the search functionality. You also emit a custom search event, which notifies the AppComponent that the search panel can be closed, as shown in the following listing.





The matInput directive has the errorStateMatcher property, which takes an instance of the ErrorStateMatcher object. This object must implement the isErrorState() method that takes the form control and the form and has the app logic to decide whether the error message has to be shown. In this case, this function returns true (show the error) if the control’s value is invalid or if the minLessThanMax validator returned an error.

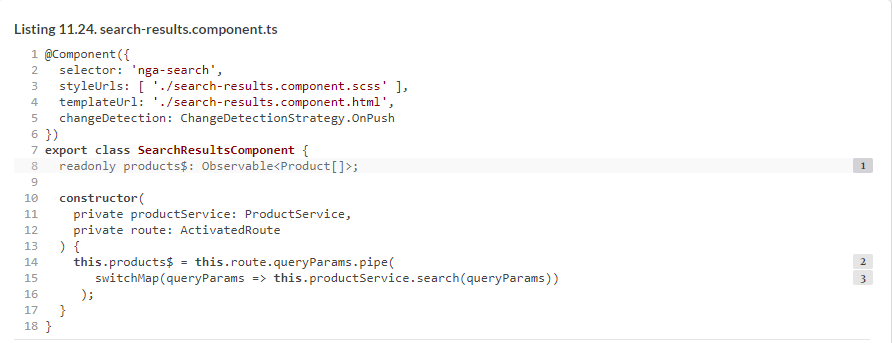
While searching for products, keep an eye on the URL, which will contain the search parameters. For example, if you enter red in the title field and click Search, you’ll invoke Router.navigate(), and the URL will change to localhost:4200/search-results?title=red. The function withoutEmptyValues() ensures that if some search parameters weren’t used (for example, min and max price), they won’t be used in the query parameters.

##### NOTE

Jn nhYntiocu mklt [chapter 10](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-10/ch10), pkr home component in cdulde <mat-grid-list> ltx eidnrrnge rvq rjfz lk usptrdco. Jn jzbr orsvnie vl npYociutn, wk ectdreaxt orb tgjq jfra xnrj z epsareat udtprco-ytuj ontcopnem prcr vwn aj rduees qd rwe components: satrgoecei cun schear-lsrsteu (yred lgnebo er xrq home module).

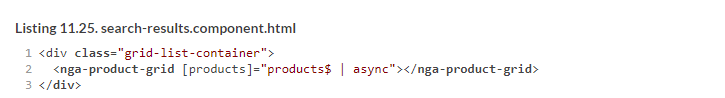
#### 11.9.2. The search-results component

The search-results component receives the query parameters via the observable property queryParams of the ActivatedRoute object. Using the switchMap operator, you pass the value emitted by the queryParams observable to another observable, the search() method on ProductService, as shown in the following listing.



If you need a refresher on the switchMap operator, see [section D.8](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/165#!/book/angular-development-with-typescript-second-edition/appendix-d/app04lev1sec8) of [appendix D](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/165#!/book/angular-development-with-typescript-second-edition/appendix-d/app04). You can read about pipeable operators in [section D.4.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/165#!/book/angular-development-with-typescript-second-edition/appendix-d/app04lev2sec1).

The template of the search-results component incorporates the product-grid component and uses the async pipe to unwrap the observable products$, as shown in the next listing.



The product-grid component receives the products via its input parameter as follows:



Then it renders the grid with products as described in [section 9.3.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/170#!/book/angular-development-with-typescript-second-edition/chapter-9/ch09lev2sec8) in [chapter 9](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/170#!/book/angular-development-with-typescript-second-edition/chapter-9/ch09).

#### 11.9.3. Other code refactoring

##### We won’t be providing complete code listings of other ngAuction components that underwent refactoring but rather will highlight the changes. You’re encouraged to go through the code of ngAuction that comes with this chapter. If you have specific questions about the code, post them on the book forum at <https://forums.manning.com/forums/angular-development-with-typescript-second-edition>.Showing and hiding the search-form component

In the ngAuction from [chapters 2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/170#!/book/angular-development-with-typescript-second-edition/chapter-2/ch02), [3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/170#!/book/angular-development-with-typescript-second-edition/chapter-3/ch03), and [4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/170#!/book/angular-development-with-typescript-second-edition/chapter-4/ch04), the search component was always present on the UI, occupying 25% of the screen width. Why does it take so much space even when the user isn’t searching for products? In this version of the app, the search-form component is represented by a small search icon, which is a part of the app toolbar, as shown in [figure 11.9](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-11/170#!/book/angular-development-with-typescript-second-edition/chapter-11/ch11fig09).



Angular Material offers components for side navigation with which you can add collapsible side content to a full-screen app. You use the <mat-sidenav-container> component, which acts as a structural container for both the side-navigation panel (the search-form component) and the toolbar of ngAuction. <mat-sidenav> represents the added side content—the search-form component, in your case—as shown in the following listing.

