Collecting User Data with Forms

Web applications use forms when it comes to collecting data from the user. Use cases vary from allowing users to log in, filling in payment information, booking a flight, or even performing a search. Form data can later be persisted on local storage or be sent to a server using a backend API. A form usually has the following characteristics that enhance the user experience of a web application:

- Can define different kinds of input fields
- Can set up different kinds of validations and display validation errors to the user
- Can support different strategies for handling data if the form is in an error state

The Angular framework provides two approaches to handling forms: **template-driven** and **reactive**. Neither approach is considered better than the other; you have to go with the one that suits your scenario the best. The main difference between the two approaches is how they manage data:

- Template-driven forms are easy to set up and add to an Angular application, but they do not scale well. They operate solely on the component template to create elements and configure validation rules; thus, they are not easy to test. They also depend on the change detection mechanism of the framework.
- Reactive forms are more robust when it comes to scaling and testing and when they are not interacting with the change detection cycle. They operate in the component class to manage input controls and set up validation rules. They also manipulate data using an intermediate form model, maintaining their immutable nature. This technique is for you if you use reactive programming techniques extensively or if your Angular application comprises many forms.

In this chapter, we will focus mainly on reactive forms due to their wide popularity in the Angular community. More specifically, we will cover the following topics:

- Introducing forms to web apps
- Data binding with template-driven forms
- Using reactive patterns in Angular forms
- Validating controls in a reactive way
- Modifying forms dynamically
- Manipulating form data

- Watching state changes and being reactive

Technical requirements

The chapter contains various code samples to walk you through the concept of creating and managing forms in Angular. You can find the related source code in the `ch10` folder of the following GitHub repository:

https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition

Introducing forms to web apps

A form in a web application consists of a `<form>` HTML element that contains some `<input>` elements for entering data and a `<button>` element for handling that data. The form can retrieve data and either save it locally or send it to a server for further manipulation. The following is a simple form that is used for logging a user in to a web application:

```
<form>
    <div>
      <input type="text" name="username" placeholder="Username">
    </div>
    <div>
      <input type="password" name="password" placeholder="Password">
    </div>
    <button type="submit">Login</button>
  </form>
```

The preceding form has two `<input>` elements: one for entering the username and another for entering the password. The type of the **password** field is set to `password` so that the content of the input control is not visible while typing.

The type of the `<button>` element is set to `submit` so that the form can collect data by clicking on the button or pressing *Enter* on any input control. We could have added another button with a **reset** type to clear form data. Notice that an HTML element must reside inside the `<form>` element to be part of it. The following screenshot shows what the form looks like when rendered on a page:

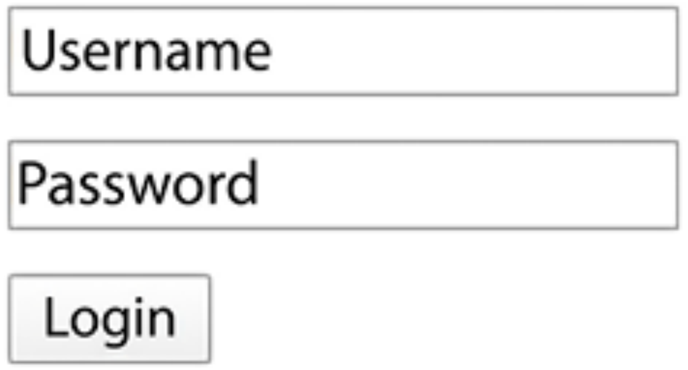


Figure 10.1: Login form

Web applications can significantly enhance the user experience by using forms that provide features such as autocomplete in input controls or prompting to save sensitive data. Now that we have understood what a web form looks like, let’s learn how all that fits into the Angular framework.

Data binding with template-driven forms

Template-driven forms are one of two different ways of integrating forms with Angular. It is an approach that is not widely embraced by the Angular community for the reasons described previously. Nevertheless, it can be powerful in cases where we want to create small and simple forms for our Angular application. Template-driven forms can stand out when used with the `ngModel` directive to provide two-way data binding in our components.

We learned about data binding in *Chapter 4*, *Enabling User Experience with Components*, and how we can use different types to read data from an HTML element or component and write data to it. In this case, binding is either one way or another, which is also called **one-way binding**. We can combine both ways and create a **two-way binding** that can read and write data simultaneously. Template-driven forms provide the `ngModel` directive, which we can use in our components to get this behavior. Before we can start using Angular forms, we need to configure our Angular application by importing `FormsModule`, an appropriate built-in Angular module for working with template-driven forms:

1. Run the following command to create a new Angular application:

```
ng new my-app --routing --style=css
```

The preceding command will create an Angular application that enables routing and uses CSS for component styling.

2. Copy the contents of the `src\app` folder from the source code of *Chapter 9*, *Navigate through Application with Routing*, in the `src\app` folder of the current Angular project. Replace any files if needed.
3. Copy the `styles.css` file from the source code of *Chapter 9*, *Navigate through Application with Routing*, in the `src` folder of the current Angular project and replace it.
4. Open the `products.module.ts` file and add the following `import` statement:

```
import { FormsModule } from '@angular/forms';
```

   We add template-driven forms to an Angular application by importing `FormsModule` from the `@angular/forms` npm package.

5. Add `FormsModule` in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
    declarations: [
      ProductListComponent,
      ProductDetailComponent,
      SortPipe,
      ProductViewComponent,
      ProductCreateComponent
    ],
    imports: [
      CommonModule,
      ProductsRoutingModule,
      FormsModule
    ],
    exports: [ProductListComponent]
  })
```

We have already established the infrastructure in our Angular application to start using Angular forms. We will now create our first form for changing the product price in the product details component:

1. Open the `product-detail.component.html` file and modify the `<input>` element as follows:

```
<input placeholder="New price" name="price" [(ngModel)]="product.price" />
```

In the preceding snippet, we bind the `price` property of the `product` template variable to the `ngModel` directive of the `<input>` element.

The syntax of the `ngModel` directive is known as *a banana in a box*, which is a memory rule for you to be able to remember how to type it. We create it in two steps. First, we create the banana by surrounding `ngModel` in parentheses `()`. Then, we put the banana in a box by surrounding it with square brackets `[()]`. Remember, it's called banana in a box, not box in a banana.

The `name` attribute is required in the `<input>` element so that Angular can create a unique form control internally to distinguish it.

2. Modify the `<button>` element as follows:

```
<button type="submit">Change</button>
```

In the preceding snippet, we removed the `click` event from the `<button>` element because our form will be responsible for updating the price. We also added the `submit` type to indicate that the form submission can happen by clicking the button.

3. Surround the `<input>` and `<button>` elements with a `<form>` element:

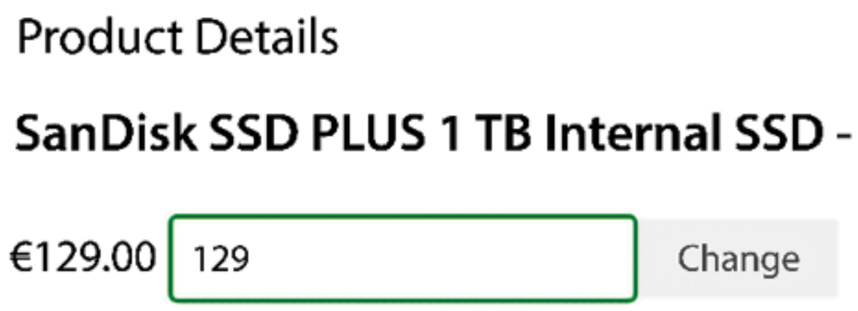
```
<form (ngSubmit)="changePrice(product, product.price)">
    <input placeholder="New price" name="price" [(ngModel)]="product.price" />
    <button type="submit">Change</button>
  </form>
```

In the preceding snippet, we bind the `changePrice` method to the `ngSubmit` event of the form. The binding will trigger the execution of the `changePrice` method if we press *Enter* inside the input box or click the button. The `ngSubmit` event is part of the Angular `FormsModule` and hooks on the native submit event of an HTML form.

4. Open the `product-detail.component.css` file and add the following CSS style to target the `form` element:

```
form {
    display: inline;
  }
```

5. Run the application using the `ng serve` command and select a product from the list.
6. You will notice that the current product price is already displayed inside the input box. Try to change the price, and you will notice that the current price of the product also changes:


Product Details

SanDisk SSD PLUS 1 TB Internal SSD - SATA III 6 Gb/s

€129.00 129

Figure 10.2: Two-way binding

The behavior of our application depicted in the preceding image is the magic behind two-way binding and `ngModel`. While we type inside the input box, the `ngModel` directive updates the value of the product price. The new price is directly reflected in the template because we use Angular interpolation syntax to display its value.

The syntax of a banana in a box that we use for the `ngModel` directive is not random. Under the hood, `ngModel` is a directive that contains an `@Input` binding named `ngModel` and an `@Output` binding named `ngModelChange`. It implements a particular interface called `ControlValueAccessor` that is used to create custom controls for forms. By convention, when a directive or a component contains both bindings that start with the same name, but the output binding ends in `Change`, the property can be used as a two-way binding.

In our case, updating the current product price while entering a new one is a bad user experience. The user should be able to view the current price of the product at all times. We will modify the product details component so that the price is displayed correctly:

1. Open the `product-detail.component.ts` file and create a `price` property inside the `ProductDetailComponent` class:

```
price: number | undefined;
```

2. Open the `product-detail.component.html` file and replace the bindings in the `<input>` and `<form>` elements to use the new component property:

```
<form (ngSubmit)="changePrice(product, price!)">
    <input placeholder="New price" name="price" [(ngModel)]="price" />
    <button type="submit">Change</button>
  </form>
```

In the preceding snippet, we use the non-null assertion operator in the form binding because the `price` property has been declared as `number | undefined`.

If we run the application and try to enter a new price inside the **New price** input box, we will notice that the current price displayed does not change. The functionality of changing the price also works correctly as before.

We have seen how template-driven forms can come in handy when creating small and simple forms. In the next section, we dive deeper into the alternative approach offered by the Angular framework: reactive forms.

Using reactive patterns in Angular forms

Reactive forms, as the name implies, provide access to web forms in a reactive manner. They are built with reactivity in mind, where input controls and their values can be manipulated using observable streams. They also maintain an immutable state of form data, making them easier to test because we can be sure that the state of the form can be modified explicitly and consistently.

Reactive forms have a programmatic approach to creating form elements and setting up validation rules. We set everything up in the component class and merely point out our created artifacts in the template.

The Angular key classes involved in this approach are the following:

- `FormControl`: Represents an individual form control, such as an `<input>` element.
- `FormGroup`: Represents a collection of form controls. The `<form>` element is the topmost `FormGroup` in the hierarchy of a reactive form.
- `FormArray`: Represents a collection of form controls, just like `FormGroup`, but can be modified at

runtime. For example, we can add or remove `FormControl` objects dynamically as needed.

All these classes are available from the `@angular/forms` npm package.
The `FormControl` and `FormGroup` classes inherit from `AbstractControl`, which contains a lot of interesting properties. We can use these properties to render the UI differently based on what status a particular control or group has. We might want to differentiate UI-wise between a form we have never interacted with and one we have. It could also be interesting to know whether we have interacted with a particular control. As you can imagine, there are many scenarios where it is interesting to know a specific status. We will explore all these properties using the `FormControl` and `FormGroup` classes.

In the next section, we will explore how to work with reactive forms in Angular using our component for creating new products.

Interacting with reactive forms

The Angular application we have built contains a component to add new products for our e-shop. When we built the component in the previous chapter, you may have noticed that the name and price input controls were not cleared upon creating a product. Implementing such functionality would be complex because we would need to access the native `<input>` elements inside the component class. However, Angular forms provide a helpful and convenient API that we can use to accomplish this task. We will learn how to use reactive forms by integrating them into the product create component:

1. Open the `products.module.ts` file and import `ReactiveFormsModule` from the `@angular/forms` npm package:

   ```
   import { FormsModule, ReactiveFormsModule  } from '@angular/forms';
   ```

   The Angular forms library provides `ReactiveFormsModule`, which we can use to start creating reactive forms in our Angular application.

2. Add `ReactiveFormsModule` in the `imports` array of the `@NgModule` decorator:

   ```
   @NgModule({
       declarations: [
         ProductListComponent,
         ProductDetailComponent,
         SortPipe,
         ProductViewComponent,
         ProductCreateComponent
   ```

```
  ],
  imports: [
    CommonModule,
    ProductsRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
  exports: [ProductListComponent]
})
```

`ProductsModule` imports both `FormsModule` and `ReactiveFormsModule` in the previous example. There is no harm in doing this. You can use them simultaneously in an Angular application, and everything will work fine.

3. Open the `product-create.component.ts` file and add the following `import` statement:

```
import { FormControl, FormGroup } from '@angular/forms';
```

4. Define the following `productForm` property in the `ProductCreateComponent` class:

```
productForm = new FormGroup({
    name: new FormControl('', { nonNullable: true }),
    price: new FormControl<number | undefined>(undefined, { nonNullable: true })
  });
```

5. The constructor of the `FormGroup` class accepts an object that contains key-value pairs of `FormControl` instances. The key denotes a unique name for the form control that `FormGroup` can use to keep track of, while the value is an instance of `FormControl`.
6. The constructor of the `FormControl` class accepts the default value of the input control as the first parameter. For the `name` form control, we pass an empty string so that we do not set any value initially. For the `price` form control that should accept numbers as values, we set it initially to `undefined`.
7. The second parameter passed in the `FormControl` instance is an object that sets the `nonNullable` property to indicate that the form control does not accept null values.
8. After we have created the form group and its controls, we need to associate them with the respective HTML elements in the template. Open the `product-create.component.html` file and surround the component template with the following `<form>` element:

```
<form [formGroup]="productForm">
```

```
  <div>
    <label for="name">Name</label>
    <input id="name" #name />
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" #price />
  </div>
  <div>
    <button (click)="createProduct(name.value, price.valueAsNumber)">Create</button>
  </div>
</form>
```

In the preceding template, we use the `formGroup` directive, exported from `ReactiveFormsModule`, to connect a `FormGroup` instance to a `<form>` element.

9. `ReactiveFormsModule` also exports the `formControlName` directive, which we use to connect a `FormControl` instance to an `<input>` element. Modify the `<input>` elements of the form as follows:

```
<div>
    <label for="name">Name</label>
    <input id="name" formControlName="name" />
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" />
  </div>
```

In the preceding snippet, we set the value of the `formControlName` directive to the name of the `FormControl` instance.

Currently, we access the `name` and `price` template variables in the binding of the button `click` event. In reactive forms, this is not the case since the form model is the source of truth. So, we need to get input control values from the `FormGroup` or `FormControl` classes. The `FormGroup` class exposes the `controls` property, which we can use to get a specific `FormControl` instance:

1. Open the `product-create.component.ts` file and create the following getter properties:

```
get name() { return this.productForm.controls.name }
  get price() { return this.productForm.controls.price }
```

2. Modify the `createProduct` method so that it uses the newly created properties:

```
createProduct() {
    this.productsService.addProduct(this.name.value, Number(this.price.value)).subscribe(p
      this.productForm.reset();
      this.added.emit(product);
    });
  }
```

The `FormControl` class contains various properties, such as the value of the associated input control. In the preceding method, we also use the `reset` method of the `productForm` property to reset the form in its initial values.

The `FormGroup` class also contains a `value` property, which we can use to access form control values as a single object. We usually use this property when posting whole entities in a backend

API.

3. Open the `product-create.component.html` file and modify its content so that the `createProduct` method is called on form submission:

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
    <div>
      <label for="name">Name</label>
      <input id="name" formControlName="name" />
    </div>
    <div>
      <label for="price">Price</label>
      <input id="price" formControlName="price" />
    </div>
    <div>
      <button type="submit">Create</button>
    </div>
  </form>
```

If we run the application, we will see that the functionality of adding a new product still works as expected. The **Name** and **Price** fields are also cleared upon creating a new product.

Click the **Create** button without entering any values in the input fields and observe what happens in the **Network** tab inside the developer tools of your browser. The application will try to create a product with an empty name and price set to 0. It is a situation that we should avoid in a real-world scenario. We should be aware of the status of a form control and take action accordingly.

In the next section, we'll investigate different properties that we can check to get the status of a form control and provide feedback to the user according to that status.

Providing form status feedback

The Angular framework sets the following CSS classes automatically in a form control according to the current status of the control:

- `ng-untouched`: Indicates that we have not interacted with the control yet
- `ng-touched`: Indicates that we have interacted with the control
- `ng-dirty`: Indicates that we have set a value to the control
- `ng-pristine`: Indicates that the control does not have a value yet

- `ng-valid`: Indicates that the value of the control is valid
- `ng-invalid`: Indicates that the value of the control is not valid

Each class name has a similar property in the form model. The property name is the same as the class name without the `ng-` prefix. We could try to leverage both and provide a unique experience with our forms.

Suppose we would like to display a highlighted border in an input control when interacting with that control for the first time. We should define a global CSS style in the `styles.css` file, such as the following:

```
input.ng-touched {
    border: 3px solid lightblue;
  }
```

We can also combine some of the CSS classes according to the needs of our application. Suppose we would like to display a green border when an input control has a value and a red one when it does not have any at all. The red border should be visible only if we initially entered a value in the input control and deleted it immediately afterward. We should create the following CSS rules in the `styles.css` file:

```
input.ng-dirty.ng-valid {
    border: 2px solid green;
  }
  input.ng-dirty.ng-invalid {
    border: 2px solid red;
  }
```

We must add a validation rule to our input elements for the preceding classes to work. We can use many built-in Angular validators, as we will learn later in this chapter. In this case, we will use the `required` validator, which indicates that an input control must have a value to be valid. To apply it, add the `required` attribute to both `<input>` elements in the `product-create.component.html` file:

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
    <div>
      <label for="name">Name</label>
      <input id="name" formControlName="name" required />
```

```
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" required />
  </div>
  <div>
    <button type="submit">Create</button>
  </div>
</form>
```

Later, in the *Validating controls in a reactive way* section, we will learn how to apply a validator to a `FormControl` instance directly.

Run the application using the `ng serve` command and follow these steps to check the applied CSS rules:

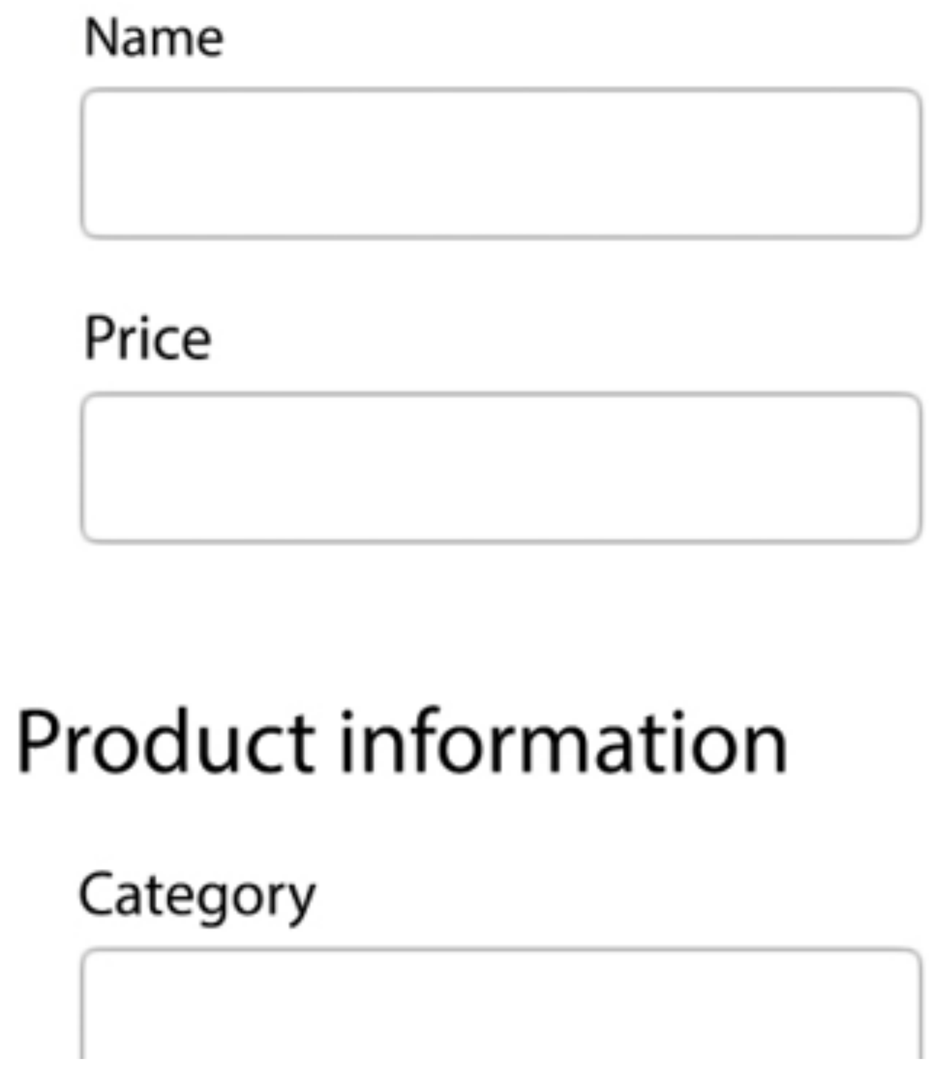
1. Click on the **Name** field and then on the **Price** field. The former should now display a light blue border.
2. Enter some text into the **Name** field and click outside the input control. Notice that it has a green border.

3. Remove the text from the **Name** field and click outside the input control. The border should now turn red.
4. Repeat all previous steps for the **Price** field.

We can now understand what happens when the status of an input control changes and notify users visually about that change. In the next section, we’ll learn that the status of a form can be spawned across many form controls and groups at different levels.

Creating nesting form hierarchies

We have already seen how to create a form with two input controls. The product create form is a simple form that needs one `FormGroup` and two `FormControls`. There are use cases in enterprise applications that require more advanced forms that involve creating nested hierarchies of form groups. Consider the following form, which is used to add product information along with basic details:


Name

Price

Product information

Category

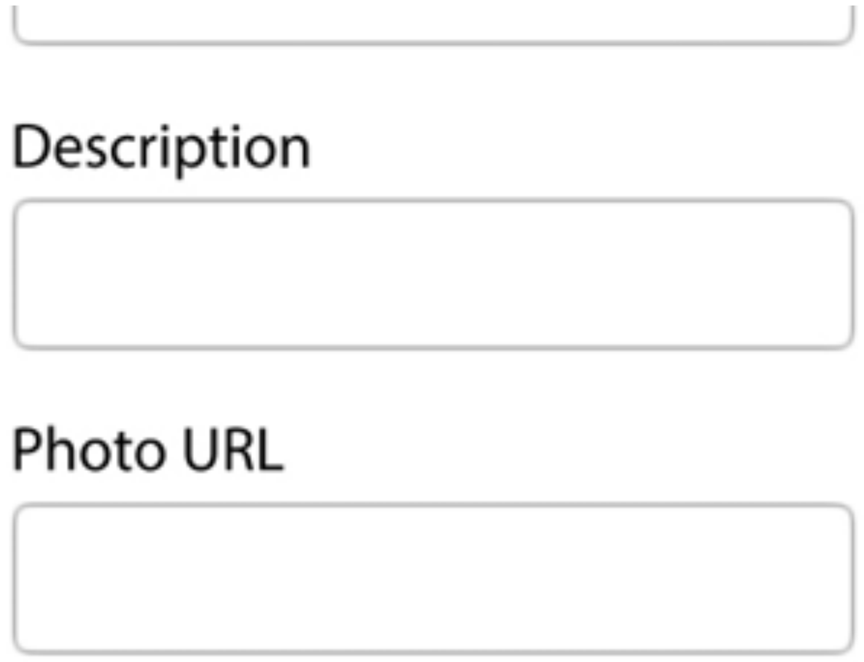


Figure 10.3: New product form

The preceding form may look like a single one, but if we take a better look at the component class, we will see that it consists of two `FormGroup` instances, one nested inside the other:

```
productForm = new FormGroup({
    name: new FormControl('', {
```

```
    nonNullable: true
  }),
  price: new FormControl<number | undefined>(undefined, {
    nonNullable: true
  }),
  info: new FormGroup({
    category: new FormControl(''),
    description: new FormControl(''),
    image: new FormControl('')
  })
});
```

The `productForm` property is the parent form group, while `info` is its child. A parent form group can have as many children form groups as it wants. If we take a look at the component template, we will see that the child form group is defined differently from the parent one:

```
<form formGroupName="info">
    <h2>Product information</h2>
    <div>
      <label for="category">Category</label>
      <input id="category" formControlName="category" />
    </div>
    <div>
      <label for="descr">Description</label>
      <input id="descr" formControlName="description" />
    </div>
    <div>
      <label for="photo">Photo URL</label>
      <input id="photo" formControlName="image" />
    </div>
  </form>
```

In the preceding HTML template, we use the `formGroupName` directive to bind the inner form element to the `info` property.

You may have expected to bind it directly to the `productForm.info` property, but this will not work. The Angular framework is pretty smart because it understands that `info` is a child form group of `productForm`. It can deduce this information because the form element related to `info` is inside the form element that binds to the `productForm` property.

The status of a child form is shared with its parent in a nested form hierarchy. In our case, when the `info` form becomes invalid, its parent form, `productForm`, will also be invalid. The change of status is not the only thing that bubbles up to the parent form. The value of the child form also propagates up the hierarchy, thereby maintaining a consistent form model:

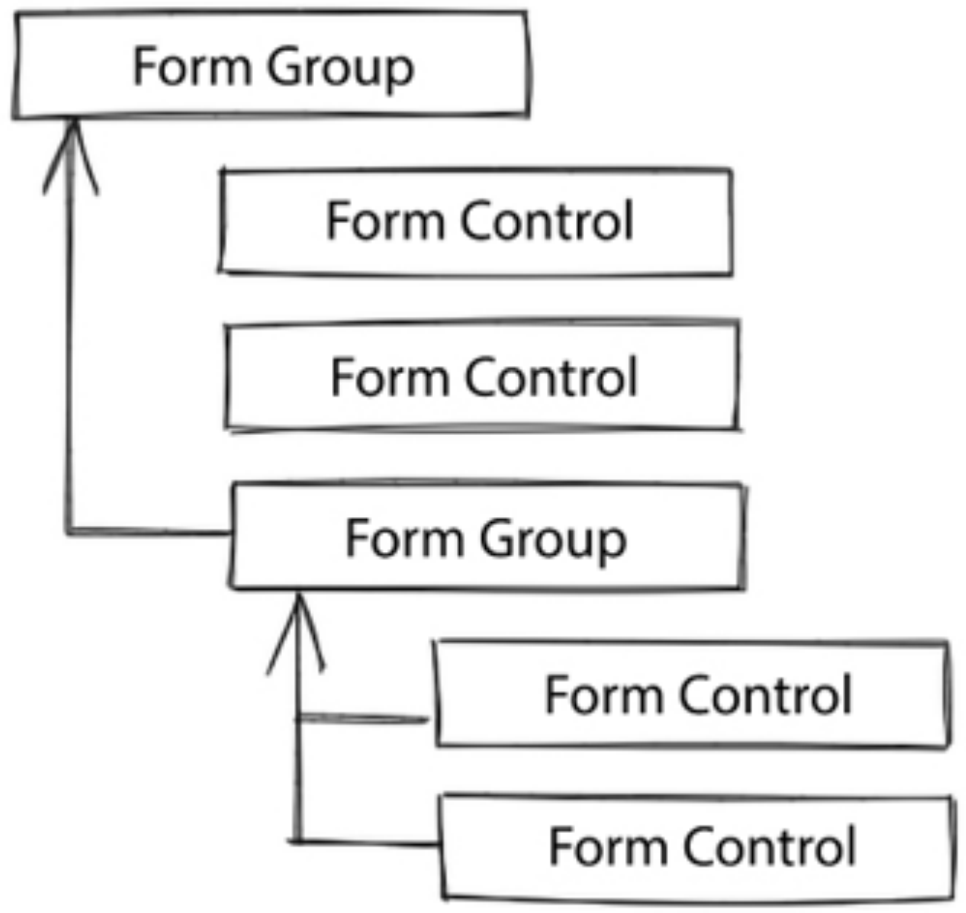


Figure 10.4: Status and value propagation in nested forms hierarchy

Nested hierarchies add a useful feature for Angular forms to the developer's toolchain when organizing large form structures. The status and value of each form propagate through the hierarchy to provide stability to our models.

So far, we have been using the constructor of `FormGroup` and `FormControl` classes to create an Angular form. However, it constitutes a lot of noise, especially in forms that contain many controls. In the following section, we will learn how to create Angular forms using an Angular service.

Creating elegant reactive forms

The Angular forms library exposes a service called `FormBuilder` that we can use to simplify form creation. We will learn how to use it by converting the form we created in the product create component:

1. Open the `product-create.component.ts` file and import the `FormBuilder` artifact from the `@angular/forms` npm package:

   ```
   import { FormBuilder, FormControl, FormGroup } from '@angular/forms';
   ```

2. Inject `FormBuilder` in the `constructor` of the `ProductCreateComponent` class:

```
constructor(private productsService: ProductsService, private builder: FormBuilder) {}
```

3. Convert the `productForm` property as follows:

```
productForm: FormGroup<{
    name: FormControl<string>,
    price: FormControl<number | undefined>
  }> | undefined;
```

4. Create the following `buildForm` method:

```
private buildForm() {
    this.productForm = this.builder.nonNullable.group({
      name: this.builder.nonNullable.control(''),
      price: this.builder.nonNullable.control<number | undefined>(undefined, {})
    });
  }
```

We use the `group` method of the `FormBuilder` service to group form controls together. We also use its `control` method to create the form controls. Notice that we also use the `nonNullable` property to indicate that the form and its controls are not nullable.

5. Make sure you use the non-null assertion operator in all references of the `productForm` property because it does not have an initial value anymore.

Using the `FormBuilder` service to create Angular forms, we don’t have to deal with the `FormGroup` and `FormControl` data types explicitly, although that is what is being created under the hood.

We have already seen how to define validation rules in a template by triggering a change of status using CSS styling. In the next section, we will learn how to define them in the component class and give visual feedback using appropriate messages.

Validating controls in a reactive way

We have already learned how to apply validation to the template of a form. We used the `required` attribute in the *Using reactive patterns in Angular forms* section to indicate that an input control needs to have a value. In reactive forms, the source of truth is our form model, so we need to be able to define validation rules when building the `FormGroup` instance.

To add validation rules, we use the second parameter of the `FormControl` constructor:

1. Open the `product-create.component.ts` file and import the `Validators` artifact from the `@angular/forms` npm package:

```
import { FormControl, FormGroup, Validators } from '@angular/forms';
```

2. Modify the declaration of the `productForm` property so that each `FormControl` instance passes `Validators.required` as a second parameter:

```
productForm = new FormGroup({
    name: new FormControl('', {
      nonNullable: true,
      validators: Validators.required
    }),
    price: new FormControl<number | undefined>(undefined, {
      nonNullable: true,
      validators: Validators.required
    })
  });
```

When we add a validator using the constructor of `FormControl`, we can remove the respective HTML attribute from the HTML template. However, it is recommended to keep it for accessibility purposes so that screen readers can understand how the form control should be validated.

3. The `Validators` class contains almost the same validator rules that are available for template-driven forms, such as the required validator. We can combine multiple validators by adding them to an array. To configure the price form control so that its value is at least 1, we use the `Validators.min` method:

```
productForm = new FormGroup({
    name: new FormControl('', {
      nonNullable: true,
      validators: Validators.required
    }),
    price: new FormControl<number | undefined>(undefined, {
      nonNullable: true,
      validators: [Validators.required, Validators.min(1)]
    })
  });
```

4. We can now use the status of validation rules and react to their changes. To disable the `<button>` element when the form is not valid, we need to bind the status of the form to the `disabled` button property in the `product-create.component.html` file:

```
<div>
    <button type="submit" [disabled]="!productForm.valid">Create</button>
  </div>
```

5. We can also display specific messages to the user upon changing the validity of each control:

```
<div>
    <label for="name">Name</label>
    <input id="name" formControlName="name" required />
    <span *ngIf="name.touched && name.invalid">
      The name is not valid
    </span>
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" required />
    <span *ngIf="price.touched && price.invalid">
      The price is not valid
    </span>
  </div>
```

6. It would be nice, though, if we could display different messages depending on the validation rule. We could display a more specific message when the price is less than 1. The `FormControl` class contains the `hasError` method, which accepts the validation property as a parameter and checks if the control has set the particular validation error:

```
<div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" required />
    <span *ngIf="price.touched && price.hasError('required')">
      The price is required
    </span>
    <span *ngIf="price.touched && price.hasError('min')">
      The price should be greater than 1
    </span>
```

```
</div>
```

The Angular framework provides a set of built-in validators that we can use in our forms. A validator is a function that returns either a `ValidationErrors` object or `null` when the control does not have any errors. According to the scenario, a validator can also return a value synchronously or asynchronously. In the following section, we will learn how to create a custom synchronous validator.

Building a custom validator

Sometimes, default validators won't cover all the scenarios we might encounter in an Angular application. It is easy to write a custom validator and use it in an Angular reactive form. In our case, we will build a validator to check whether the price of a product is in a predefined amount range.

We have already learned that a validator is a function that needs to return a `ValidationErrors` object with the error specified or a `null` value. Let's define such a function in a file named `price-range.directive.ts` inside the `src\app\products` folder:

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';
  export function priceRangeValidator(): ValidatorFn {
    return (control: AbstractControl<number>): ValidationErrors | null => {
      const inRange = control.value > 1 && control.value < 10000;
      return inRange ? null : { outOfRange: true };
    };
  }
```

The validator is a function that returns another function called the configured validator function. It accepts the form control object to which it will be applied as a parameter. If the value of the control does not fall into a defined price range, it returns a validation error object. Otherwise, it returns `null`.

The key of the validation error object specifies a descriptive name for the validator error. It is a name we can later check with the `hasError` method of the control to find out if it has any errors. The value of the validation error object can be any arbitrary value that we can pass in the error message.

To use our new validator, all we must do is import it into our product create component and add it to the `price` `FormControl` instance:

```
productForm = new FormGroup({
    name: new FormControl('', {
      nonNullable: true,
      validators: Validators.required
    }),
    price: new FormControl<number | undefined>(undefined, {
      nonNullable: true,
      validators: [Validators.required, priceRangeValidator()]
    })
  });
```

We removed the min validator from the price control because it is already checked from the functionality of our price range validator.

We can now modify the `price` field in the `product-create.component.html` file to display an appropriate error message if that specific error occurs:

```
<div>
    <label for="price">Price</label>
```

```
  <input id="price" formControlName="price" required />
  <span *ngIf="price.touched && price.hasError('required')">
    The price is required
  </span>
  <span *ngIf="price.touched && price.hasError('outOfRange')">
    The price is out of range
  </span>
</div>
```

Angular forms are not only about checking statuses but also about setting values. In the next section, we'll learn how to programmatically set values in a form.

Modifying forms dynamically

So far, we have used the `FormGroup` and `FormControl` classes extensively throughout this chapter. However, we have not seen what `FormArray` is all about.

Consider the scenario where we have added some products to the shopping cart of our e-shop application and want to update their quantities before checking out the order.

Currently, our application does not have any functionality for a shopping cart, so we will now add one:

1. Create a new service to manage the shopping cart by running the following Angular CLI command in the `src\app\cart` folder:

```
ng generate service cart
```

2. Open the `cart.service.ts` file and add the following `import` statement:

```
import { Product } from '../products/product';
```

3. Create a `cart` property in the `CartService` class and initialize it to an empty array:

```
export class CartService {
    cart: Product[] = [];
    constructor() { }
  }
```

The preceding `cart` property will be an intermediate local cache for storing selected products before checking out.

4. Add the following method to add a product to the cart:

```
addProduct(product: Product) {
    this.cart.push(product);
  }
```

5. Open the `product-detail.component.ts` file and add the following `import` statement:

```
import { CartService } from '../../cart/cart.service';
```

6. Inject `CartService` in the `ProductDetailComponent` class:

```
constructor(
    private productService: ProductsService,
    public authService: AuthService,
    private route: ActivatedRoute,
    private cartService: CartService
  ) { }
```

7. Modify the `buy` method to call the `addProduct` method of the `CartService` class:

```
buy(product: Product) {
    this.cartService.addProduct(product);
  }
```

8. Finally, open the `product-detail.component.html` file and modify the `Buy Now` button:

```
<button *ngIf="authService.isLoggedIn" (click)="buy(product)">Buy Now</button>
```

Now that we have implemented the basic functionality for storing the selected products that users want to buy, we need to modify the cart component for displaying the cart items:

1. Open the `cart.component.ts` file and add the following `import` statements:

```
import { FormArray, FormControl, FormGroup } from '@angular/forms';
  import { Product } from '../products/product';
  import { CartService } from './cart.service';
```

2. Create the following properties in the `CartComponent` class:

```
cartForm = new FormGroup({
    products: new FormArray<FormControl<number>>([])
  });
  cart: Product[] = [];
```

   In the preceding snippet, we created a `FormGroup` object containing a `products` property. We set the value of the `products` property to an instance of the `FormArray` class. The constructor of the `FormArray` class accepts a list of `FormControl` instances with type `number` as a parameter. For now, the list is empty since the cart does not have any products initially.

3. We have also created a `cart` property to store the details of the current shopping cart.
4. Inject `CartService` in the `constructor` of the `CartComponent` class:

```
constructor(private cartService: CartService) { }
```

5. Import the `OnInit` interface from the `@angular/core` npm package:

```
import { Component, OnInit } from '@angular/core';
```

6. Add the `OnInit` interface to the list of implemented interfaces of the `CartComponent` class:

```
export class CartComponent implements OnInit {
```

7. Add the following `ngOnInit` method:

```
ngOnInit(): void {
    this.cart = this.cartService.cart;
    this.cart.forEach(() => {
      this.cartForm.controls.products.push(
        new FormControl(1, { nonNullable: true })
      );
    });
  }
```

In the preceding method, we get the `cart` property from the `CartService` class and store it in the `cart` component property. We iterate through the products of the shopping cart and add the respective `FormControl` instances in the `FormArray`. We set the value of each form control to `1` to indicate that the cart contains one piece from each product by default.

8. Open the `cart.component.html` file and replace its HTML template with the following content:

```
<h2>My Cart</h2>
  <div [formGroup]="cartForm">
    <div
      formArrayName="products"
      *ngFor="let product of cartForm.controls.products.controls; let i=index">
        <label>{{cart[i].name}}</label>
        <input type="number" [formControlName]="i" />
    </div>
  </div>
```

In the preceding template, we use the `*ngFor` directive to iterate over the `controls` property of the `products` form array and create an `<input>` element for each one. We use the `index` keyword of the `*ngFor` directive to give a dynamically created name to each form control using the `formControlName` binding. We have also added a label that displays the product name from the `cart` component property. The product name is fetched using the index of the current product in the array.

9. Open the `app.module.ts` file and add the following `import` statements:

```
import { ReactiveFormsModule } from '@angular/forms';
  import { CommonModule } from '@angular/common';
```

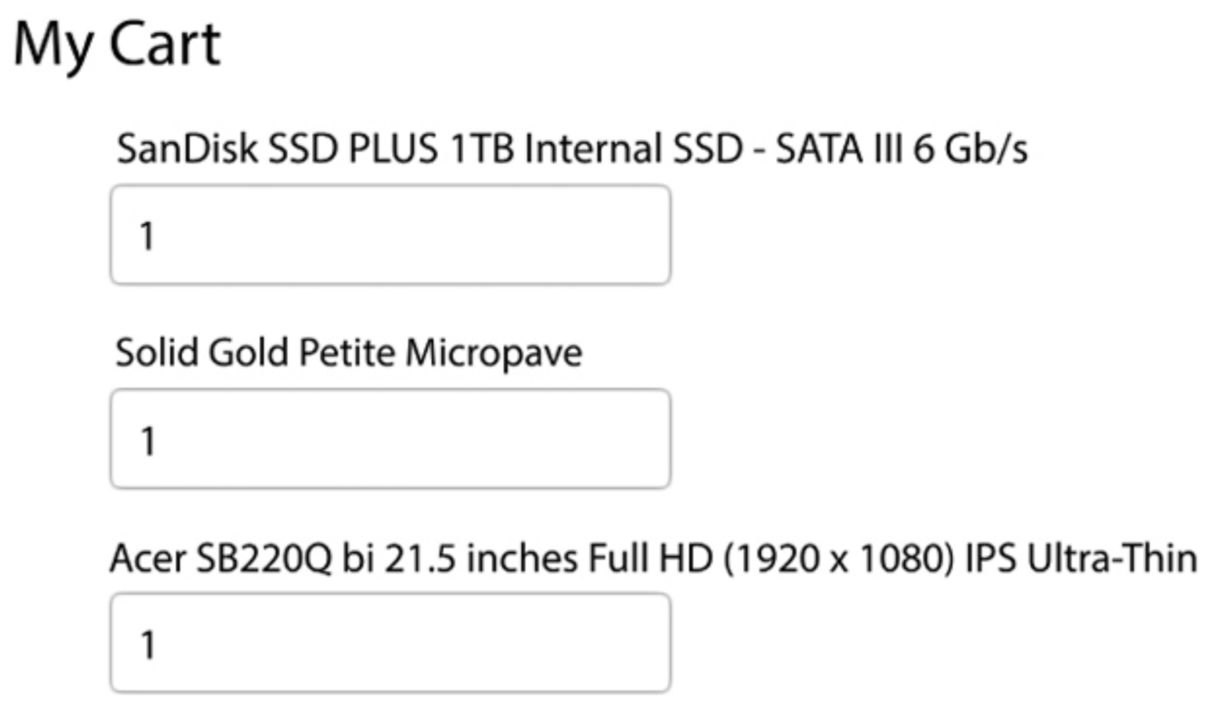
10. Add `ReactiveFormsModule` and `CommonModule` in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
    declarations: [
      AppComponent,
      CartComponent,
      PageNotFoundComponent
    ],
    imports: [
    BrowserModule,
    ProductsModule,
    AppRoutingModule,
    HttpClientModule,
    AuthModule,
    ReactiveFormsModule,
    CommonModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

To see the cart component in action, run the application using the `ng serve` command. Add some products to the cart by navigating to their details page and clicking the **Buy Now** button.

Do not forget to log in using the **Login** button because the functionality that adds a product to the cart is available only to authenticated users.

After you have added some products to the cart, click the **Cart** link to view your shopping cart. It should look like the following:


My Cart

SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s

1

Solid Gold Petite Micropave

1

Acer SB220Q bi 21.5 inches Full HD (1920 x 1080) IPS Ultra-Thin

1

Figure 10.5: Shopping cart

The real power of the `FormArray` class is that it can be used not only with `FormControl` instances but also with more complicated structures and other form groups.

With the `FormArray`, we have completed our knowledge range about the most basic building blocks of an

Angular form. In the next section, we'll learn how to use the reactive forms API and set values programmatically to an Angular form.

Manipulating form data

The `FormGroup` class contains two methods that we can use to change the values of a form programmatically:

- `setValue`: Replaces values in all the controls of the form
- `patchValue`: Updates values in specific controls of the form

The `setValue` method accepts an object as a parameter that contains key-value pairs for all form controls. If we want to fill in the details of a product in the product create component programmatically, we should use the following snippet:

```
this.productForm.setValue({
    name: 'New product',
    price: 150
  });
```

In the preceding snippet, each key of the object passed in the `setValue` method must match the name of each control in the form. If we omit one, Angular will throw an error.

If, on the contrary, we want to fill in some of the details of a product, we can use the `patchValue` method:

```
this.productForm.patchValue({
    price: 150
  });
```

The `setValue` and `patchValue` methods of the `FormGroup` class help us set data in a form. Another interesting aspect of reactive forms is that we can also be notified when these values change, as we will see in the following section.

Watching state changes and being reactive

We have already learned how to create forms programmatically and specify all our fields and their validations in the code. A reactive form can listen to changes in the form controls when they happen and react accordingly. A suitable reaction could be to disable/enable a control, provide a visual hint, or do something else according to your needs.

How can we make this happen? A `FormControl` instance contains two observable properties: `statusChanges` and `valueChanges`. The first one notifies us when the status of the control changes, such as going from invalid to valid. On the other hand, the second one notifies us when the value of the control changes. Let’s explore this one in more detail, using an example.

The **Price** field in the form of the product create component contains a custom validator to check if the price is within a valid range. From an end-user point of view, it would be better to display a hint about this validation as soon as the user has started entering values in the field:

1. First, add a `<span>` element in the `product-create.component.html` file to contain an appropriate hint message:

```
<div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" required />
    <span *ngIf="price.touched && price.hasError('required')">
      The price is required
    </span>
    <span *ngIf="price.touched && price.hasError('outOfRange')">
      The price is out of range
    </span>
    <span *ngIf="showPriceRangeHint">
      Price should be between 1 and 10000
    </span>
  </div>
```

   The hint will be displayed according to the `showPriceRangeHint` property of the component.

2. Open the `product-create.component.ts` file and import the `OnInit` artifact:

```
import { Component, OnInit, EventEmitter, Output } from '@angular/core';
```

3. Add the `OnInit` artifact in the list of the `ProductCreateComponent` class implemented interfaces:

```
export class ProductCreateComponent implements OnInit {
```

4. Create the `showPriceRangeHint` property in the `ProductCreateComponent` class:

```
showPriceRangeHint = false;
```

5. Create the following `ngOnInit` method to subscribe to the `valueChanges` property of the `price` form control:

```
ngOnInit(): void {
    this.price.valueChanges.subscribe(price => {
      if (price) {
        this.showPriceRangeHint = price > 1 && price < 10000;
      }
    });
```

```
  }
```

In the preceding method, we check if the price entered is within a valid range and set the `showPriceRangeHint` property appropriately.

The `valueChanges` and `statusChanges` properties in a `FormControl` instance are standard observable streams. Do not forget to unsubscribe from them when the component is destroyed.

Of course, there is more that we can do with the `valueChanges` observable. For example, we could check if the product name is already reserved by sending it to a backend server, but this code shows off the reactive nature. Hopefully, this has conveyed how you can take advantage of the reactive nature of forms and respond accordingly.

Summary

In this chapter, we have learned that Angular provides two different flavors for creating forms – template-driven and reactive forms – and that neither approach can be said to be better than the other. We have merely focused on reactive forms because of their many advantages and learned how to build them. We have also covered the different types of validations and now know how to create our custom validations. We also learned how to fill our forms with values and get notified when they change.

In the next chapter, we will learn how to skin our application to look more beautiful with the help of Angular Material. Angular Material has many components and styling ready for you to use in your projects. So, let's give your Angular project the love it deserves.

End of Chapter 10