7 Laying out pages with Flex Layout

This chapter covers

- Implementing responsive web design using the Flex Layout library
- Using the `ObservableMedia` service
- Changing the layout based on the viewport size

When it comes to developing a web app, you need to decide whether you'll have separate apps for desktop and mobile versions or reuse the same code on all devices. The former approach allows you to use the native controls on mobile devices so the UI looks more natural, but you need to maintain separate versions of the code for each app. The latter approach is to use a single code base and implement *responsive web design* (RWD) so the UI layout will adapt to the device screen size.

NOTE The term RWD was coined by Ethan Marcotte in the article "Responsive Web Design," available at http://alistapart.com/article/responsive-web-design.

There's a third approach: in addition to your web application that works on desktops, develop a *hybrid* application, which is a web application that works inside the mobile browser but can invoke the native API of the mobile device too.

In this chapter, you'll see how to make your app look good and be functional on large and small screens using the RWD approach. Chapter 6 covered observables that can push notifications when certain important events happen in your app. Let's see if you can use observables to let you know if the user's screen size changes and change the UI layout based on the width of the viewport of the user's device. Users with smartphones and users with large monitors should see different layouts of the same app.

We'll show you how to use the Flex Layout library for implementing RWD and how to use its `ObservableMedia` service to spare you from writing lots of CSS.

Finally, you'll start rewriting the ngAuction app, illustrating many of the techniques you've learned, with the main goal to remove Bootstrap from the app, using only the Angular Material and Flex Layout libraries.

7.1 Flex Layout and ObservableMedia

Imagine that you've laid out the UI of your application, and it looks great on a user's monitor with a width resolution of 1200 pixels or more. What if the user opens this app on a smartphone with a viewport width of 640 pixels? Depending on the device, it may either render only a part of your app's UI, adding a horizontal bar at the bottom, or scale down the UI so it fits in a small viewport, making the app difficult to use. Or consider another scenario: users with large monitors who reduce the width of their browser window because they need to fit another app on their monitor.

To implement RWD, you can use CSS media queries, represented by the `@media` rule. In the CSS of your app, you can include a set of media queries offering different layouts for various screen widths. The browser constantly checks the current window width, and as soon as the width crosses a *breakpoint* set in the `@media` rules (for example, the width becomes smaller than 640 pixels), a new page layout is applied.

Another way to implement flexible layouts is by using CSS Flexbox with media queries (see http://mng.bz/6B42). The UI of your app is styled as a set of flexible boxes, and if the browser can't fit the Flexbox content horizontally (or vertically), the content is rendered in the next row (or column).

You can also implement RWD with the help of CSS Grid (see http://mng.bz/k29F). Both Flexbox and CSS Grid require a good understanding of `@media` rules.

The Angular Flex Layout library (see https://github.com/angular/flex-layout) is a UI layout engine for implementing RWD without writing media queries in your CSS files. The library provides a set of simple Angular directives that internally apply the rules of the `flexbox` layout and offer you the `ObservableMedia` service that notifies your app about the current width of the viewport on the user's device.

Angular Flex Layout has the following advantages over the standard CSS API:

- It produces cross-browser-compatible CSS.
- It provides an Angular-friendly API for dealing with media queries using directives and observables.

> **NOTE** In this section, we provide a minimal description of the Flex Layout library to get you started quickly. For more details and demos, refer to the Flex Layout documentation at https://github.com/angular/flex-layout/wiki.

The Flex Layout library provides two APIs: static and responsive. The static API allows you to use directives to specify layout attributes for containers and their children. The responsive API enhances static API directives, enabling you to implement RWD so app layouts change for different screen sizes.

7.1.1 Using Flex Layout directives

There are two types of directives in the Flex Layout library: one for containers and one for their child elements. A container's directives are used to align its children. Child directives are applied to child elements of a container managed by Flex Layout. With child directives, you can specify the order of each child, the amount of space it takes, and some other properties, as shown in table 7.1.

Table 7.1 Frequently used Flex Layout directives

Directive	Description
Container directives	
`fxLayout`	Instructs the element to use CSS Flexbox for laying out child elements.
`fxLayoutAlign`	Aligns child elements in a particular way (to the left, to the bottom, evenly distribute, and so on). Allowed values depend on the `fxLayout` value attached to the same container element—see Angular Flex Layout documentation.
`fxLayoutGap`	Controls space between child elements.
Child directives	
`fxFlex`	Controls the amount of space a child element takes within the parent container.
`fxFlexAlign`	Allows selectively changing a child's alignment within the parent container prescribed by the `fxLayoutAlign` directive.
`fxFlexOrder`	Allows changing the order of a child element within the parent container. For example, it can be used to move an important component to the visible area when switching from desktop to a mobile screen.

> **NOTE** Child directives expect to be inside an HTML element with a container directive attached.

Let's take a look at how to use the Flex Layout library to align two `<div>` elements next to each other in a row. First, you need to add the Flex Layout library and its peer dependency, `@angular/cdk`, to your project:

```
npm i @angular/flex-layout @angular/cdk
```

The next step is to add the `FlexLayoutModule` to the root `@NgModule()` decorator, as shown in the following listing.

Listing 7.1 Adding the `FlexLayoutModule`

```
import {FlexLayoutModule} from '@angular/flex-layout';

@NgModule({
  imports: [
    FlexLayoutModule
    //...
  ]
})
export class AppModule {}
```

The next listing creates a component that displays the `<div>` elements next to each other from left to right.

Listing 7.2 flex-layout/app.component.ts

```
@Component({
  selector: 'app-root',
  styles: [`
    .parent {height: 100px;}
    .left   {background-color: cyan;}
    .right  {background-color: yellow;}
  `],
  template: `
    <div class="parent" fxLayout="row" >
       <div fxFlex class="left">Left</div>
       <div fxFlex class="right">Right</div>
     </div>
  `
})
export class AppComponent {}
```

The fxLayout directive turns the into a flex-layout container where children are allocated horizontally (in a row).

The fxFlex directive instructs each child element to take equal space within the parent container.

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

```
ng serve --app flex-layout -o
```

Figure 7.1 shows how the browser renders the child elements. Each child takes 50% of container's available width.

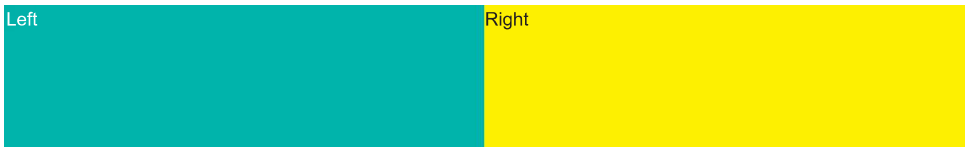


Figure 7.1 Two elements aligned in a row

To make the right div take more space than the left one, you can assign the required space values to the child `fxFlex` directives. The following template uses the child-level directive `fxFlex` to allocate 30% of the available width to the left child and 70% to the right one:

```
<div fxLayout="row" class="parent">
  <div fxFlex="30%" class="left">Left</div>
  <div fxFlex="70%" class="right">Right</div>
</div>
```

Now the UI is rendered as shown in figure 7.2.

Figure 7.2 The right element takes more space than the left one.

To lay out the container's children vertically, change the direction of the container's layout from rows to columns, as in `fxLayout="column"`:

```
<div fxLayout="column" class="parent">
  <div fxFlex="30%" class="left">Left</div>
  <div fxFlex="70%" class="right">Right</div>
</div>
```

Figure 7.3 shows how the child elements are rendered vertically.

Figure 7.3 Column layout of the container element

Say that on a wide screen you have enough room to render your left and right components horizontally next to each other, but if the user opens the same app on a smaller screen, you want to automatically change the layout to vertical so the right component is shown under the left one.

Each directive in the Flex Layout library can optionally have a *suffix* (an alias to the media query rule) that specifies which screen size it has to apply. For example, the `flexLayout.sm` directive has the suffix `.sm`, which means it should be applied only

when the screen width is small. These aliases correspond to the width breakpoints defined in the Material Design guidelines (see http://mng.bz/RmLN):

- `xs`—Extra small (less than 599 px)
- `sm`—Small (560–959 px)
- `md`—Medium (960–1279 px)
- `lg`—Large (1280–1919 px)
- `xl`—Extra large (1920–5000 px)

The next listing changes your app so its parent container lays out its children horizontally on medium and large screens, and vertically on small devices.

Listing 7.3 Adding the `.sm` suffix

```
<div class="parent"
        fxLayout="row"
         fxLayout.sm="column" >
       <div fxFlex="30%" class="left">Left</div>
      <div fxFlex="70%" class="right">Right</div>
    </div>
```

Child elements are aligned in a row by default.

On small screen sizes, child elements are aligned vertically.

To illustrate how this will change the layout, you'll use Chrome Dev Tools, which has an icon on the left side of its toolbar that allows you to toggle devices. For desktops, the small size means that the width of the window is between 600 and 959 pixels. Figure 7.4 shows how the UI is rendered if the width is 960 (still the medium size).

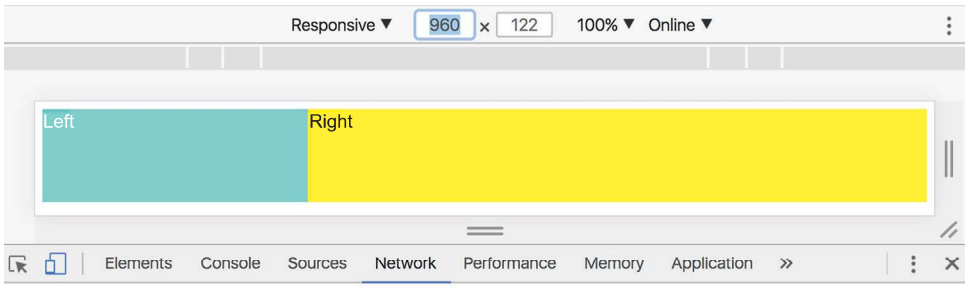


Figure 7.4 Rendering on a medium device with a width of 960 pixels

Let's cross the breakpoint and change the width to 959 to emulate a small device. Figure 7.5 shows that the layout has changed from horizontal to vertical.

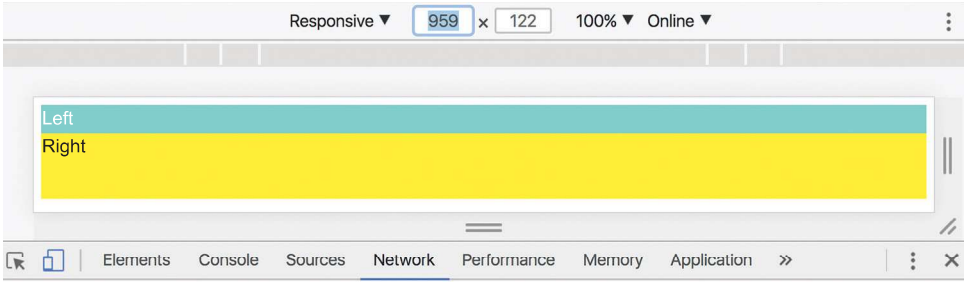


Figure 7.5 Rendering on a small device with a width of 959 pixels

Changing the width to anything smaller than 600 will cause a switch back to the horizontal layout, because you haven't specified that for extra-small devices (the `.xs` suffix), the layout should remain vertical. You can add the vertical layout for extra small (`xs`) devices:

```
<div fxLayout="row" class="parent"
     fxLayout.sm="column"
     fxLayout.xs="column">
```

You can also apply less-than (`lt-`) and greater-than (`>-`) suffixes to the media query aliases. For example, if you use the `lt-md` alias, the respective layout will be applied to the small and extra-small screens. In your app, you can specify that on any screen with a width less than medium, the column layout should be applied:

```
<div fxLayout="row" class="parent"
     fxLayout.lt-md="column">
```

Using breakpoints, you can statically define how your UI should be laid out in the component's template. What if you want not only to change the layout inside the container, but also to conditionally show or hide certain children, depending on the screen size? To dynamically decide what and how the browser should render depending on the screen size, you'll use the `ObservableMedia` service, which comes with the Flex Layout library.

7.1.2 ObservableMedia service

The `ObservableMedia` service enables subscribing to screen-size changes and programmatically changing the look and feel of your app. For example, on large screens, you may decide to display additional information. To avoid rendering unnecessary components on small screens, you may subscribe to events emitted by `ObservableMedia`, and if the screen size becomes larger, you can render more components.

To implement this functionality, import the `ObservableMedia` service and subscribe to its `Observable` object. The following listing shows how to subscribe to notifications about screen-size changes with the `async` pipe and print the current size on the console.

Listing 7.4 observable-media/app.component.ts

```
import {Component} from '@angular/core';
import {ObservableMedia} from '@angular/flex-layout';
import {Observable} from 'rxjs';
import {map} from 'rxjs/operators';

@Component({
  selector: 'app-root',
  template: `<h3>Watch the breakpoint activation messages in the console.
  </h3>
  <span *ngIf="showExtras$ | async">
   Showing extra info on medium screens</span>
  `
})
export class AppComponent {
  showExtras$: Observable<boolean>;

  constructor(private media: ObservableMedia) {
     this.showExtras$ = this.media.asObservable()
       .pipe(map(mediaChange => {
           console.log(mediaChange.mqAlias);
           return mediaChange.mqAlias === 'md'? true: false;
           })
      );
  }
}
```

Shows/hides text based on the value of showExtras$; the async pipe subscribes to showExtras$

Injects the ObservableMedia service

Subscribes to the Observable that emits values when the screen size changes

showExtras$ emits true if the screen is medium.

NOTE Note the use of the `*ngIf` structural directive. If the `showExtras$` observable emits `true`, the span is added to the DOM. If it emits `false`, the span is removed from the DOM.

The values emitted by `media.asObservable()` have the type `MediaChange` that includes the `mqAlias` property, which holds the value representing the current width—`lg` for large or `md` for medium.

To see listing 7.4 in action, run the following command, and open the browser's console:

```
ng serve --app observable-media -o
```

You'll see the text "Showing extra info on medium screens" when the screen size is `md` (medium). Reduce the width of the browser window to the `sm` size, and this text will be hidden. To see the current CSS media query and other properties of the `MediaChange` class, change the log statement to `console.log(mediaChange);`.

In listing 7.4, you explicitly declared a `showExtras$` observable and subscribed to it because you wanted to monitor `MediaChange`. But this code can be simplified by using the `ObservableMedia.isActive()` API, as shown in the following listing.

Listing 7.5 Using the `ObservableMedia.isActive()` API

```
import {Component} from '@angular/core';
import {ObservableMedia} from '@angular/flex-layout';
```

```
@Component({
  selector: 'app-root',
  template: `<h3>Using the ObservableMedia.isActive() API</h3>
  <span *ngIf="this.media.isActive('md')">
    Showing extra info on medium screens</span>
  `
})
export class AppComponent {
  constructor(public media: ObservableMedia) {}
}
```

Shows text only if the current viewport width is md

In the hands-on section later in this chapter, you'll create the new version of ngAuction that will implement RWD using the Flex Layout library and `ObservableMedia`.

Other options for implementing RWD

The Flex Layout library may be appealing to beginners because it's simple to use. But it's not the only solution for creating responsive layouts in Angular apps. Here are some other options:

- The Angular CDK (component development kit) package includes the layout module. After installing the `@angular/cdk` package, you can use the `LayoutModule` and `BreakpointObserver` or `MediaMatcher` classes that monitor changes in the viewport size. Besides, because Angular CDK is a peer dependency of Flex Layout, by working directly with Angular CDK, you'll use one library instead of two.
- At the time of writing, the Flex Layout library remains in beta, and its creators often introduce breaking changes with new beta releases. If you're not using the latest version of Angular, Flex Layout may not support the version of Angular you use. For example, the Flex Layout library doesn't support Angular 4.

To minimize the number of libraries used in your app, consider implementing RWD by using CSS Flexbox and CSS Grid. Also, using CSS that's natively supported by the browser will always be more performant than using any JavaScript library. We recommend the free CSS Grid video course by Wes Bos, available at https://cssgrid.io.

7.2 Hands-on: Rewriting ngAuction

Starting in this chapter, you'll rewrite ngAuction from scratch. The new ngAuction will be written using Angular Material from the get-go and will include images, not just the grey rectangles. The search component will be represented by a small icon on the toolbar, and you'll add shopping cart functionality. Users will be able to bid on products and buy them if they place winning bids.

7.2.1 Why rewrite the ngAuction app from scratch?

You may be thinking, "We already worked on ngAuction in chapters 2, 3, and 5. Why not just continue building the same app?" In the first chapters, the goal was to gently

introduce you to the main artifacts of the Angular framework without overloading you with information on application architecture, implementing RWD, and customizing themes.

The ngAuction app developed in the previous chapters served that goal well. This rewrite will showcase the best development practices for real-world Angular applications. You want to accomplish the following:

- Create a modularized app where each view is a lazy-loaded module.
- Use Angular Material for the UI, illustrating theme customization with SaaS.
- Use the Flex Layout library.
- Remove the dependency on the Bootstrap and JQuery libraries.
- Remove the search box from the landing page to make better use of screen space.
- Keep shared components and services in a separate folder.
- Illustrate state management using injectable services and then reimplement it using the NgRx library.
- Create scripts for unit and end-to-end testing.

You're not going to implement all of that in this chapter, but you'll get started.

In this app, you'll implement RWD using the Flex Layout library and its `ObservableMedia` service, introduced earlier. On large screens, the landing page of ngAuction will display four products per row, as shown in figure 7.6.

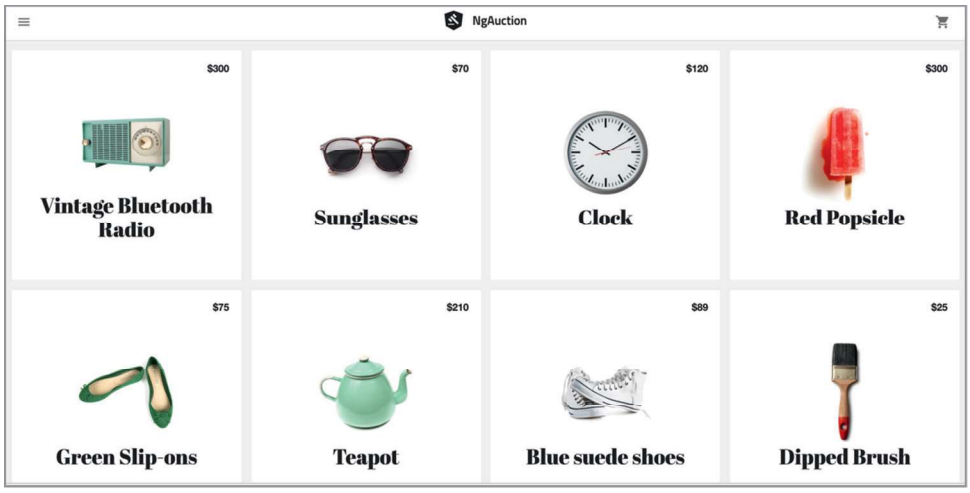


Figure 7.6 Rendering ngAuction on large screens

NOTE We borrowed the data and images from the Google app illustrating the Polymer library (see http://mng.bz/Y5d9).

The app will be subscribed to the `ObservableMedia` service using the `async` pipe and will automatically change the layout to three products per row as soon as the width of the window changes to a medium size, as shown in figure 7.7.

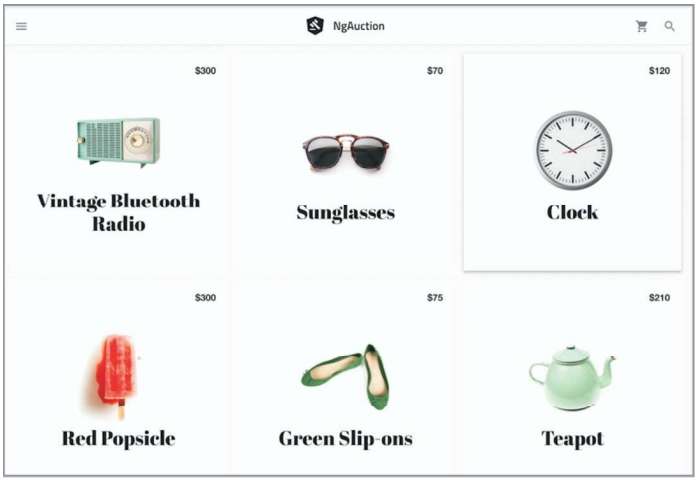


Figure 7.7 Rendering ngAuction on medium screens

On small screens, the app will change to the two-column layout, as shown in figure 7.8.

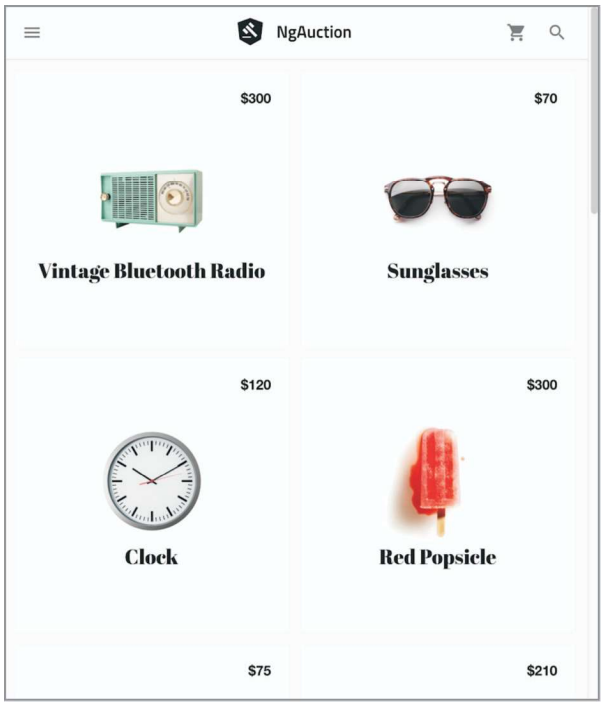


Figure 7.8 Rendering ngAuction on small screens

The app will also change its layout when rendered on extra-small (one-column layout) and extra-large screens (five-column layout).

7.2.2 Generating a new ngAuction app

> **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.

This time, you'll generate the project using the Angular CLI `new` command with options. The new ngAuction will use the Sass preprocessor for styles with the SCSS syntax. You also want to specify the `nga-` prefix, so each newly generated component will have this prefix in its selector:

```
ng new ng-auction --prefix nga --style scss
```

> **NOTE** We discuss the benefits of using SCSS in the next section, "Creating a custom Angular Material theme with Sass."

Change to the ng-auction directory and run the following commands to add Angular Material and Flex Layout libraries to the project:

```
npm install @angular/material @angular/cdk
npm i @angular/flex-layout
```

Installs the Angular Material library and Component Development Kit. The Angular Material library also requires the animations package, which was already installed by Angular CLI during the project generation.

Installs the Flex Layout library

The Angular Material library comes with four prebuilt themes, and you had a chance to try one of them in section 5.6.1 in chapter 5. But what if none of the prebuilt themes fits your UI requirements?

7.2.3 Creating a custom Angular Material theme with Sass

If you want to create a custom Angular Material theme for your app, read the Theming Guide at https://material.angular.io/guide/theming. In this section, we'll just give you a code review of the .scss files that we created to customize the theme for ngAuction.

When you generated the ngAuction app, you used the option `--style scss`. By doing so, you informed Angular CLI that you're not going to use CSS files, but instead will use the Syntactically Awesome Style Sheets, also known as Sass (see http://sass-lang.com). Sass is an extension to CSS with its own preprocessor. Some Sass benefits include the following:

- *Variables*—Assigning styles to variables and reusing them in multiple stylesheets
- *Nesting*—An easy-to-write and -read syntax for nested CSS selectors
- *Mixins*—Blocks of styles that can include variables

Sass provides two syntaxes, Sass and SCSS, and you'll use the latter in this book. If you were installing SaaS separately, you'd need to run your .scss files through the preprocessor to compile them into regular .css files before deployment. But Angular CLI supports Sass out of the box, so the preprocessor does its job during bundling.

SCSS syntax

Here's a quick introduction to SCSS syntax:

- *Variables*—A variable name starts with the dollar sign. The following code snippet declares and uses the variable `$font-stack`:

```
$font-stack: Helvetica, sans-serif;

body {
  font: 100% $font-stack;
}
```

  This variable can be used in multiple places, and if you decide to change the Helvetica font to another one, you do it in one place instead of making changes in each and every .css file where you used it.
- *Nesting*—It's an easy-to-read syntax for writing nested CSS selectors. The following sample shows how to nest the `ul` and `a` style selectors inside the `div` selector:

```
div {
  ul {
    margin: 0;
  }
  a {
    display: block;
  }
}
```

- *Mixins*—A mixin is a block of Sass style. A mixin can be added to your styles with `@include`. Mixins can also use variables and can be invoked as functions with arguments, as in `mat-palette($mat-red);`.
- *Partials*—Partials are just files with code fragments meant to be imported by other Sass files. Partials must have names that start with an underscore, such as _theme.scss. When you import a partial, the underscore isn't needed, as in `@import './theme';` Partials aren't compiled into separate CSS files—their content is compiled only as a part of .scss files that import them.
- *Imports*—The `@import` statement allows you to import styles located in other files. Although CSS also has an `@import` keyword, it makes an additional HTTP request for each file. With Sass, all imports are combined into a single CSS file during preprocessing, so only one HTTP request is needed to load the CSS.

In your ngAuction app, you'll create the styles directory, move the generated styles.scss file there, and add one more partial, _theme.scss. The content of _theme.scss is shown in the following listing. You use the `$mat-cyan` palette defined in the imported file _theming.scss.

Listing 7.6 _theme.scss

```
@import '~@angular/material/theming';

$nga-primary: mat-palette($mat-cyan);
$nga-accent:  mat-palette($mat-cyan, A200, A100, A400);
$nga-warn:    mat-palette($mat-red);

$nga-theme:   mat-light-theme($nga-primary, $nga-accent, $nga-warn);

$nga-background: map-get($nga-theme, background);
$nga-foreground: map-get($nga-theme, foreground);

$nga-typography: mat-typography-config();
```

Declares a variable for the primary palette and initializes it with $mat-cyan palette

Declares and initializes a variable for the accent palette specifying a default, lighter, and darker hue of $mat-cyan

Declares and initializes a variable for the warning palette

Creates the theme (Sass object containing all palettes)

Declares and initializes the variable for typography

Declares and initializes a variable for the foreground palette

Declares and initializes a variable for the background palette

In the _theme.scss file, you used the cyan color for the primary and accent palettes. You can find their definitions in node_modules/@angular/material/_theming.scss.

In the following listing, you add styles in styles.scss, starting from importing the preceding _theme.scss.

Listing 7.7 styles.scss

```
@import './theme';

@import url('https://fonts.googleapis.com/icon?family=Material+Icons');
 @import url('https://fonts.googleapis.com/css?family=Titillium+Web:600');
 @import url('https://fonts.googleapis.com/css?family=Abril+Fatface');

// Be sure that you only ever include this mixin once!
@include mat-core();

@include angular-material-theme($nga-theme);

// Global styles.
html {
  -moz-osx-font-smoothing: grayscale;
  -webkit-font-smoothing: antialiased;
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
```

Imports Google Material icons

Imports the Titillium Web fonts (you'll use it for the toolbar title and later for bid values)

Imports the Abril Fatface fonts (you'll use it for product titles)

Imports Angular Material core styles that aren't theme-dependent

Loads your custom theme configured in _theme.scss

```
  box-sizing: border-box;
  height: 100%;
}

body {
  color: #212121;
  background-color: #f3f3f3;
  font-family: mat-font-family($nga-typography);
  line-height: mat-line-height($nga-typography, body-1);
  font-size: mat-font-size($nga-typography, body-1);
  height: 100%;
  margin: 0;
}
```

The styles.scss and _theme.scss files define your global styles for the entire app, and you'll specify them in the `styles` property in the .angular-cli.json file. In ngAuction, you'll also be styling individual components, and _theme.scss will be reused in each component. We've intentionally broken the style definition into two files so you can reuse _theme.scss (just the variables definitions) in components without duplicating the core styles, images, and fonts used in styles.scss.

Now your custom theme is configured, and you can start working on the UI of the landing page of ngAuction.

7.2.4 Adding a toolbar to the top-level component

Figure 7.6 shows the landing page of ngAuction, which includes the Material toolbar and the `HomeComponent`. To be more precise, it includes the toolbar and a `<router-outlet>` tag where you render the `HomeComponent`. Let's start with creating the first version of the toolbar. This toolbar will include the menu icon on the left, the logo of ngAuction in the middle, and the shopping cart icon on the right. It won't include the Search button (you'll add that in section 11.8 in chapter 11) and will look like figure 7.9.

Figure 7.9 The toolbar

On the left, you use the Google Material icon `menu`, and on the right, `shopping_cart`. For the logo, you place the Google Material `gavel` icon on top of the shape resembling the Angular logo, and save it in the logo.svg file, included with the book's source code.

As you learned in the hands-on section of chapter 5, to use Angular Material components, you should include the corresponding modules in the `imports` section of the root module of your app. In your toolbar, you'll need `MatToolbarModule`, `MatButtonModule`, and `MatIconModule`. Since you're going to use the Flex Layout library, you'll also need to add `FlexLayoutModule` to the root module. Later in this section, you'll

use `HttpClient` to read the product data, so you need to add the `HttpClientModule` to the root module.

Update the CLI-generated app.module.ts to include the modules in the following listing.

Listing 7.8 app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {MatButtonModule} from '@angular/material/button';
import {MatIconModule} from '@angular/material/icon';
import {MatToolbarModule} from '@angular/material/toolbar';
import {FlexLayoutModule} from '@angular/flex-layout';
import {HttpClientModule} from '@angular/common/http';
import {AppComponent} from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    MatButtonModule,
     MatIconModule,
     MatToolbarModule,
     FlexLayoutModule,
     HttpClientModule
   ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Adds the required modules from the Angular Material library

Adds the Flex Layout module

Adds HttpClientModule—you'll use HttpClient for getting the product data

Replace the generated app.component.html with the following listing.

Listing 7.9 app.component.html

```
<mat-toolbar class="toolbar">
  <button class="toolbar__icon-button" mat-icon-button>
     <mat-icon>menu</mat-icon>
  </button>

  <div class="toolbar__logo-title-group"
       fxLayout
       fxLayoutAlign="center center">
     <a routerLink="/"
        fxLayout>
      <img class="toolbar__logo"
           src="/assets/logo.svg"
           alt="NgAuction Logo">
    </a>
    <a class="toolbar__title"
       routerLink="/">NgAuction</a>
  </div>
```

The menu button with the icon

Displays the logo in the center of the toolbar

Turns the logo into a clickable link that will display HomeComponent after you configure routes

Turns the text ngAuction into a clickable link

```
    <div fxFlex></div>

    <button mat-icon-button class="toolbar__icon-button
                    toolbar__shopping-cart-button">
      <mat-icon>shopping_cart</mat-icon>
    </button>

</mat-toolbar>

<!--<router-outlet></router-outlet>-->
```

A filler to push the shopping cart icon to the right

The shopping cart button with an icon

You'll keep the router outlet commented out until you configure routes.

To make the toolbar look like figure 7.9, you need to add the following listing's styling to the app.component.scss file.

Listing 7.10 app.component.scss

```
@import '../styles/theme';

:host {
   display: block;
  height: 100%;
}

.toolbar {
   background-color: mat-color($nga-background, card);
  position: relative;
  box-shadow: 0 1px mat-color($nga-foreground, divider);
}

.toolbar__logo-title-group {
   position: absolute;
  right: 50%;
  left: 50%;
}

.toolbar__logo {
   height: 32px;
  margin-right: 16px;
}

.toolbar__title {
   color: mat-color($nga-foreground, text);
  font-family: 'Titillium Web', sans-serif;
  font-weight: 600;
  text-decoration: none;
}

.toolbar__icon-button {
   color: mat-color($nga-foreground, icon);
}

.toolbar__shopping-cart-button {
   margin-right: 8px;
}
```

Imports your custom theme

Uses Angular pseudo selector :host to style the component that hosts the AppComponent

Applies the same background as in the Material card component in this theme (it's white in your theme)

Styles the logo name

Styles the logo image

Styles the toolbar title

Styles the icon foreground

Styles the shopping cart button

Running the `ng serve` command will render the ngAuction app that looks like figure 7.9.

You have a toolbar UI rendered, and now you need to show products under the toolbar. First, you need to create the `ProductService` that will provide the product data, and then you'll create the `HomeComponent` that will render the data. Let's start with the `ProductService`.

7.2.5 Creating the product service

The product service needs data. In real-world apps, the data would be supplied by the server, and you'll do that in chapter 12. For now, you'll just use the JSON file that contains the information about the product. The product images will be located on the client side as well. The code samples that come with the book include the src/data/products.json file, of which a fragment is shown in the following listing.

Listing 7.11 A fragment from src/data/products.json

```
[
  {
    "id": 1,
    "description" : "Isn't it cool when things look old, but they're not...",
    "imageUrl" : "data/img/radio.png",
    "price" : 300,
    "title" : "Vintage Bluetooth Radio"
  },
  {
    "id": 2,
    "description" : "Be an optimist. Carry Sunglasses with you at all times..
     .",
    "featured" : true,
    "imageUrl" : "data/img/sunnies.png",
    "price" : 70,
    "title" : "Sunglasses"
  }
  ...
  ]
```

This file includes URLs of the product images located in the data/img folder. If you're following along and are trying to build ngAuction by yourself, copy the src/data directory into your project from the code that comes with the book and add the line `"data"` to the app property `assets` in the .angular-cli.json file.

You'll use the `ProductService` class in more than one component; you'll generate it in the folder src/app/shared/services. You'll be adding other reusable services in this folder later on (such as `SearchService`). You'll generate `ProductService` using the following Angular CLI command:

```
ng generate service shared/services/product
```

Then you'll add the provider for this service to the app.module:

```
...
import {ProductService} from './shared/services/product.service';

@NgModule({
  ...
  providers: [ProductService]
})
export class AppModule {}
```

Best practice

The import statement for `ProductService` is rather long, and it points at the file where this service is implemented. As your application grows, the number of services as well as the number of import statements in your module increases, polluting the module code.

Create the file named index.ts in the services folder like so:

```
import {Product, ProductService} from './product.service';

export {Product, ProductService} from './product.service';
```

You import the `Product` and `ProductService` classes and reexport them right away. Now the import statement in the app.module can be simplified to look like this:

```
import {Product, ProductService} from './shared/services';
```

If you have just one reexported class, this may look like overkill. But if you have multiple classes in the services folder, you can write just one import statement for all classes, functions, or variables that you want to import—for example:

```
import { ProductService, Product, SearchService } from './shared/services';
```

Keep in mind, this will work only if the file with such reexports is called index.ts.

The product.service.ts file includes the `Product` interface and the `ProductService` class. The `Product` interface defines the type of objects returned by the methods of the `ProductService` class: `getAll()` and `getById()`. The code for your `ProductService` is shown in the following listing.

Listing 7.12 product.service.ts

```
import {Injectable} from '@angular/core';
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs';
import {map} from 'rxjs/operators';

export interface Product {        <-- Defines the Product type
   id: number;
  title: string;
  price: number;
  imageUrl: string;
```

```
  description: string;
}

@Injectable()
export class ProductService {
  constructor(private http: HttpClient) {}

  getAll(): Observable<Product[]> {
     return this.http.get<Product[]>('/data/products.json');
  }

  getById(productId: number): Observable<Product> {
     return this.http.get<Product[]>('/data/products.json')
  .pipe(
    map(products => <Product>products.find(p => p.id === productId));
   )
  }
}
```

Injects the HttpClient object

This function declares an Observable that can return all Product objects.

This function declares an Observable that can return products by ID.

map() finds the product ID that matches the function argument.

Because you don't have a real data server, both methods read the entire products.json file, and the `getById()` method also applies `find ()` to the array of products to find the one with a matching ID.

> **Best practice**
>
> You defined the type `Product` as an interface and not a class. Because JavaScript doesn't support interfaces, the compiled code won't include `Product`. If you were to define `Product` as a class, the TypeScript compiler would turn the `Product` class into either a JavaScript function or a class and would include it in the executable code. Defining types as TypeScript interfaces instead of classes reduces the size of the runnable code.

In the next section, you'll create the feature module that will include `HomeComponent`—the first consumer of the `ProductService`.

7.2.6 Creating the home module

You want to create each view as a feature module. This will allow you to lazy load them, and the code of each view will be built as a separate bundle. Generate a feature home module as follows:

```
ng generate module home
```

This command will create a src/app/home directory containing the home.module.ts file with the content shown in the following listing.

Listing 7.13 home.module.ts

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class HomeModule {}
```

You can generate the home component with the following command:

```
ng generate component home
```

After running this command, Angular CLI will print the message that four files were generated (the home component) and one file was updated (the home module)—the `HomeComponent` was added to the `declarations` section in the `@NgModule` decorator of the module:

```
create src/app/home/home.component.scss (0 bytes)
create src/app/home/home.component.html (23 bytes)
create src/app/home/home.component.spec.ts (614 bytes)
create src/app/home/home.component.ts (262 bytes)
update src/app/home/home.module.ts (251 bytes)
```

You'll use the Flex Layout library in this module, so you want to configure the default route so that it renders `HomeComponent`. Also, you're going to display products using the `<mat-grid-list>` component from the Angular Material library. Add the required code to home.module.ts so it looks like the following listing.

Listing 7.14 modified home.module.ts

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {RouterModule} from '@angular/router';
import {FlexLayoutModule} from '@angular/flex-layout';
import {MatGridListModule} from '@angular/material/grid-list';
import {HomeComponent} from './home.component';

@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([
       {path: '', component: HomeComponent}
    ]),
    FlexLayoutModule,
     MatGridListModule
   ],
  declarations: [HomeComponent]
})
export class HomeModule {}
```

Adds the route configuration for your feature module

Adds the Flex Layout library

Adds the Angular Material module required by <mat-grid-list>

The next step is to update the `HomeComponent` in the generated home.component.ts file. You'll inject two services into this component: `ProductService` and `ObservableMedia`. You'll invoke the `getAll()` method on `ProductService` to get product data. `ObservableMedia` will be watching the viewport width to change the UI layout accordingly. To be more specific, the product data will be shown in a grid, and the `ObservableMedia` service will change the number of columns in the grid from one to five, based on the current viewport width. The code of the `HomeComponent` is shown in the next listing.

Listing 7.15 home.component.ts

```
import {Observable} from 'rxjs';
import {map} from 'rxjs/operators';

import {Component} from '@angular/core';
import {ObservableMedia} from '@angular/flex-layout';
import {Product, ProductService} from '../shared/services';

@Component({
  selector: 'nga-home',
  styleUrls: [ './home.component.scss' ],
  templateUrl: './home.component.html'
})
export class HomeComponent {
  readonly columns$: Observable<number>;
   readonly products$: Observable<Product[]>;

  readonly breakpointsToColumnsNumber = new Map([
    [ 'xs', 1 ],
    [ 'sm', 2 ],
    [ 'md', 3 ],
    [ 'lg', 4 ],
    [ 'xl', 5 ],
  ]);

  constructor(private media: ObservableMedia,
              private productService: ProductService) {
    this.products$ = this.productService.getAll();

    this.columns$ = this.media.asObservable()
       .pipe(
        map(mc => <number>this.breakpointsToColumnsNumber.get(mc.mqAlias))
       );
  }
}
```

An observable to supply the number of columns in the grid

An observable of products

Maps the media query alias to the number of columns in the grid

Injects ObservableMedia and ProductService

Turns the ObservableMedia object into an Observable

Gets the number of the grid column based on the emitted media query alias; <number> means casting from object to number

Gets data about all products

The `getAll()` method on `ProductService` initializes the `product$` variable of type `Observable`. You don't see the invocation of the `subscribe()` method here, because you'll use the `async` pipe in the template of the home component.

The role of `ObservableMedia` is to send the media query alias to the component, indicating the current width of the user's device viewport. This width may be changing if the viewport is a window in the browser and the user resizes it. If the user runs this app on a smartphone, the width of the viewport won't change, but `HomeComponent` needs to know it anyway to render the grid of products.

Now you need to replace the generated template in home.component.html with the markup to display products in a grid of rows and columns. For the grid, you'll use the `<mat-grid-list>` component from the Angular Material library. The content of each grid cell will be rendered in a `<mat-grid-tile>` component.

In this template, you'll use the `async` pipe twice. The first `async` pipe will subscribe to the observable that emits the number of columns in a grid, and the second pipe will subscribe to the observable that emits product data. The code of the home.component .html file is shown in the following listing.

Listing 7.16 home.component.html

Subscribes to the number of columns and binds it to the cols property of <mat-grid-list>

Renders a <mat-grid-tile> for each product using the data from the products$ observable

Wraps the content of each tile in the <a> tag to turn the tile into a clickable link

Clicking on the tile will navigate to the path /products, passing the selected product's id as a parameter.

For extra-small viewports, adds the styles defined in tile__price-tag--xs

```
<div class="grid-list-container">
  <mat-grid-list [cols]="columns$ | async"
                 gutterSize="16">
    <mat-grid-tile class="tile" *ngFor="let product of products$ | async">
      <a class="tile__content"
         fxLayout="column"
        fxLayoutAlign="center center"
        [routerLink]="['/products', product.id]">

        <span class="tile__price-tag"
              ngClass.xs="tile__price-tag--xs">
          {{ product.price | currency:'USD':'symbol':'.0' }}
        </span>

        <div class="tile__thumbnail"
             [ngStyle]="{'background-
      image': 'url(' + product.imageUrl + ')'}"></div>

        <div class="tile__title"
             ngClass.xs="tile__title--xs"
             ngClass.sm="tile__title--sm">{{ product.title }}</div>
      </a>
    </mat-grid-tile>
  </mat-grid-list>
</div>
```

NOTE Navigation to the product-detail screen isn't implemented in this version of ngAuction. Clicking the product tile will result in an error in the browser console.

We'd like to explain the last annotation in listing 7.16 a bit more. That `<span>` element is styled as defined in `tile__price-tag`, but if the size of the viewport becomes extra small (`xs`), the Flex Layout `ngClass.xs` directive will add the styles defined in `tile__price-tag--xs`. If you compare the definitions of the `tile__price-tag` and `tile__price-tag--xs` styles in listing 7.17, you see that merging these two styles would mean changing the font size from 16 px to 14 px.

TIP We use the symbols __ and -- in naming some styles, as recommended by the block, element, modifier (BEM) methodology (see http://getbem.com).

To complete the `HomeComponent`, you need to add some styles in home.component .scss.

Listing 7.17 home.component.scss

```
@import '../../styles/theme';

:host {
  display: block;
}

.grid-list-container {
  margin: 16px;
}

.tile {
  background-color: mat-color($nga-background, card);

  &:hover {
    @include mat-elevation(4);
     transition: .3s;
  }
}

.tile__content {
  display: block;
  height: 100%;
  width: 100%;
  padding: 16px;
  position: relative;
  text-align: center;
  text-decoration: none;
}

.tile__price-tag {
   color: mat-color($nga-foreground, text);
  font-size: 16px;
  font-weight: 700;
  position: absolute;
  right: 20px;
  top: 20px;
}

.tile__price-tag--xs {
   font-size: 14px;
```

Imports your customized theme

Makes the tile background color the same as the Angular Material card (white)

If the user hovers over the tile, elevates the tile to level 4 by adding the shadow effect (returned by the mat-elevation mixin)

Default style for the product price tag

Style for the product price tag for extra-small viewports

```
}

.tile__thumbnail {
  background: no-repeat 50% 50%;
  background-size: contain;
  height: 50%;
  width: 50%;
}

.tile__title {
   color: mat-color($nga-foreground, text);
  font-family: 'Abril Fatface', cursive;
  font-size: mat-font-size($nga-typography, display-1);
   line-height: mat-line-height($nga-typography, display-1);
}

.tile__title--
     sm {
   font-size: mat-font-size($nga-typography, headline);
  line-height: mat-line-height($nga-typography, headline);
}

.tile__title--xs {
   font-size: mat-font-size($nga-typography, title);
  line-height: mat-line-height($nga-typography, title);
}
```

Default style for the product title

As per Material Design spec, uses Display 1 for font styles instead of specifying the hardcoded size

Style for the product title for small viewports

Style for the product title for extra-small viewports

The `HomeComponent` is ready. What do you need to do to render it under the toolbar?

7.2.7 Configuring routes

In the beginning of this hands-on exercise, we stated that each view on ngAuction will be a separate module, and you created the `HomeComponent` as a module. Now you need to configure the route for this module. Create an src/app/app.routing.ts file with the following content:

```
import {Route} from '@angular/router';

export const routes: Route[] = [
  {
    path: '',
    loadChildren: './home/home.module#HomeModule'
  }
];
```

As you see, you use the syntax for lazy-loaded modules, as explained in section 4.3 in chapter 4. You to load this configuration in app.module.ts by invoking `Router.forRoot()`:

```
...
import {RouterModule} from '@angular/router';
import {routes} from './app.routing';

@NgModule({
  ...
  imports: [
```

```
    ...
    RouterModule.forRoot(routes)
  ]
  ...
})
export class AppModule { }
```

The last step is to uncomment the last line in app.component.html that has the `<router-outlet>` tag, so the app component template is laid out as follows:

```
<mat-toolbar>...</mat-toolbar>

<router-outlet></router-outlet>
```

The coding part of the landing page is done.

7.2.8 Running ngAuction

The first version of the new ngAuction is ready, so let's build the dev bundles and see how it looks in the browser. Running `ng serve` produces the output shown in figure 7.10.

```
chunk {home.module} home.module.chunk.js, home.module.chunk.js.map () 46.7 kB {main}  [rendered]
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB [entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 15 kB {vendor} [initial] [rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 217 kB {inline} [initial] [rendered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 68.4 kB {inline} [initial] [rendered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 3.37 MB [initial] [rendered]
```

Figure 7.10 Bundling ngAuction with `ng serve`

Note the first line: Angular CLI placed the home module in a separate bundle. It did that because in configuring routes, you used the syntax for lazy-loaded modules, but when you open the browser at http://localhost:4200, you'll see that the home module was loaded, as shown in figure 7.11.

The home module was eagerly loaded because it was configured as a default route (mapped to an empty path). The landing page of ngAuction is ready, except it doesn't have the Search button on the toolbar. You'll add it in section 11.8 in chapter 11.

> **TIP** If you click on any of the product tiles, the browser console shows an error, as in "Cannot match any routes. URL Segment: 'products/2'." This error will disappear in chapter 9's version of ngAuction, after you develop the product-detail page.

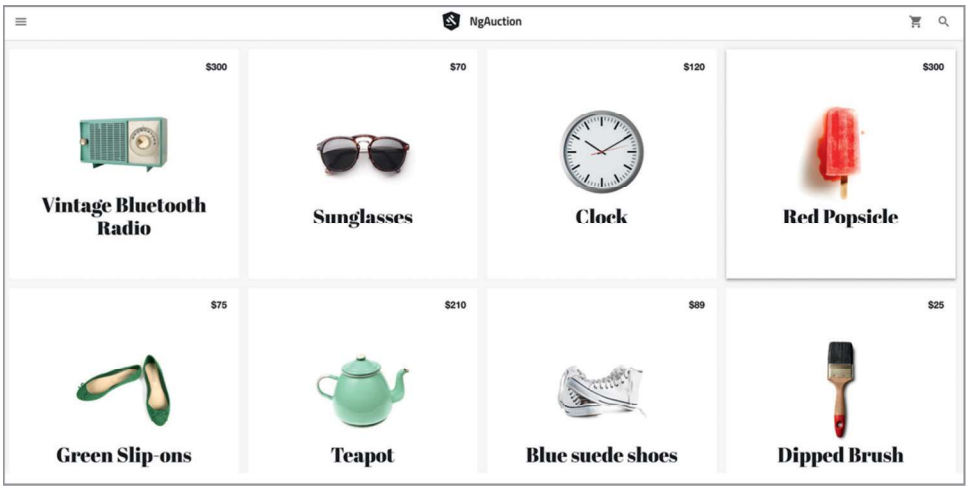


Figure 7.11 Running ngAuction

Summary

- You can keep a single code base of the web app that will adopt its UI based on the available width of the user device.
- The Flex Layout library allows you to subscribe to notifications about viewport width changes and apply the respective UI layout.
- The Flex Layout library includes the `ObservableMedia` class, which can notify you about the current width of the viewport, sparing you from writing CSS for this purpose.

8 Implementing component communications

This chapter covers

- Creating loosely coupled components
- How a parent component should pass data to its child, and vice versa
- Implementing the Mediator design pattern for component communication

An Angular application is a tree of views represented by components. While designing components, you need to ensure that they're self-contained and at the same time have some means of communicating with each other. In this chapter, we'll focus on how components can pass data to each other in a loosely coupled manner.

First, we'll show you how a parent component can pass data to its children by binding to their input properties. Then, you'll see how a child component can send data to its parent by emitting events via its output properties.

We'll continue with an example that applies the Mediator design pattern to arrange data exchange between components that don't have parent-child relationships. Mediator is probably the most important design pattern in any component-based framework.

8.1 Intercomponent communication

Figure 8.1 shows a view that consists of components that are numbered and have different shapes for easier reference. Some of the components contain other components (let's call the outer ones *containers*), and others are peers. To abstract this from any particular UI framework, we've avoided using HTML elements like input fields, drop-downs, and buttons, but you can extrapolate this into a view of your real-world application.

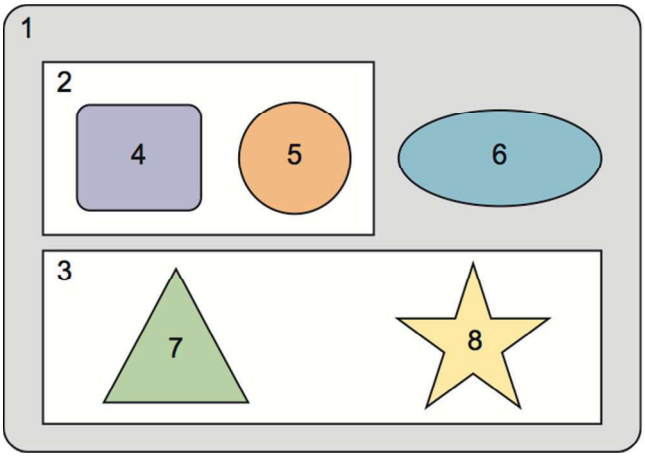


Figure 8.1 A view consists of components.

When you design a view that consists of multiple components, the less they know about each other, the better. Say a user clicks the button in component 4, which has to initiate some actions in component 5. Is it possible to implement this scenario without component 4 knowing that component 5 exists? Yes, it is.

You've seen already examples of loosely coupling components by using dependency injection. Now we'll show you a different technique for achieving the same goal by using bindings and events.

8.2 Input and output properties

Think of an Angular component as a black box with outlets. Some of them are marked as `@Input()`,and others are marked as `@Output()`. You can create a component with as many inputs and outputs as you want.

If an Angular component needs to receive values from the outside world, you can bind the producers of these values to the corresponding inputs of the component. Who are they received from? The component doesn't have to know. The component just needs to know what to do with these values when they're provided.

If a component needs to communicate values to the outside world, it can *emit events* through its output properties. Whom are they emitted to? The component doesn't have to know. Whoever is interested can subscribe to the events that a component emits.

Let's implement these loosely coupled principles. First, you'll create an `OrderProcessorComponent` that can receive order requests from its parent component.

8.2.1 Input properties

The input properties of a component annotated with the `@Input()` decorator are used to get data from the parent component. Imagine that you want to create a UI component for placing orders to buy stocks. It will know how to connect to the stock exchange, but that's irrelevant in the context of this discussion of input properties. You want to ensure that `OrderProcessorComponent` receives data from other components via its properties marked with `@Input()` decorators. Your `OrderProcessorComponent` will look like the following listing.

Listing 8.1 order.component.ts

```
@Component({
    selector: 'order-processor',
    template: `
    <span *ngIf="!!stockSymbol">
        Buying {{quantity}} shares of {{stockSymbol}}
    </span>
  `,
    styles:[`:host {background: cyan;}`]
})
export class OrderProcessorComponent {
    @Input() stockSymbol: string;
     @Input() quantity: number;
 }
```

Doesn't show the text unless the stockSymbol is truthy

Declares the input property to receive the stock symbol

Declares the input property to receive the quantity

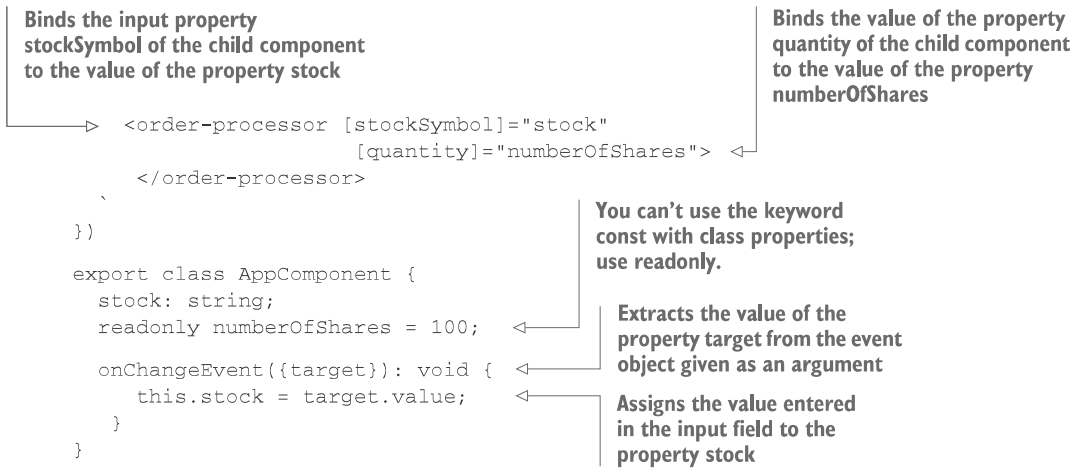
The `OrderProcessorComponent` doesn't know who will provide the values for these properties, which makes this component completely reusable.

Next, we'll look at the `AppComponent`, which in your app is the parent of `OrderComponent`. `AppComponent` allows users to enter a stock symbol in the input field, and the entered value is passed to the `OrderProcessorComponent` via property binding. The following listing shows the code of the `AppComponent`.

Listing 8.2 input/app.component.ts

```
@Component({
  selector: 'app-root',
  template: `
    <input type="text" placeholder="Enter stock (e.g. AAPL)"
           (change)="onChangeEvent($event)">
```

When the user moves the focus from the input field (change event), invokes the event handler passing the event object to it



Both properties of the `<order-processor>` component are surrounded with square brackets to denote property binding. If you change the value of `stockSymbol` or `quantity` inside the `OrderProcessorComponent`, the change won't affect the property values of the parent component. Property binding is unidirectional: from parent to child.

To see this app in action, run `npm install` in the chapter8/inter-component folder, and run the following command:

```
ng serve --app input -o
```

Best practice

Though we praised TypeScript for allowing specification of variable types, we didn't declare the type for the `numberOfShares` property. Because we initialized it with a numeric value, TypeScript compiler will use *type inference* to guess the type of `NumberOfShares` when it gets initialized. Explicitly declare types in a public API, for example, public class properties, function parameters and return types, and so on.

Figure 8.2 shows the browser window after the user types `IBM` in the input field. The `OrderProcessorComponent` received the input values `100` and `IBM`.

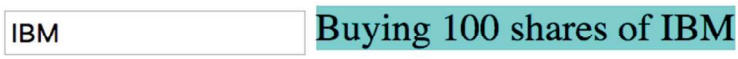


Figure 8.2 The `OrderProcessorComponent` receives values.

How can a component intercept the moment when the value of the input property `stockSymbol` changes to perform some additional processing? A simple way is to turn `stockSymbol` into a setter. If you want to use `stockSymbol` in the template of the component, create a getter as well, as shown in the following listing.

Listing 8.3 Adding the setter and getter

```
...
private _stockSymbol: string;

@Input() set stockSymbol(value: string)
  if (value !== undefined) {
    this._stockSymbol = value;
    console.log(`Buying ${this.quantity} shares of ${value}`);
  }
}

get stockSymbol(): string {
   return this._stockSymbol;
}
```

This private variable isn't accessible from the template.

Defines an input property as a setter

Defines a getter so stockSymbol is accessible from the template

When this application starts, the change detection mechanism qualifies the initialization as a change of the bound variable `stockSymbol`. The setter is invoked, and, to avoid sending an order for the undefined `stockSymbol`, you check its value in the setter.

> **NOTE** In section 9.2.1 in chapter 9, we'll show you how to intercept the changes in input properties without using setters.

8.2.2 Output properties and custom events

Angular components can dispatch custom events using the `EventEmitter` object. These events are to be consumed by the component's parent. `EventEmitter` is a subclass of `Subject` (explained in appendix D) that can serve as both observable and observer, but typically you use `EventEmitter` just for emitting custom events that are handled in the template of the parent component.

> **Best practice**
> If you need to have an object that's both an observable and an observer, use the RxJS `BehaviorSubject`. You'll see how to do that in section 8.3.2. In future releases, the internal implementation of `EventEmitter` may change, so it's better to use it only for emitting custom events.

Let's say you need to write a UI component that's connected to a stock exchange and displays changing stock prices. In addition to displaying prices, the component should also send events with the latest prices so its parent component can handle it and apply business logic to the changing prices. Let's create a `PriceQuoterComponent` that implements

such functionality. In this component, you won't connect to any financial servers but will rather emulate the changing prices using a random number generator.

Displaying changing prices inside `PriceQuoterComponent` is pretty straightforward—you'll bind the `stockSymbol` and `lastPrice` properties to the component's template.

You'll notify the parent about the latest prices by emitting custom events via the `@Output` property of the component. Not only will you fire an event as soon as the price changes, but this event will also carry a payload: an object with the stock symbol and its latest price. The type of the payload will be defined as `PriceQuote` interface, as shown in the following listing.

Listing 8.4 iprice.quote.ts

```
export interface PriceQuote {
  stockSymbol: string;
  lastPrice: number;
}
```

The `PriceQuoterComponent` will generate random quotes and will emit them every two seconds.

Listing 8.5 price.quoter.component.ts

```
@Component({
  selector: 'price-quoter',
  template: `<strong>Inside PriceQuoterComponent:
                  {{priceQuote?.stockSymbol}}
                   {{priceQuote?.lastPrice | currency: 'USD'}}</strong>`,
  styles: [`:host {background: pink;}`]
})
export class PriceQuoterComponent {
  @Output() lastPrice = new EventEmitter<PriceQuote>();

  priceQuote : PriceQuote;

  constructor() {
    Observable.interval(2000)
        .subscribe(data =>{
         this.priceQuote = {
           stockSymbol: "IBM",
           lastPrice: 100 * Math.random()
         };

         this.lastPrice.emit(this.priceQuote);}
        )
  }
}
```

The question mark represents the safe navigation operator.

The output property lastPrice is represented by the EventEmitter object, which emits lastPrice events to the parent.

Emulates changing prices by invoking a function that generates a random number every two seconds and populates the priceQuote object

Emits new price via the output property; the lastPrice event carries the PriceQuote object as a payload

The safe navigation operator in `priceQuote?` ensures that if the `priceQuote` object isn't available yet, the code in the template won't try to access properties of an uninitialized `priceQuote`.

TIP We used the `Observable.interval()` instead of `setInterval()` because the latter is the browser-only API. Starting from Angular 6, use `interval()` instead of `Observable.interval()`.

The next listing shows how the parent component will receive and handle the `lastPrice` from the `<price-quoter>` component.

Listing 8.6 app.component.ts

```
@Component({
  selector: 'app-root',
  template: `
    AppComponent received: {{priceQuote?.stockSymbol}}
                           {{priceQuote?.lastPrice | currency:'USD'}}
   <price-quoter (lastPrice)="priceQuoteHandler($event)">
➥</price-quoter>
    `
})
export class AppComponent {
  priceQuote : IPriceQuote;

  priceQuoteHandler(event: IPriceQuote) {
     this.priceQuote = event;
  }
}
```

The AppComponent receives the lastPrice event and invokes the priceQuoteHandler, passing the received object as an argument.

Receives the IPriceQuote object and uses its properties to populate the respective properties of the AppComponent

Run this example, and you'll see the prices update every two seconds in both `PriceQuoterComponent` (on a pink background) as well as in `AppComponent` (white background), as shown in figure 8.3.

AppComponent received: IBM $21.00 **Inside PriceQuoterComponent: IBM $21.00**

Figure 8.3 Running the output properties example

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

```
ng serve --app output -o
```

Event bubbling

Angular doesn't offer an API to support event bubbling. If you try to listen to the `lastPrice` event not on the `<price-quoter>` element but on its parent, the event won't bubble up there. In the following code snippet, the `lastPrice` event won't reach the `<div>`, because it's the parent of `<price-quoter>`:

```
<div (lastPrice)="priceQuoteHandler($event)">
  <price-quoter></price-quoter>
</div>
```

If event bubbling is important to your app, don't use `EventEmitter`; use native DOM events instead. The following code snippet shows how the `PriceQuoterComponent` uses a `CustomEvent` (from Web API) that supports bubbling:

```
@Component(...)
class PriceQuoterComponent {
  stockSymbol = "IBM";
  price;

  constructor(element: ElementRef) {
    setInterval(() => {
       let priceQuote: IPriceQuote = {
        stockSymbol: this.stockSymbol,
        lastPrice: 100 * Math.random()
      };

      this.price = priceQuote.lastPrice;

      element.nativeElement
          .dispatchEvent(new CustomEvent('lastPrice', {
            detail: priceQuote,
            bubbles: true
          }));
    }, 1000);
  }
}
```

Angular injects an `ElementRef` object, which has a reference to the DOM element that represents `<price-quoter>`, and then a `CustomEvent` is dispatched by invoking `element.nativeElement.dispatchEvent()`. Event bubbling will work here, but using `ElementRef` works only in browser-based apps and won't work with non-HTML renderers.

The `AppComponent` shown next handles the `lastPrice` event in the `<div>`, which is a parent of the `<price-quoter>` component. Note that the type of the argument of the `priceQuoteHandler()` is `CustomEvent`, and you can access its payload via the `detail` property:

```
@Component({
  selector: 'app',
  template: `
    <div (lastPrice)="priceQuoteHandler($event)">
      <price-quoter></price-quoter>
    </div>
    <br>
    AppComponent received: {{stockSymbol}}
                           {{price | currency: 'USD'}}
  `
})
class AppComponent {

  stockSymbol: string;
  price: number;
```

(continued)

```
priceQuoteHandler(event: CustomEvent) {
  this.stockSymbol = event.detail.stockSymbol;
  this.price = event.detail.lastPrice;
}}
```

We established that each UI component should be self-contained and shouldn't rely on the existence of other UI components, and using `@Input()` and `@Output()` decorators allows you to create reusable components. But how do you arrange communication between two components if they don't know about each other?

8.3 Implementing the Mediator design pattern

Communication between loosely coupled components can be implemented using the Mediator design pattern, which, according to Wikipedia, "defines how a set of objects interact" (https://en.wikipedia.org/wiki/Mediator_pattern). We'll explain what this means by analogy with interconnecting toy bricks.

Imagine a child playing with building bricks (think *components*) that "don't know" about each other. Today this child (the *mediator*) can use some blocks to build a house, and tomorrow they'll construct a boat from the same components.

> **NOTE** The role of the mediator is to ensure that components properly fit together according to the task at hand while remaining loosely coupled.

Coming back to the web UI realm, we'll consider two cases:

- Arranging communication when components have a common parent
- Arranging communication when components don't have a common parent

8.3.1 Using a common parent as a mediator

Let's revisit the first figure of this chapter, shown again in figure 8.4. Each component except 1 has a parent (a container) that can play the role of mediator. The top-level mediator is container 1, which is responsible for making sure its direct children 2, 3, and 6 can communicate if need be. On the other hand, component 2 is a mediator for 4 and 5. Component 3 is a mediator for 7 and 8.

The mediator needs to receive data from one component and pass it to another. Let's go back to examples of monitoring stock prices.

Imagine a trader monitoring the prices of several stocks. At some point, the trader clicks the Buy button next to a stock symbol to place a purchase order with the stock exchange. You can easily add a Buy button to the `PriceQuoterComponent` from the previous section, but this component doesn't know how to place orders to buy stocks. `PriceQuoterComponent` will notify the mediator (`AppComponent`) that the trader wants to purchase a particular stock at that moment.

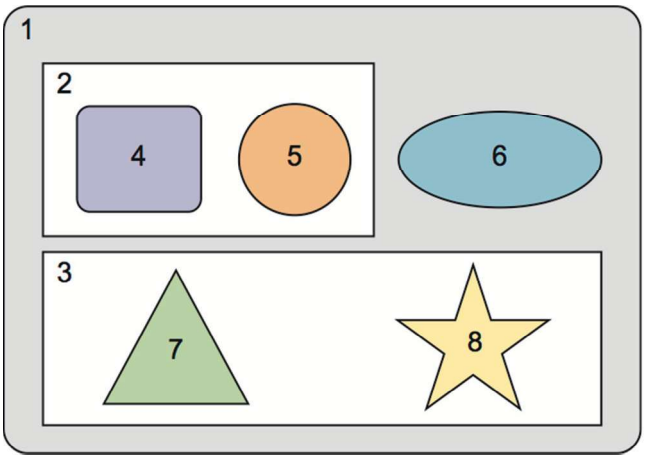


Figure 8.4 A view consists of components.

The mediator should know which component can place purchase orders and how to pass the stock symbol and quantity to it. Figure 8.5 shows how an `AppComponent` can mediate the communication between `PriceQuoterComponent` and `OrderComponent`.

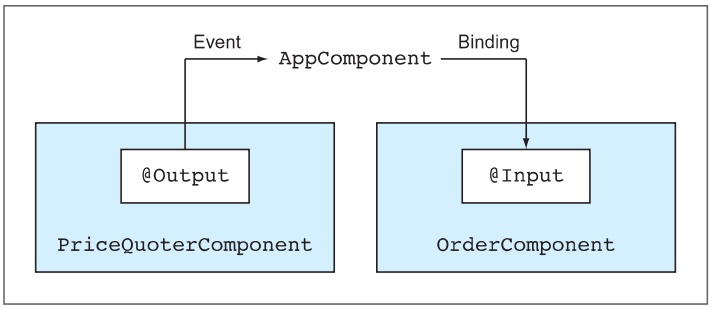


Figure 8.5 Mediating communications

NOTE Emitting events works like broadcasting. `PriceQuoterComponent` emits events via the `@Output()` property without knowing who will receive them. `OrderComponent` waits for the value of its `@Input()` property to change as a signal for placing an order.

To demonstrate the Mediator pattern in action, let's write a small app that consists of the two components shown in figure 8.5. You can find this application in the mediator-parent directory, which has the following files:

- *istock.ts*—The `Stock` interface defining a value object that represents a stock
- *price.quoter.component.ts*—`PriceQuoterComponent`
- *order.component.ts*—`OrderComponent`

- *app.component.ts*—A parent component (the mediator) that contains `<price-quoter>` and `<order-processor>` in its template
- *app.module.ts*—The `AppModule` class

You'll use the `Stock` interface in two scenarios:

- To represent the payload of the event emitted by the `PriceQuoterComponent`
- To represent the data given to the `OrderComponent` via binding

The content of the istock.ts file is shown in the following listing.

Listing 8.7 istock.ts

```
export interface Stock {
  stockSymbol: string;
  bidPrice: number;
}
```

The `PriceQuoterComponent`, shown in the next listing, has a Buy button and the `buy` output property. It emits the `buy` event only when the user clicks the Buy button.

Listing 8.8 price.quoter.component.ts

```
@Component({
  selector: 'price-quoter',
  template: `<strong>
              <button (click)="buyStocks()">Buy</button>
              {{stockSymbol}} {{lastPrice | currency: "USD"}}
            </strong>
           `,
  styles:[`:host {background: pink; padding: 5px 15px 15px 15px;}`]
})
export class PriceQuoterComponent {
  @Output() buy: EventEmitter<Stock> = new EventEmitter();

  stockSymbol = "IBM";
  lastPrice: number;

  constructor() {
    Observable.interval(2000)
      .subscribe(data =>
      this.lastPrice = 100 * Math.random());
  }

  buyStocks(): void {

    let stockToBuy: Stock = {
      stockSymbol: this.stockSymbol,
      bidPrice: this.lastPrice
    };

    this.buy.emit(stockToBuy);
   }
}
```

The buy output property will be used as a custom buy event.

Emits the custom buy event

When the mediator (`AppComponent`) receives the `buy` event from `<price-quoter>`, it extracts the payload from this event and assigns it to the `stock` variable, which is bound to the input parameter of `<order-processor>`, as shown in the following listing.

Listing 8.9 app.component.ts

```
@Component({
  selector: 'app-root',
  template: `
    <price-
     quoter (buy) = "priceQuoteHandler($event)">
     </price-quoter>

    <order-processor
           [stock] = "receivedStock">
     </order-processor>
  `
})
export class AppComponent {
  receivedStock: Stock;

  priceQuoteHandler(event: Stock) {
    this.receivedStock = event;
  }
}
```

When the mediator receives the buy event, it invokes the event handler.

The stock received from <price-quoter> is passed to <order-processor>.

When the value of the `buy` input property on `OrderComponent` changes, its setter displays the message "Placed order ...," showing the `stockSymbol` and the `bidPrice`.

Listing 8.10 order.component.ts

```
@Component({
  selector: 'order-processor',
  template: `{{message}}`,
  styles:[`:host {background: cyan;}`]
})
export class OrderComponent {

  message = "Waiting for orders...";

  @Input() set stock(value: Stock) {
     if (value && value.bidPrice != undefined) {
      this.message = `Placed order to buy 100 shares
                        of ${value.stockSymbol} at
                      \$${value.bidPrice.toFixed(2)}`;
    }
  }
}
```

Receives the stock object through this setter

Prepares a message to be displayed in the template

Figure 8.6 shows what happens after the user clicked the Buy button when the price of the IBM stock was $36.53. `PriceQuoterComponent` is rendered on the left, and `OrderComponent` is on the right. They're self-contained, loosely coupled, and still can communicate with each other via the `AppComponent` mediator.

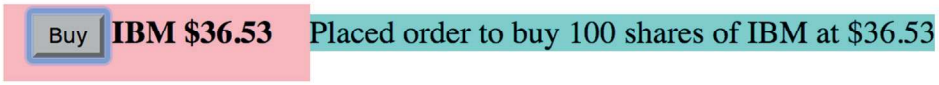


Figure 8.6 Running the mediator example

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

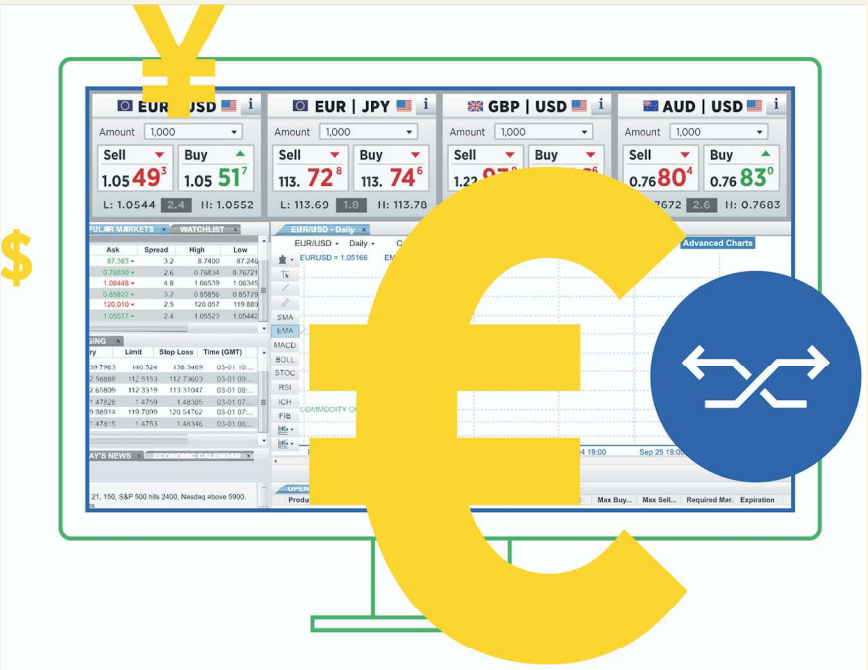
```
ng serve --app mediator-parent -o
```

The Mediator design pattern is a good fit for ngAuction as well. Imagine the last minutes of a bidding war for a hot product. Users monitor frequently updated bids and click a button to increase their bids.

A real-world example of a multicomponent UI

You can find many UIs that consist of multiple components in real-world web apps. We'll show you a UI taken from the publicly available site www.forex.com that offers a web platform for trading currencies. A trader can monitor the prices of multiple currency pairs (for example, US dollars and euros) in real time and place orders to buy the currencies when the price is right.

Here's a snapshot of a trader's screen that you can find at http://mng.bz/M9Af.



A sample trader's screen from forex.com

We don't know which JavaScript framework (if any) was used for creating this UI, but we can clearly see that it consists of multiple components. If we needed to develop such an app in Angular, we'd create, say, a `CurrencyPairComponent` and would place four of its instances at the top. Below, we'd use other components, such as `PopularMarketComponent`, `WatchListComponent`, and so on.

Within the `CurrencyPairComponent`, we'd create two child components: `SellComponent` and `BuyComponent`. Their Sell and Buy buttons would emit a custom event that would be received by the parent `CurrencyPairComponent`, which in turn would need to communicate with an `OrderComponent` to place a sell or buy order. But what if `CurrencyPairComponent` and `OrderComponent` don't have a common parent? Who will mediate their communications?

8.3.2 Using an injectable service as a mediator

In the last section, you saw how sibling components use their parent as a mediator. If components don't have the same parent or aren't displayed at the same time (the router may not display the required component at the moment), you can use an injectable service as a mediator. Whenever the component is created, the mediator service is injected, and the component can subscribe to events emitted by the service (as opposed to using `@Input()` parameters like `OrderComponent` did).

Figure 8.7 shows a diagram representing a scenario when component 5 needs to send data to components 6 and 8. As you see, they don't have a common parent, so you use an injectable service as a mediator.

The same instance of the service will be injected into components 5, 6, and 8. Component 5 can use the API of the service to provide some data, and components 6

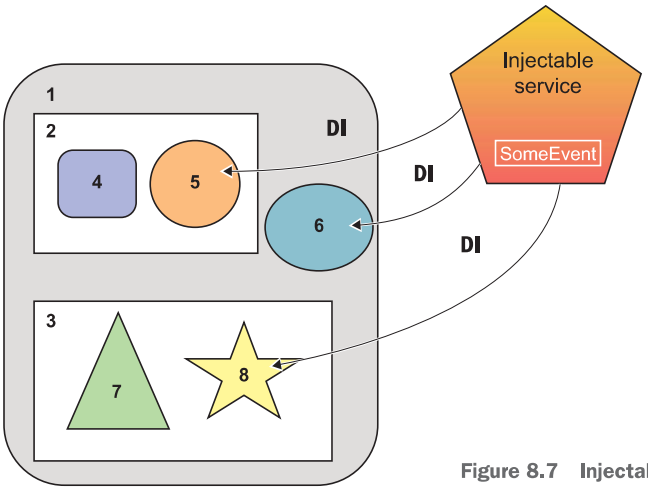


Figure 8.7 Injectable service as mediator

and 8 will subscribe to the data as soon as they're instantiated. By creating a subscription in the constructors of components 6 and 8, you ensure that no matter when these components are created, they'll start getting data from the service right away.

Let's consider a practical example to illustrate how this works. Imagine you have a UI with which you can search for products by typing a product name in an input box of a component. You want to offer searching for products either on eBay or Amazon. Initially, you'll render the eBay component, but if users aren't satisfied with the deal offered on eBay, they'll try to find the same product on Amazon. Figure 8.8 shows the UI of this app after the user enters `aaa` as a product name in the search field. Initially, the eBay component is rendered and receives aaa as a product to search for.

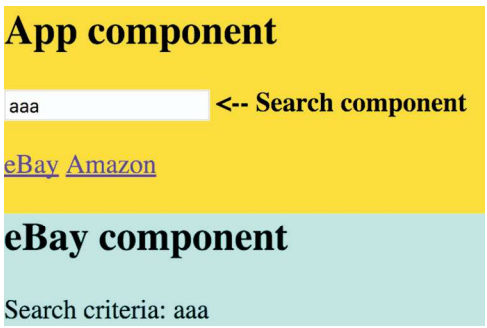


Figure 8.8 Searching for product aaa on eBay

Say eBay provides details and pricing for aaa, but the user isn't happy and clicks the link to find the same product on Amazon. Your UI has two links, one for eBay and another for Amazon. When the user clicks the Amazon link, the router destroys the eBay component and creates the Amazon one. You want to maintain the application state so the user doesn't need to reenter the product name, and the Amazon component has to be rendered showing aaa—the saved search criteria, as shown in figure 8.9.

If the user changes their mind and decides to search for a different product on Amazon, and then returns back to eBay, the new search criteria has to be shown in the eBay component.

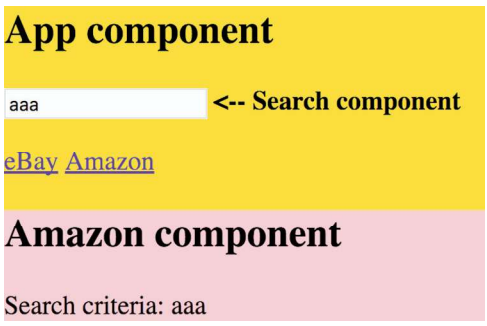


Figure 8.9 Searching for product aaa on Amazon

So you need to implement two features:

- Communication between the search, eBay, and Amazon components.
- State management so the latest search criteria is preserved while the user navigates between eBay and Amazon.

The code for this app is located in the mediator-service-subject folder and contains the following files:

- *app.component.ts*—The top-level component `AppComponent`
- *app.module.ts*—The `AppModule` that includes routes configuration
- *state.service.ts*—The injectable service that also stores the app state
- *search.component.ts*—The `SearchComponent` with an `<input>` field
- *amazon.component.ts*—The `AmazonComponent`
- *ebay.component.ts*—The `EbayComponent`

`AppComponent` serves as a parent for `SearchComponent`, provides two links to eBay and Amazon components, and includes `<router-outlet>`, as shown in the following listing.

Listing 8.11 app.component.ts

```
@Component({
  selector: 'app-root',
  template: ` <div class="main">
               <h2>App component</h2>

               <search></search>  <b><-- Search component</b>
                <p>

               <a [routerLink]="['/']">eBay</a>
                <a [routerLink]="['/amazon']">Amazon</a>
                <router-outlet></router-outlet>
               </div>`,
  styles: ['.main {background: yellow}']
})

export class AppComponent {}
```

The user enters the product name here.

A link used for navigation

The eBay or Amazon component is rendered under the links.

`AppModule` loads the routes configuration as follows:

```
RouterModule.forRoot([
    {path: '',        component: EbayComponent},
    {path: 'amazon', component: AmazonComponent}])
]
```

You want to create an injectable `StateService` that would accept the search criteria from the `SearchComponent` and emit it to its subscribers (eBay or Amazon components). In appendix D, we explain how the RxJS `Subject` works. It contains both observable and observer and would fit your needs except it wouldn't remember the emitted value (the search criteria). You could create a separate variable to store the value provided by the `SearchComponent`, but there's a better solution.

The RxJS library includes `BehaviorSubject`, which supports the functionality of `Subject`—plus it reemits the latest emitted value. Let's see how it'll work in your app:

1. The user enters `aaa`, and the `SearchComponent` invokes the API on the `StateService` to emit `aaa` to the subscriber, which is initially an eBay component. The `BehaviorSubject` emits `aaa` and remembers it (stores the app state).
2. The user navigates to the Amazon component, which immediately subscribes to the same `BehaviorSubject`, which reemits `aaa`.

The code of the `StateService` is shown in the next listing.

Listing 8.12 state.service.ts

```
@Injectable()
export class StateService {

  private stateSubject: BehaviorSubject<string> = new BehaviorSubject('');

  set searchCriteria(value: string) {

    this.stateSubject.next(value);
   }

  getState(): Observable<string> {
     return this.stateSubject.asObservable();
  }
}
```

Creates an instance of BehaviorSubject to reemit the last emitted value to new subscribers

SearchComponent will invoke this method.

Emits the search criteria to subscriber(s)

Returns the reference to Observable of the Subject

The `getState()` method returns the observable portion of `BehaviorSubject` so the eBay or Amazon components can subscribe to it. Technically, these components could subscribe to the subject directly, but if they had a reference to your `BehaviorSubject`, they could use the `next()` API to emit data on the subject's observers. You want to allow eBay or Amazon components to only use the `subscribe()` API—that's why you'll give them only the reference to the observable property from the `BehaviorSubject`.

> **NOTE** We used the `Injectable()` decorator, but it's optional here because we don't inject other services into `StateService`. If we injected into this service the `HttpClient` or any other service, using `Injectable()` would be required.

The code of the `SearchComponent` is shown next. You use the Forms API to subscribe to the `valueChanges` observable, as explained in chapter 6. Note that you inject the `StateService` into this component, and as the user types in the input field, you assign the values to the `searchCriteria` property on the `StateService`. The `searchCriteria` property is implemented as a setter, which emits the values entered by the user to the subscriber(s) of the `stateSubject`, as shown in the following listing.

Listing 8.13 search.component.ts

```
@Component({
  selector: "search",
  template: `
      <input type="text" placeholder="Enter product"
             [formControl]="searchInput">
    `
})
export class SearchComponent {

  searchInput: FormControl;

  constructor(private state: StateService) {

    this.searchInput = new FormControl('');

    this.searchInput.valueChanges
       .pipe(debounceTime(300))
      .subscribe(searchValue =>
                    this.state.searchCriteria = searchValue);
   }
}
```

An observable that emits the content of the input field

Passes the entered value to StateService

The following listing shows the code of `EbayComponent`, which gets `StateService` injected and subscribes to the observable of `stateSubject`.

Listing 8.14 ebay.component.ts

```
@Component({
  selector: 'product',
  template: `<div class="ebay">
               <h2>eBay component</h2>
              Search criteria: {{searchFor$ | async}}
               </div>`,
  styles: ['.ebay {background: cyan}']
})
export class EbayComponent {

  searchFor$: Observable<string>;

  constructor(private state: StateService){

      this.searchFor$ = state.getState();
   }
}
```

The async pipe autosubscribes to the observable searchFor$.

Injects the StateService

Stores the observable in the class variable

NOTE The code in the `AmazonComponent` should be identical, but in the source code that comes with this chapter, we keep a more verbose version that uses `subscribe()` and `unsubscribe` so you can compare and appreciate the benefits of the `async` pipe.

When the eBay (or Amazon) component is created, it gets the existing state of the `stateSubject` and displays it. Figure 8.10 shows how the components of the sample app communicate.

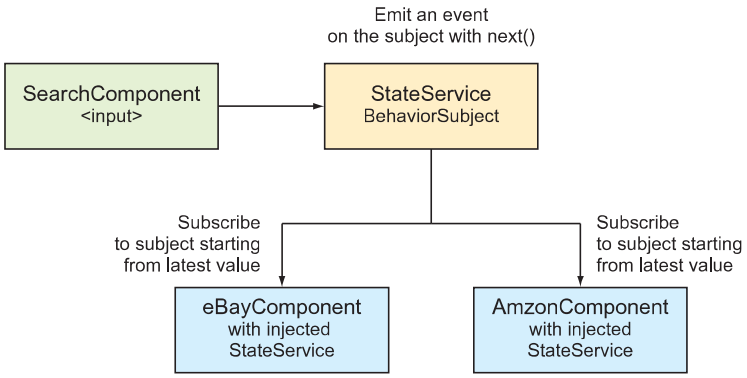


Figure 8.10 The application's workflows

> **NOTE** This sample app not only illustrates how you can arrange intercomponent communication using an injectable service as a mediator, it also shows how you can keep track of the app state in small and mid-size applications. If your application is large, consider implementing app state using the NgRx library, as explained in chapter 15.

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

```
ng serve --app mediator-service -o
```

You can also watch a short video that explains how this app works at http://mng.bz/oE0s.

> **TIP** Don't start implementing the UI components of your application until you've identified your mediators, the reusable components, and the means of communication between them.

Now you know that a parent component can pass data to other components without knowing either their content or existence. But what if a parent component knows that it has a child that implements a certain API; can the parent invoke this API on the child directly?

8.4 Exposing a child component's API

You've learned how a parent component can pass data to its child using bindings to input properties. But there are other cases when the parent just needs to use the API exposed by the child. We'll show you an example that illustrates how a parent

component can invoke the child's API from both the template and the TypeScript code of the parent.

Let's create a simple application in which a child component has a `greet()` method that will be invoked by the parent. In particular, the parent component includes the following lines in its template:

```
<child name= "John" #child1></child>
<button (click) = "child1.greet()">Invoke greet() on child 1</button>
```

Local template variables are meant to be used within the template. In the preceding code, the parent's template invokes the `greet()` method on the child component, `#child1`.

You can also invoke the child's API from TypeScript. Let's create two instances of the same child component to illustrate how to do this:

```
<child name= "John" #child1></child>
<child name= "Mary" #child2></child>
```

The DOM references to these instances will be stored in template variables `#child1` and `#child2`, respectively. Now you can declare a property in your TypeScript class, decorated with `@ViewChild()` so you can use these objects from your TypeScript code. The `@ViewChild()` decorator is handy when you need a reference to a child component.

Here's how you can pass a reference to the child component from the template variable `#child1` to the TypeScript variable `firstChild`:

```
@ViewChild('child1')
firstChild: ChildComponent;
...
this.firstChild.greet();
```

The `@ViewChildren()` decorator would give you references to several children of the same type. Let's write a small app that will illustrate the use of these decorators. The code of the child component is located in the childapi/child.component.ts file and is shown in the following listing.

Listing 8.15 child.component.ts

```
@Component({
  selector: 'child',
  template: `<h3>Child {{name}}</h3>`

})
export class ChildComponent {

  @Input() name: string;

  greet() {
    console.log(`Hello from ${this.name}`);
  }
}
```

The parent will include two instances of the child and will use both `@ViewChild()` and `@ViewChildren()` decorators. The full code of the parent component that uses both decorators is shown in the following listing.

Listing 8.16 app.component.ts

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Parent</h1>
    <child name = "John" #child1></child>
    <child name = "Mary" #child2></child>

    <button (click) = "child2.greet()">
            Invoke greet() on child 2
    </button>
    <button (click) = "greetAllChildren()">
            Invoke greet() on both children
    </button>
  `
})
export class AppComponent implements AfterViewInit {
  @ViewChild('child1')
  firstChild: ChildComponent;

  @ViewChildren(ChildComponent)
  allChildren: QueryList<ChildComponent>;

  ngAfterViewInit() {
     this.firstChild.greet();
   }

  greetAllChildren() {
     this.allChildren.forEach(child => child.greet());
   }
}
```

Obtains the reference to the first child instance

Obtains the references to both children (returns a list of children)

Invokes the greet() method on the first child

Invokes the greet() method on both children

Uses the lifecycle hook ngAfterViewInit()

> **NOTE** In this class, you use the component lifecycle hook `ngAfterViewInit()` to ensure that you use the child's API after the child is rendered. See section 9.2 in chapter 9 for more details.

If you run this app, the browser renders the window shown in figure 8.11.

You'll also see following line on the browser console:

```
Hello from John
```

Figure 8.11 Accessing the children API

On app startup, John is greeted, but to be fair, both children should be greeted. Clicking the button will use the reference to the entire list of children and produce the following output:

```
Hello from John
Hello from Mary
```

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

```
ng serve --app childapi -o
```

You used different techniques for component communications to send data or invoke the API, but can you send an HTML fragment from one component to be used in another?

8.5 Projecting templates at runtime with ngContent

In some cases, a parent component needs to render arbitrary markup within a child at runtime, and you can do that in Angular using *projection*. You can project a fragment of the parent component's template onto its child's template by using the `ngContent` directive. This is a two-step process:

1. In the child component's template, include the tags `<ng-content></ng-content>` (the *insertion point*).
2. In the parent component, include the HTML fragment that you want to project into the child's insertion point between tags representing the child component (for example, `<my-child>`):

```
template: `
  ...
  <my-child>
    <div>Passing this div to the child</div>
  </my-child>
  ...
```

In this example, the parent component won't render the content placed between `<my-child>` and `</my-child>`. Listings 8.17 and 8.18 illustrate this technique. Notice that both components declare a CSS style selector with the same name, `.wrapper`, but each of them defines a different background color. This illustrates what Angular has to offer in terms of style encapsulation, described in the next section.

Consider an example with two components—parent and child. The parent component will pass an HTML fragment to the child for rendering. The code of the child component is shown in the following listing.

Listing 8.17 child.component.ts

```
import {Component, ViewEncapsulation} from "@angular/core";

@Component({
  selector: 'child',
  styles: ['.wrapper {background: lightgreen;}'],
```

The class selector to render the UI on the light green background

```
   template: `
    <div class="wrapper">
     <h2>Child</h2>
      <div>This content is defined in child</div>
      <p>
      <ng-content></ng-content>
     </div>
  `,
  encapsulation: ViewEncapsulation.Native
 })
export class ChildComponent {}
```

The content that comes from the parent is displayed here.

For styles, use the ViewEncapsulation.Native mode (we explain view encapsulation modes in the next section).

The parent component is shown in the next listing.

Listing 8.18 app.component.ts

```
@Component({
  selector: 'app-root',
  styles: ['.wrapper {background: deeppink;}'],
   template: `
    <div class="wrapper">
     <h2>Parent</h2>
      <div>This div is defined in the Parent's template</div>
      <child>
        <div ><i>Child got this line from parent </i></div>
       </child>
    </div>
  `,
  encapsulation:ViewEncapsulation.Native
})
export class AppComponent {}
```

The class selector to render the UI on the light green background

The content will be projected onto the child's template.

Run this app with the following command in the Chrome browser:

```
ng serve --app projection1 -o
```

The Chrome browser will render the UI shown in figure 8.12.

The text "Child got this line from parent" was projected from the `AppComponent` onto the `Child-Component`. You may ask why you would want to run this app in the Chrome browser: because you specified `ViewEncapsulation.Native`, assuming that the browser supports Shadow DOM, and Chrome supports this feature. The next section provides more details.

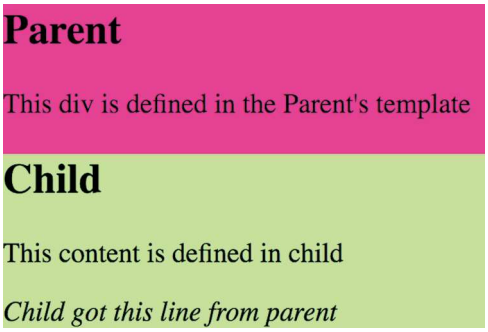


Figure 8.12 Running the projection1 app with `ViewEncapsulation.Native`

NOTE `ViewEncapsulation` modes aren't related to projection and can be used in any component, but we wanted to use the app that has a differently styled parent and child components to introduce this feature.

8.5.1 View encapsulation modes

JavaScript modules allow you to introduce scope to your scripts so they don't pollute the global space in the browser or any other execution environment. What about CSS? Imagine a parent and child components that coincidentally declare a style with the same CSS class selector name, but define different background colors. Will the browser render components using different backgrounds or will both of them have the same background?

In short, Shadow DOM introduces scopes for CSS styles and encapsulation of DOM nodes in the browser. Shadow DOM allows you to hide the internals of a selected component from the global DOM tree. Shadow DOM is well explained in the article "Shadow DOM v1: Self-Contained Web Components" by Eric Bidelman, available at http://mng.bz/6VV6.

We'll use the app from the previous section to illustrate how Shadow DOM and Angular's `ViewEncapsulation` mode works. The `encapsulation` property of the `@Component()` decorator can have one of three values:

- `ViewEncapsulation.Native` —This can be used with browsers that support Shadow DOM.
- `ViewEncapsulation.Emulated` —By default, Angular emulates Shadow DOM support.
- `ViewEncapsulation.None` —If the styles have the same selectors, the last one wins.

TIP Read about CSS specificity at https://css-tricks.com/specifics-on-css-specificity.

As mentioned earlier, both parent and child components use the `.wrapper` style. In a regular HTML page, this would mean that the CSS rules of the child's `.wrapper` would override the parent's. Let's see if you can encapsulate styles in child components so they don't clash with parent styles, even if their names are the same.

Figure 8.13 shows the running application in `ViewEncapsulation.Native` mode with the Developer Tools panel open. The browser creates `#shadow-root` nodes for parent and child (see the two `#shadow-root` nodes on the right). If you're reading this book in color (the e-book), you'll see that the `.wrapper` style paints the background of the `<app-root>` a deep pink color. The fact that the child also has the `.wrapper` style that uses a light green color doesn't affect the parent. Styles are encapsulated. The child's `#shadow-root` acts like a wall preventing the child's styles from overriding the parent's styles. You can use `ViewEncapsulation.Native` only if you're sure that the users of your app will use browsers that support Shadow DOM.

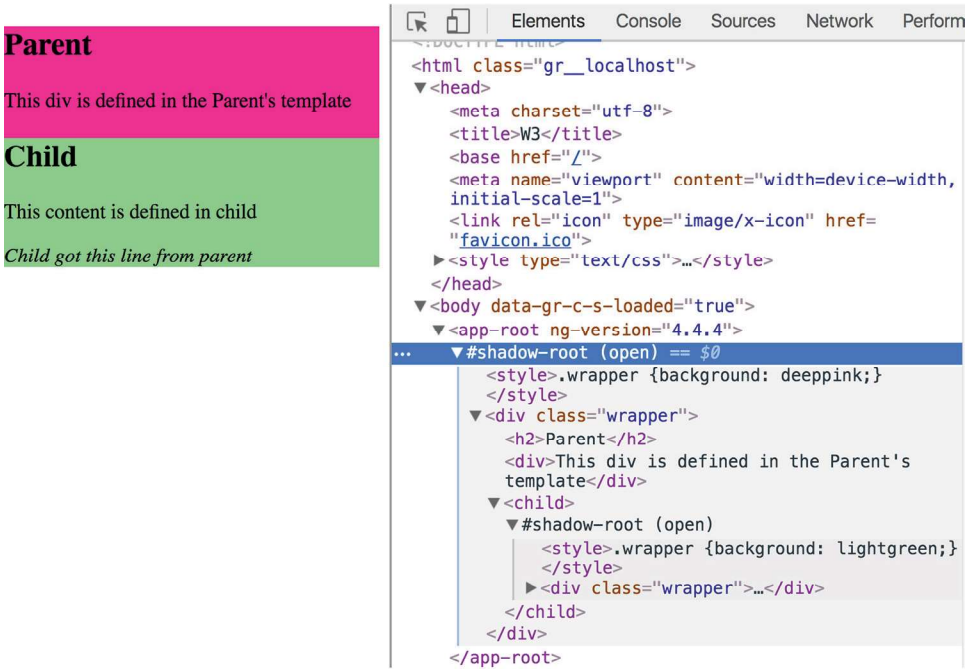


Figure 8.13 Browser creating two `#shadow-root` nodes

Figure 8.13 shows what happens after changing the value of the `encapsulation` property to `ViewEncapsulation.Emulated`. Angular uses this mode by default, so the effect is the same as if you didn't add the `encapsulation` property to the `@Component()` decorator. The DOM doesn't have any `#shadow-root` nodes inside the `<app-root>` element, but Angular generates additional attributes for the parent and child elements to differentiate styles in parent and child. Angular modifies all CSS selectors in component styles to incorporate generated attributes:

```
<div _ngcontent-c0="" class="wrapper">
 ...
<div _ngcontent-c1="" class="wrapper">
```

The styles in the <app-root> component

The styles in the <child> component

The UI is rendered the same way, using different background colors for these components as in figure 8.14, but the underlying code is not the same compared to figure 8.13.

Figure 8.15 shows the same example running with encapsulation set to `ViewEncapsulation.None`. In this case, the child's `wrapper` wins, and the entire window is shown with the child's light green background.

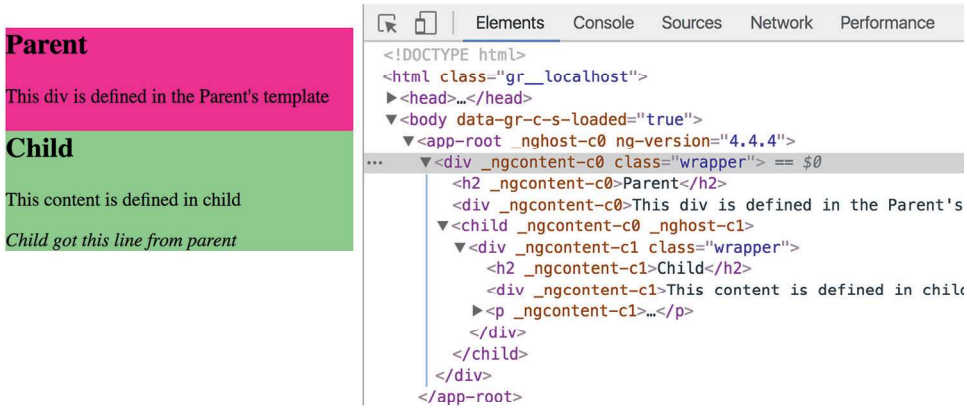


Figure 8.14 Running the projection1 app with `ViewEncapsulation.Emulated`

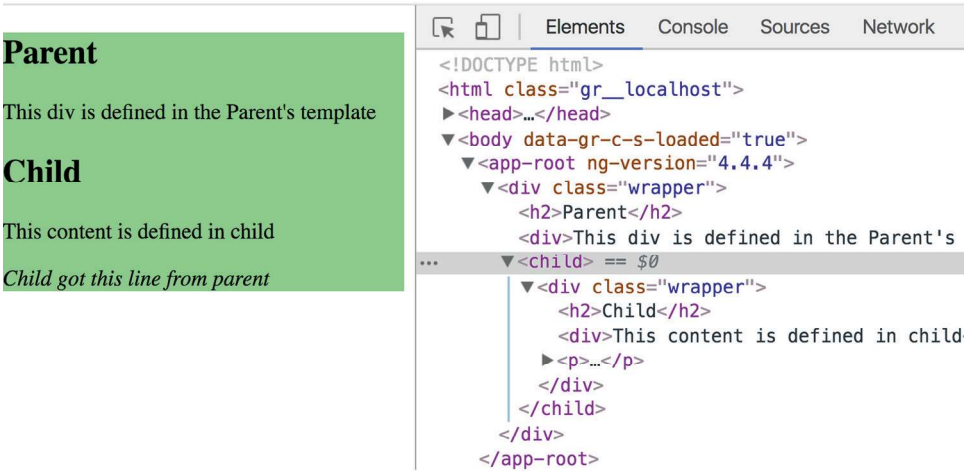


Figure 8.15 Running the projection1 app with `ViewEncapsulation.None`

Now that you understand encapsulation modes and basic projection, you may be wondering whether it's possible to project content into multiple areas of the component template.

8.5.2 Projecting onto multiple areas

A component can have more than one `<ng-content>` tag in its template. Let's consider an example where a child component's template is split into three areas: header, content, and footer, as in figure 8.16. The HTML markup for the header and footer could be projected by the parent component, and the content area could be defined in the child component. To implement this, the child component needs to include

two separate pairs of <ng-content></ng-content>s populated by the parent (header and footer).

To ensure that the header and footer content will be rendered in the proper <ng-content> areas, you'll use the select attribute, which can be any valid CSS selector (a CSS class, tag name, and so on). The child's template could look like this:

```
<ng-content select=".header"></ng-content>
<div>This content is defined in child</div>
<ng-content select=".footer"></ng-content>
```

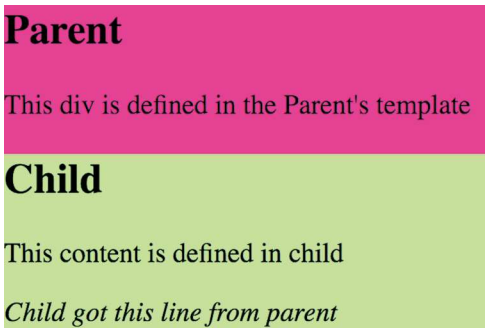


Figure 8.16 Running the projection2 app

The content that arrives from the parent will be matched by the selector and rendered in the corresponding area. We created a separate app in the folder projection2 to illustrate projection onto multiple areas. The following listing shows the child component.

Listing 8.19 child.component.ts

```
@Component({
  selector: 'child',
  styles: ['.wrapper {background: lightgreen;}'],
  template: `
    <div class="wrapper">
     <h2>Child</h2>
      <ng-content select=".header"></ng-content><p>

      <div>This content is defined in child</div><p>

      <ng-content select=".footer"></ng-content>
    </div>
  `
})
export class ChildComponent {}
```

Note that you have two <ng-content> slots now—one with the selector .header and another with .footer. The parent component will project different content into each slot. To make this example more dynamic, you use binding to display today's date in the header, as shown in the following listing.

Listing 8.20 app.component.ts

```
@Component({
  selector: 'app-root',
  styles: ['.wrapper {background: deeppink;}'],
  template: `
  <div class="wrapper">
    <h2>Parent</h2>
    <div>This div is defined in the Parent's template</div>
    <child>

      <div class="header">
         <i>Child got this header from parent {{todaysDate}}</i>
       </div>

      <div class="footer">
          <i>Child got this footer from parent</i>
      </div>
    </child>
  </div>
`
})
export class AppComponent {
  todaysDate = new Date().toLocaleDateString();
}
```

Projects this div onto the child's element with the header selector

Binds the current date to the projected content

Projects this div onto the child's element with the footer selector

NOTE The projected HTML can only bind the properties visible in the parent's scope, so you can't use the child's properties in the parent's binding expression.

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

```
ng serve --app projection2 -o
```

Running this app will render the page shown earlier in figure 8.16.

Using `<ng-content>` with the `select` attribute allows you to create a universal component with a view divided into several areas that get their markup from the parent.

Projection vs. direct binding to innerHTML

Alternatively, you can programmatically change the HTML content of a component by binding a component to `innerHTML`:

```
<p [innerHTML]="myComponentProperty"></p>
```

But using `<ng-content>` is preferable to binding to `innerHTML` for these reasons:

- `innerHTML` is a browser-specific API, whereas `<ng-content>` is platform independent.
- With `<ng-content>`, you can define multiple slots where the HTML fragments will be inserted.
- `<ng-content>` allows you to bind the parent component's properties into projected HTML.

Summary

- Parent and child components should avoid direct access to each other's internals but should communicate via input and output properties.
- A component can emit custom events via its output properties, and these events can carry an application-specific payload.
- Communications between unrelated components should be arranged using the Mediator design pattern. Either a common parent component or an injectable service can serve as a mediator.

9 Change detection and component lifecycle

This chapter covers

- How Angular knows that a UI update is needed
- Reviewing the milestones in the life of a component
- Writing code in component lifecycle hooks

All the apps you've developed so far have been properly updating the UI when the user or program updates the properties of your components. How does Angular know when to update the UI? In this chapter, we'll discuss the change detection (CD) mechanism that monitors the asynchronous events of your app and decides whether the UI should be updated or not.

We'll also discuss the lifecycle of an Angular component and the callback method hooks you can use to provide application-specific code that intercepts important events during a component's creation, lifespan, and destruction.

Finally, we'll continue working on ngAuction. This time, you'll add the view that displays product details.

9.1 A high-level overview of change detection

As the user works with your app, things change and the values of component properties (the model) get modified. Most of the changes happen asynchronously—for example, the user clicks a button, data is received from a server, an observable starts emitting values, a script invokes the `setTimeout()` function, and so on. Angular needs to know when the result of an asynchronous operation becomes available, to update the UI accordingly.

For automatic CD, Angular uses the library zone.js (the Zone). Angular subscribes to Zone events to trigger CD, which keeps the component's model and UI in sync. The CD cycle is initiated by any asynchronous event that happens in the browser. The change detector keeps track of all async calls made in components, services, and so forth; and when they complete, it makes a single pass from top to bottom of the component tree to see whether the UI of any component has to be updated.

> **NOTE** The CD mechanism applies changes in the component's properties to its UI. CD never changes the value of the component's property.

The zone.js library is one of the dependencies in your Angular project. It spares you from manually writing code to update UI, but starting with Angular 5, using the Zone is optional. To illustrate the role of zone.js, let's do an experiment: you'll create a simple project managed by the Zone first, and then you'll turn the Zone off. This project includes the `AppComponent` shown in the following listing.

Listing 9.1 The Zone is on

```
@Component({
  selector: 'app-root',
  template: `<h1>Welcome to {{title}}!</h1>`
})
export class AppComponent {

  title = 'app';

  constructor() {
    setTimeout(() => {this.title = 'Angular 5'}, 5000);
   }
}
```

Invokes the code asynchronously so the Zone will update the UI in five seconds

Running this app renders "Welcome to app!" Five seconds later, the message changes to "Welcome to Angular 5!" Let's change the app bootstrap code in the main.ts file to use the empty Zone object `noop`, introduced in Angular 5:

```
platformBrowserDynamic().bootstrapModule(AppModule, {ngZone: 'noop'});
```

Now running the same app will render "Welcome to app!" and this message will never change. You just turned off the Zone, and the app didn't update the UI.

> **NOTE** You can still initiate CD by injecting the `ApplicationRef` servicer in the app constructor and invoking its `tick()` method after updating the value of the `title` property.

An Angular application is structured as a tree of views (components), with the root component at the top of the tree. When Angular compiles component templates, each component gets its own change detector. When CD is initiated by the Zone, it makes a single pass, starting from the root down to the leaf components, checking to see whether the UI of each component needs to be updated (see the sidebar "Lifecycle hooks, change detection, and production mode" at the end of section 9.2 about CD in dev versus production). Is there a way to instruct the change detector not to visit each and every component upon every async property change?

9.1.1 Change detection strategies

For UI updates, Angular offers two CD strategies: `Default` and `OnPush`. If all components use the `Default` strategy, the Zone checks the entire component tree, regardless of where the change happened.

If a particular component declares the `OnPush` strategy, the Zone checks this component and its children only if the bindings to the component's input properties have changed, or if the component uses `AsyncPipe`, and the corresponding observable started emitting values.

If a component that has the `OnPush` strategy changes a value of one of its properties bound to its template, the change detection cycle won't be initiated. To declare the `OnPush` strategy, add the following line to the `@Component()` decorator:

```
changeDetection: ChangeDetectionStrategy.OnPush
```

Figure 9.1 illustrates the effect of the `OnPush` strategy using three components: the parent, a child, and a grandchild. Let's say a property of the parent was modified. CD will begin checking the component and all of its descendants.

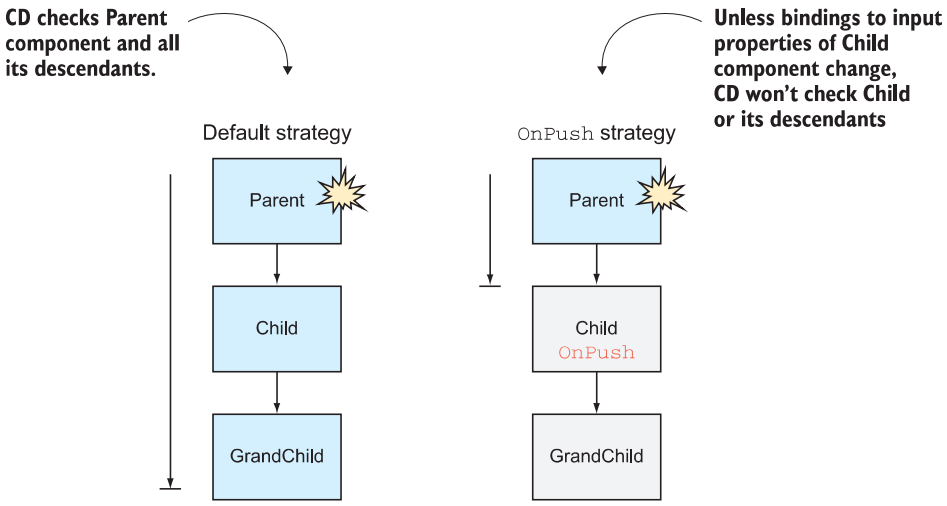


Figure 9.1 Change detection strategies

The left side of figure 9.1 illustrates the default CD strategy: all three components are checked for changes. The right side of figure 9.1 illustrates what happens when the child component has the `OnPush` CD strategy. CD starts from the top, but it sees that the child component has declared the `OnPush` strategy. If no bindings to the input properties have changed and no observable with `AsyncPipe` emits values (for example, via the `ActivatedRoute` parameters), CD doesn't check either the child or the grandchild.

Figure 9.1 shows a small application with only three components, but real-world apps can have hundreds of components. With the `OnPush` strategy, you can opt out of CD for specific branches of the tree.

Figure 9.2 shows a CD cycle caused by an event in the GrandChild1 component. Even though this event happened in the bottom-left leaf component, the CD cycle starts from the top; it's performed on each branch except the branches that originate from a component with the `OnPush` CD strategy and have no changes in the bindings to this component's input properties. Components excluded from this CD cycle are shown on a white background.

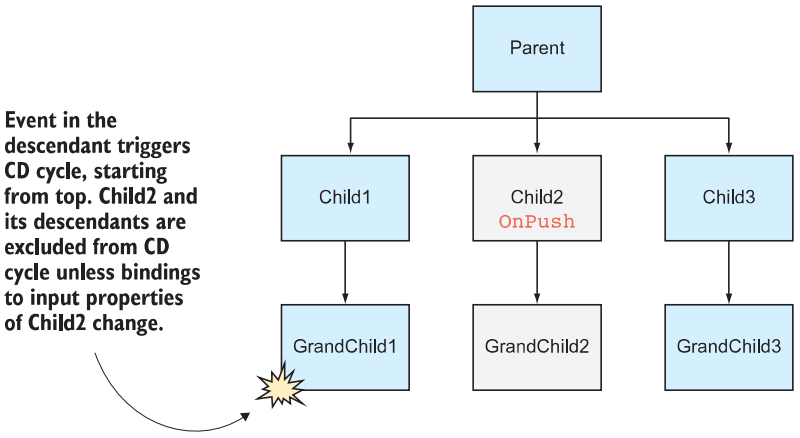


Figure 9.2 Excluding a branch from a CD cycle

This has been a brief overview of the CD mechanism. You should learn about CD in depth if you need to work on performance tuning of a UI-intensive application, such as a data grid containing hundreds of cells with constantly changing values. For in-depth coverage of change detection, see the article "Everything you need to know about change detection in Angular" by Maxim Koretskyi at http://mng.bz/0YqE.

In general, it's a good idea to make `OnPush` a default CD strategy for each component. If you see that the UI of a component doesn't get updated as expected, review the code and either switch back to the `Default` change detection strategy or manually initiate the CD pass by injecting the `ChangeDetectorRef` object and using its API (see https://angular.io/api/core/ChangeDetectorRef).

What if you have a slow-running component with lots of changing template elements? Could multiple passes of the change detector contribute to this slowness?

9.1.2 Profiling change detection

Listing 9.2 shows you how to profile change detection by enabling Angular debug tools. Change the app bootstrap code in main.ts to look like the following.

Listing 9.2 Enabling Angular debug tools

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {AppModule} from './app/app.module';
import {ApplicationRef} from '@angular/core';
import {enableDebugTools} from '@angular/platform-browser';

platformBrowserDynamic().bootstrapModule(AppModule).then((module) => {
  const applicationRef = module.injector.get(ApplicationRef);
  const appComponent = applicationRef.components[0];
  enableDebugTools(appComponent);
});
```

Gets a reference to the bootstrapped app

Enables Angular debug tools

Gets a reference to the app's top-level component

Launch your app, and in the browser console, enter the following command:

```
ng.profiler.timeChangeDetection({record: true})
```

Now your app will start reporting the time spent on each CD cycle, as shown in figure 9.3.

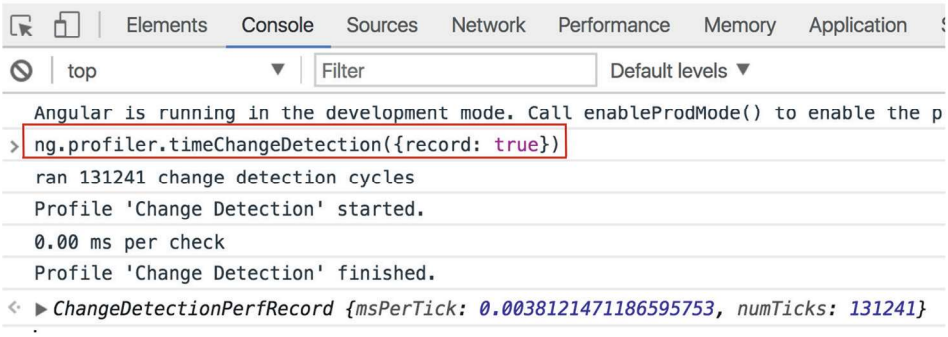


Figure 9.3 Profiling change detection

We've covered change detection, so now let's get familiar with the private life of a component.

9.2 Component lifecycle

Various events happen during the lifecycle of an Angular component: it gets created, reacts to different events, and gets destroyed. As explained in the last section, when a component is created, the CD mechanism begins monitoring it. The component is initialized, added to the DOM, and rendered by the browser. After that, the state of the component (the values of its properties) may change, causing rerendering of the UI, and, finally, the component is destroyed.

Figure 9.4 shows the lifecycle hooks (methods) where you can add custom code to intercept the lifecycle event and add your code there. If Angular sees any of these methods implemented in your app, it'll invoke them.

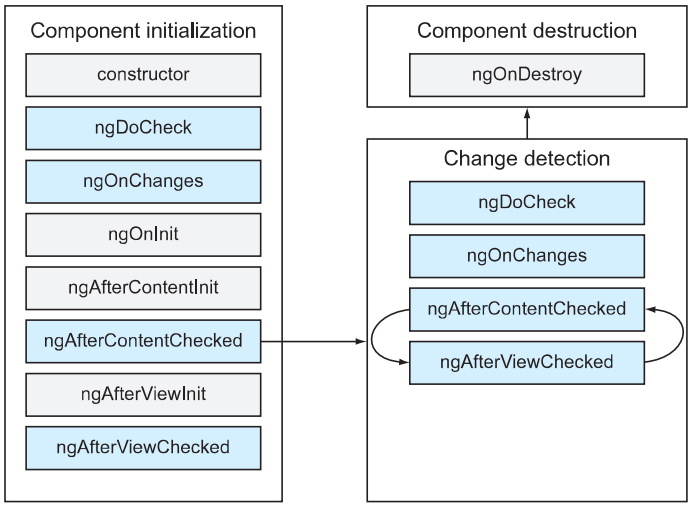


Figure 9.4 A component's lifecycle hooks

The callbacks shown on the light-gray background will be invoked only once, and those on the darker background can be invoked multiple times during the component life span. The user sees the component after the initialization phase is complete. Then the change detection mechanism ensures that the component's properties stay in sync with its UI. If the component is removed from the DOM tree as a result of the router's navigation or a structural directive (such as `*ngIf`), Angular initiates the destroy phase.

The constructor is invoked first when the instance of the component is being created, but the component's properties aren't initialized yet in the constructor. After the constructor's code is complete, Angular will invoke the following callbacks *if you implemented them*:

- `ngOnChanges()`—Called when a parent component modifies (or initializes) the values bound to the input properties of a child. If the component has no input properties, `ngOnChanges()` isn't invoked.

- `ngOnInit()`—Invoked after the first invocation of `ngOnChanges()`, if any. Although you might initialize some component variables in the constructor, the properties of the component aren't ready yet. By the time `ngOnInit()` is invoked, the component properties will have been initialized, which is why this method is mainly used for the initial data fetch.
- `ngDoCheck()`—Called on each pass of the change detector. If you want to implement a custom change detection algorithm or add some debug code, write it in `ngDoCheck()`. But keep in mind that placing any code in the `ngDoCheck()` method can affect the performance of your app because this method is invoked on each and every pass of the change detection cycle.
- `ngAfterContentInit()`—Invoked when the child component's state is initialized and the projection completes. This method is called only if you used `<ng-content>` in your component's template.
- `ngAfterContentChecked()`—During the change detection cycle, this method is invoked on the component that has `<ng-content>` after it gets the updated content from the parent if the bindings used in the projected content change.
- `ngAfterViewInit()`—Invoked after a component's view has been fully initialized. We used it in section 8.4 in chapter 8.
- `ngAfterViewChecked()`—Invoked when the change detection mechanism checks whether there are any changes in the component template's bindings. This callback may be called more than once as the result of modifications in this or other components.
- `ngOnDestroy()`—Invoked when the component is being destroyed. Use this callback to clean unneeded resources, for example, to unsubscribe from explicitly created subscriptions or remove timers.

Each lifecycle callback is declared in the interface with a name that matches the name of the callback without the prefix `ng`. For example, if you're planning to implement functionality in the `ngOnChanges()` callback, add `implements OnChanges` to your class declaration.

Let's consider some code samples illustrating the use of lifecycle hooks. The following code listing illustrates the use of `ngOnInit()`.

Listing 9.3 Fetching data in `ngOnInit()`

```
@Input() productId: number;  <-- Declares an input property

constructor(private productService: ProductService) { }

ngOnInit() {
   this.product = this.productService.getProductById(this.productId);
   }
```

Injects a service, but doesn't use it in the constructor

Uses the service in ngOnInit() to ensure that the productId is already initialized

This code uses the value of the input property `productId` as an argument of the `getProductById()` method. If you'd invoked `getProductById()` in the constructor, the `productId` property wouldn't be initialized yet. By the time `ngOnInit()` is invoked, `productId` is initialized, and you can safely invoke `getProductById()`.

The `ngOnDestroy()` hook is invoked when a component gets destroyed. For example, when you use the router to navigate from component A to component B, component A gets destroyed, and component B is created. If you created an explicit subscription in component A, don't forget to unsubscribe in `ngOnDestroy()`. This hook is also supported by Angular services.

9.2.1 Catching changes in the ngOnChanges hook

Now let's write a small app that uses `ngOnChanges()` and illustrates the different effects of bindings on primitive values versus object values. This app will include parent and child components, and the latter will have two input properties: `greeting` and `user`. The first property is a `string`, and the second is an `Object` with one property: `name`. To understand why the `ngOnChanges()` callback may or may not be invoked, you need to become familiar with the concept of mutable versus immutable values.

Mutable vs. immutable values

JavaScript strings are primitives, which are *immutable*—when a string value is created at a certain location in memory, you can't change it there. Consider the following code snippet:

```
let greeting = "Hello";
greeting = "Hello Mary";
```

The first line creates the value `Hello` in memory. The second line doesn't change the value at that address but creates the new string `Hello Mary` at a different memory location. Now you have two strings in memory, and each of them is immutable.

If the `greeting` variable was bound to an input property of a component, then its binding changed, because the value of this variable was initially at one memory location, and then the address changed.

JavaScript objects (as well as functions and arrays) are *mutable* and are stored in heap memory, and only references to objects are stored on the stack. After the object instance is created at a certain memory location, the reference to this object on the stack doesn't change when the values of the object's properties change in the heap memory. Consider the following code:

```
var user = {name: "John"};
user.name = "Mary";
```

After the first line, the object is created, and the reference to the instance of the `user` object is stored in stack memory and points at a certain memory location. The string `"John"` has been created at another memory location, and the `user.name` variable knows where it's located in memory.

After the second line of the preceding code snippet is executed, the new string `"Mary"` is created at another location. But the reference variable `user` is still stored in the same location on the stack. In other words, you mutated the content of the object but didn't change the value of the reference variable that points at this object. To make an object immutable, you need to create a new instance of the object whenever any of its properties changes.

TIP You can read more about JavaScript data types and data structures at http://mng.bz/bzL4.

Let's add the `ngOnChanges()` hook to the child component to demonstrate how it intercepts modifications of the input properties. This application has parent and child components. The child has two input properties (`greeting` and `user`). The parent component has two input fields, and the user can modify their values, which are bound to the input properties of the child. Let's see if `ngOnChanges()` will be invoked and which values it's going to get. The code of the parent component is shown in the following listing.

Listing 9.4 app.component.ts

```
@Component({
  selector: 'app-root',
  styles: ['.parent {background: deeppink}'],
  template: `
     <div class="parent">
       <h2>Parent</h2>
       <div>Greeting: <input type="text" [(ngModel)]="myGreeting">
        </div>
       <div>User name: <input type="text" [(ngModel)]="myUser.name">
        </div>
       <child [greeting]="myGreeting"
              [user]="myUser">
        </child>
    </div>
  `
})
export class AppComponent {
  myGreeting = 'Hello';
  myUser: {name: string} = {name: 'John'};
}
```

Uses two-way binding to synchronize entered greeting and myGreeting

Uses two-way binding to synchronize entered username and myUser.name

Binds myGreeting to child's input property greeting

Binds myUser to child's input property user

The child component receives the values from the parent component via its input variables. This component implements the `OnChanges` interface. In the `ngOnChanges()` method , you print the received data as soon as the binding to any of the input variable changes, as shown in the following listing.

Listing 9.5 child.component.ts

```
@Component({
  selector: 'child',
  styles: ['.child {background: lightgreen}'],
  template: `
    <div class="child">
      <h2>Child</h2>
      <div>Greeting: {{greeting}}</div>
      <div>User name: {{user.name}}</div>
    </div>
  `
})
export class ChildComponent implements OnChanges {
   @Input() greeting: string;
  @Input() user: {name: string};

  ngOnChanges(changes: {[key: string]: SimpleChange}) {

    console.log(JSON.stringify(changes, null, 2));
  }
}
```

Implements the OnChanges interface

Angular invokes ngOnChanges() when the bindings to input properties change.

When Angular invokes `ngOnChanges()`, it provides a `SimpleChange` object containing the old and new values of the modified input property and the flag indicating whether this is the first binding change. You use `JSON.stringify()` to pretty-print the received values.

Let's see if changing `greeting` and `user.name` in the UI results in the invocation of `ngOnChanges()` on the child component. We ran this app, deleted the last letter in the word *Hello*, and changed the name of the user from John to John Smith, as shown in figure 9.5.

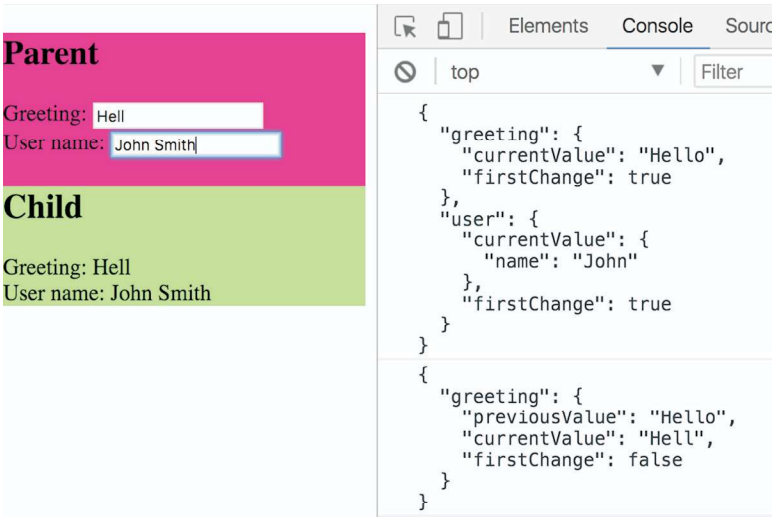


Figure 9.5 `ngOnChanges()` is invoked after the greeting change

Initially, `ngOnChanges()` was invoked for both properties. Note the `"firstChange": true`—this was the very first change in bindings. After we deleted the letter *o* in the greeting *Hello*, `ngOnChanges()` was invoked again, and the `firstChange` flag became `false`. But changing the username from *John* to *John Smith* didn't invoke `ngOnChanges()`, because the binding of the mutable object `myUser` didn't change.

To see this app in action, run `npm install` in the project lifecycle, and then run the following command:

```
ng serve --app lifecycle -o
```

Angular doesn't update bindings to input properties if only the object properties change, and that's why the `ngOnChanges()` on the child wasn't invoked. But the change detection mechanism still catches the change. That's why `"John Smith"`, the new value of the property `user.name`, has been rendered in the child component.

> **TIP** Add `changeDetection: ChangeDetectionStrategy.OnPush` to the template of `ChildComponent`, and its UI won't reflect changes in the parent's username. The binding to the child's `user` property doesn't change; hence, the change detector won't even visit the child for UI updates.

You probably appreciate the change detector for properly updating the UI, but what if you still need to programmatically catch the moment when the username changes and implement some code that handles this change?

9.2.2 Catching changes in the ngDoCheck hook

Suppose you want to catch the moment when a JavaScript object gets mutated. Let's rewrite the child component from the preceding section to use the `ngDoCheck()` callback instead of `ngOnChanges()`. The goals are as follows:

- Catch the moment when the object bound to an `Input()` property mutates.
- Find out which property of the bound object changed.
- Get the previous value of the changed property.
- Get the new value of this property.

To achieve these goals, you'll implement the `DoCheck` interface and use Angular's `KeyValueDiffers`, `KeyValueChangeRecord`, and `KeyValueDiffer`. You want to monitor the `user` object and its properties.

First, you'll inject the `KeyValueDiffers` service, which implements diffing strategies for various Angular artifacts. Second, you need to create an object of type `KeyValueDiffer` that will specifically monitor `user` object changes. When a change happens, you'll get an object of type `KeyValueChangeRecord` containing the properties `key`, `previousValue`, and `currentValue`. The code of the new child component is shown in the following listing.

Listing 9.6 child.component-docheck.ts

```
import {
  DoCheck, Input, SimpleChange, Component, KeyValueDiffers,
  ➥KeyValueChangeRecord, KeyValueDiffer} from "@angular/core";

@Component({
  selector: 'child',
  styles: ['.child {background: lightgreen}'],
  template: `
    <div class="child">
      <h2>Child</h2>
      <div>Greeting: {{greeting}}</div>
      <div>User name: {{user.name}}</div>
    </div>
  `
})
export class ChildComponent implements DoCheck {
  @Input() greeting: string;
  @Input() user: {name: string};

  differ: KeyValueDiffer<string, string>;

  constructor(private _differs: KeyValueDiffers) { }

  ngOnInit() {
   this.differ = this._differs.find(this.user).create();
   }

  ngDoCheck() {

    if (this.user && this.differ) {

      const changes = this.differ.diff(this.user);

      if (changes) {
        changes.forEachChangedItem(
            (record: KeyValueChangeRecord<string, string>) =>
             console.log(`Got changes in property ${record.key}
                 before: ${record.previousValue} after: ${record.currentValue}
      `));
      }
    }
  }
}
```

Declares a variable for storing differences

Injects the service for monitoring changes

Initializes the differ variable for storing differences in the user object

Implements the callback ngDoCheck()

Checks whether the properties of the user object changed

Gets the record of changes for each user property

Prints the changes on the console

The `diff()` method returns a `KeyValueChanges` object that includes the record about the change and offers such methods as `forEachAddedItem()`, `forEachChangedItem()`, `forEachRemovedItem()`, and more. In your component, you're interested only in catching changes, so you use `forEachChangedItem()`, which returns the `KeyValueChangeRecord` for each changed property.

The `KeyValueChangeRecord` interface defines the properties `key`, `currentValue`, and `previousValue`, which you print on the console. Figure 9.6 shows what happens after you delete the letter *n* in the User name input field, which was *John* originally.

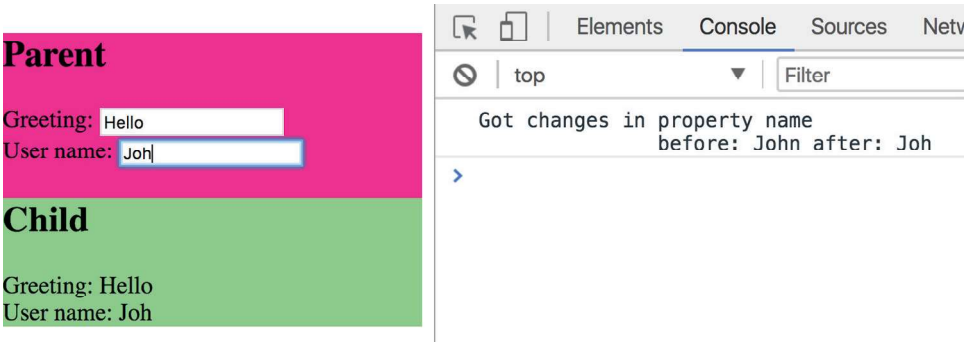


Figure 9.6 `ngDoCheck()` is invoked after each pass of the change detector.

Catching the username changes doesn't seem to be a practical use case, but some applications do need to invoke specific business logic whenever the value of a property changes. For example, financial applications may need to log each of a trader's steps. If a trader places a buy order at $101 and then immediately changes the price to $100, that must be tracked in a log file. This may be a good use case for catching such a change and adding logging in the `DoCheck()` callback.

To see this app in action, in the lifecycle/app.module.ts file, modify the import statement for the child component to `import {ChildComponent} from "./child.component-docheck";` and run the following command:

```
ng serve --app lifecycle -o
```

CAUTION We want to warn you once again: use `ngDoCheck()` only if you can't find another way of intercepting data changes, because it may affect the performance of your app.

Lifecycle hooks, change detection, and production mode

At the beginning of the chapter, we stated that the change detector makes one pass from top to bottom of the components tree to see if the component's UI should be updated. This is correct if your app runs in production mode, but in development mode (default), change detector makes two passes.

If you open the browser's console while running most of the apps from this book, you'll see a message stating that Angular is running in development mode, which performs assertions and other checks within the framework. One such assertion verifies that a change detection pass doesn't result in additional changes to any bindings (for example, your code doesn't modify the UI in the component lifecycle callbacks during the CD cycle). If your code tries to change the UI from one of the lifecycle callbacks, Angular will throw an exception.

(continued)
When you're ready to make a production build, turn on production mode so the change detector will make only one pass and won't perform the additional bindings check. To enable production mode, invoke `enableProdMode()` in your app before invoking the `bootstrap()` method. Enabling production mode will also result in better app performance.

Now that we've covered all the important parts of the component's life, let's continue working on ngAuction.

9.3 Hands-on: Adding the product view to ngAuction

In chapter 7, you created the home page of ngAuction. In this section, you'll create the product view, which will be rendered when the user clicks one of the product tiles in the home view. Figure 9.7 shows how the product view will look if the user selects Vintage Bluetooth Radio.

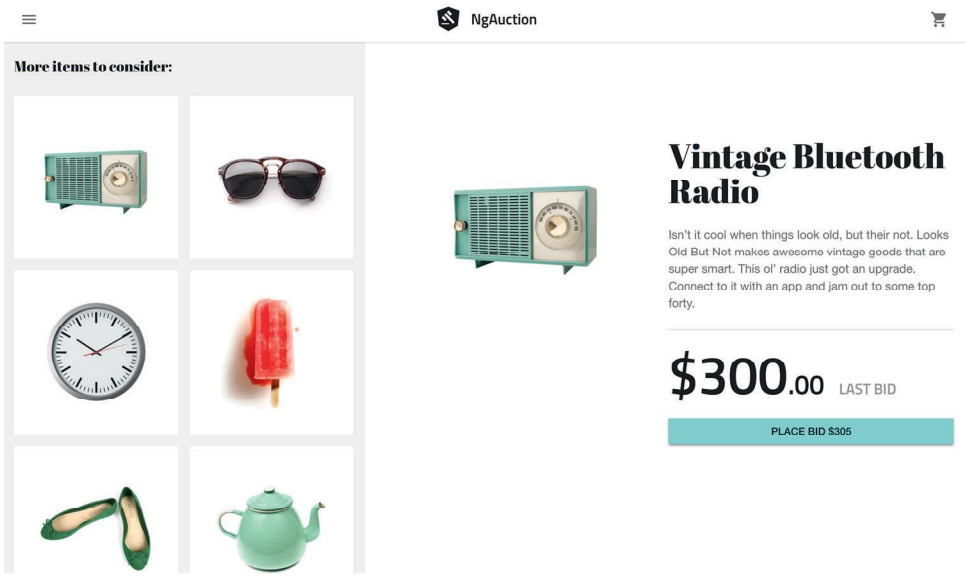


Figure 9.7 The product view

Besides information about the radio on the right, on the left are other suggested products that you want the user to consider. Amazon uses the same marketing technique while showing a product description. You've probably seen "More items to consider" or "Frequently bought together" sections on Amazon's product pages.

Depending on the viewport size, suggested products can be rendered either on the left or at the bottom of the product view.

The view shown in figure 9.7 will be implemented as `ProductComponent`, which will include two child components: `ProductDetailComponent` and `ProductSuggestionComponent`. In the product view, you'll use the Flex Layout library so that the UI layout will adjust to the available width of the viewport on the user's device.

One more thing: your product view will be implemented as a lazy-loaded feature module, explained in section 4.3 in chapter 4. Let's begin.

> **NOTE** If you created this version of ngAuction by following the explanations and instructions from the hands-on section in chapter 7, you can continue working on this app. You'll find the completed version of ngAuction with the implemented product view in the folder chapter9/ng-auction.

9.3.1 Creating product components and the module

You'll start with generating a `ProductModule` feature module by running the following command:

```
ng g m product
```

The command will create the product folder with the product.module.ts file. Because you'll use the Flex Layout library on the product view, add `FlexLayoutModule` to the `imports` property of the `@NgModule()` decorator, as shown in the following listing.

Listing 9.7 product.module.ts

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {FlexLayoutModule} from '@angular/flex-layout';

@NgModule({
  imports: [
    CommonModule,
    FlexLayoutModule
  ],
  declarations: []
})
export class ProductModule { }
```

This feature module will contain three components: `ProductComponent`, `ProductDetailComponent`, and `ProductSuggestionComponent`. The latter two will be subcomponents, and you want them to be located in separate subfolder under the product folder. You generate these components with the following commands:

```
ng g c product
ng g c product/product-detail
ng g c product/product-suggestion
```

These commands will generate three components and will add their names to the `declarations` property of the product module. You're going to lazy load the

ProductModule when the user clicks a particular product tile in the home component. To do that, you'll configure an additional route for the products/:productId path, so the app.component.ts file will look like the following listing.

Listing 9.8 app.routing.ts

```
import {Route} from '@angular/router';

export const routes: Route[] = [
  {
    path: '',
    loadChildren: './home/home.module#HomeModule'
  },
  {
    path: 'products/:productId',
    loadChildren: './product/product.module#ProductModule'
  }
];
```

Now you can proceed with implementing the components supporting the product view.

9.3.2 Implementing the product component

Your ProductComponent will serve as a wrapper for two child components: ProductDetailComponent and ProductSuggestionComponent. The product component implements the following functionality:

- It should be a default route of the product module.
- It should receive the product ID passed from the home component.
- It should get a reference to the ProductService object to receive product details.
- It should manage the layout of its children based on the viewport width.

To render the ProductComponent when the user navigates to the product view, you need to add the following listing to the ProductModule.

Listing 9.9 product.module.ts

```
...
import {RouterModule} from '@angular/router';

@NgModule({
  imports: [
   ...
    RouterModule.forChild([
      {path: '', component: ProductComponent}
    ])
  ],
  ...
})
export class ProductModule {}
```

In the preceding section, you configured the route for the path `'products/:productId'` in the root module, which means `ProductComponent` has to receive the requested product ID. You also need to inject `ProductService` in the constructor of `ProductComponent`, as shown in the following listing.

Listing 9.10 product.component.ts

```
import { filter, map, switchMap} from 'rxjs/operators';
import { Component } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { Product, ProductService } from '../shared/services';

@Component({
  selector: 'nga-product',
  styleUrls: [ './product.component.scss' ],
  templateUrl: './product.component.html'
})
export class ProductComponent {
  product$: Observable<Product>;
  suggestedProducts$: Observable<Product[]>;

  constructor(
    private route: ActivatedRoute,
    private productService: ProductService
  ) {
    this.product$ = this.route.paramMap
      .pipe(
        map(params => parseInt(params.get('productId') || '', 10)),
        filter(productId => !!productId),
        switchMap(productId => this.productService.getById(productId))
      );

    this.suggestedProducts$ = this.productService.getAll();
  }
}
```

Gets the product ID

Ensures that product ID is a valid number

Switches to the observable that retrieves details for the specified product

Initializes the observable for populating the suggested products

This component receives the product ID from the `ActivatedRoute` object. In section 6.6 of chapter 6, you saw the code that directly subscribes to the `paramMap`. In this case, you don't explicitly invoke the `subscribe()` method but will use the `async` pipe in the template. That's why you convert the given parameter from a string to a number with `parseInt()` in the `map` operator.

What if the user enters alpha characters in the URL instead of the product ID, such as http://localhost:4200/products/abc? In this case, `parseInt()` returns `NaN`, which you'll catch in the `filter` operator using double-bang syntax: `!!productId`. Non-alpha characters won't get through the `filter` operator.

The numeric product IDs will be given to the `switchMap` operator that switches over to the observable returned by the `getById()` method. To get the suggested products, you invoke the `getAll()` method .

NOTE Earlier, we stated that `ngOnInit()` is the right place for fetching data, but in this code sample, you do it in the constructor. Would it cause problems? Not in this case, because neither `getById()` nor `getAll()` uses component properties that would be initialized in the constructor.

Homework

The code for `ProductComponent` could use a couple of improvements, and we want you to implement them on your own.

Your product component invokes `productService.getAll()` to retrieve suggested products. This isn't exactly right. Say you select sunglasses. The product-detail component will show the description of the sunglasses, and the sunglasses will be also included as a suggested product. See if you can modify the implementation of the product component so it won't suggest the product that is already selected by the user.

If you enter an invalid product ID in the browser (such as http:// localhost:4200/products/abc) on the product page, you won't see any errors because the `filter()` operator will ignore this request, but the page will render only suggested products. To handle this scenario in a user-friendly manner, create a resolve guard, which will cancel the navigation if the service doesn't find the product with the provided ID and notify the user about it. For example, you can use the Angular Material snack-bar component for notifications (see http://mng.bz/1hx1).

The template of the product component will be implemented in the product .component .html file. It'll host `<nga-product-detail>` and `<nga-product-suggestion>` components and pass them the product(s) data to render, as shown in the following listing.

Listing 9.11 product.component.html

```
<div class="wrapper"
     fxLayout="column"
     fxLayout.>-md="row-reverse">

  <nga-product-detail
      fxFlex="auto"
       fxFlex.>-md="65%"
      *ngIf="product$ | async as product"
       [product]="product">
   </nga-product-detail>

  <nga-product-suggestion
      fxFlex="auto"
      fxFlex.>-md="35%"
      *ngIf="suggestedProducts$ | async as products"
       [products]="products">
   </nga-product-suggestion>
</div>
```

On larger than medium viewports, shows product detail on the right and suggested products on the left

Takes enough space to render this component but not more

If product$ emitted a value, puts it in the local template variable product

Passes the product object to <nga-product-detail> for rendering

If suggestedProducts$ emitted a value, puts it in the local template variable products

Passes products to <nga-product-suggestion> for rendering.

Best practice

Listing 9.11 uses the `async as` syntax for subscription. `async as product` means "define the local template variable `product`, and store the emitted value there." This syntax is useful when you need to reference the emitted object multiple times. Without the `async as` syntax, it could be written like this:

```
*ngIf = "product$ | async"
[product] = "product$ | async"
```

That would create two subscriptions instead of one, which you want to avoid, especially if a subscription triggers side effects like an HTTP request or some additional processing like filtering or sorting a large set of data.

NOTE From now on, to save space in this book, we won't include the content of the .scss files that contain styles of the ngAuction components. Please refer to the code samples that come with the book, found at https://github.com/Farata/angulartypescript and www.manning.com/books/angular-development-with-typescript-second-edition.

9.3.3 Implementing the product-detail component

The `ProductDetailComponent` is a presentation component that contains no business logic and renders the product provided via its input property. It's a child of the `ProductComponent`, as shown in the following listing.

Listing 9.12 product-detail.component.ts

```
@Component({
  selector: 'nga-product-detail',
  styleUrls: ['./product-detail.component.scss'],
  templateUrl: './product-detail.component.html'
})
export class ProductDetailComponent {
  @Input() product: Product;
}
```

The template of this component is shown in the next listing. It supports responsive web design (RWD) using the directives from the Flex Layout library.

Listing 9.13 product-detail.component.html

```
<div class="wrapper"
     ngClass.lt-md="wrapper--lt-md"
     ngClass.>-md="wrapper-->-md"
     fxLayout="row"
     fxLayoutAlign="center"
     fxLayout.xs="column"
     fxLayoutAlign.xs="center center">
```

Centers the content of the children both horizontally and vertically

```
<div fxFlex="50%">
   <img class="thumbnail"
       [attr.alt]="product.title"
       [attr.src]="product.imageUrl">
</div>

<div fxFlex="50%">
   <div class="info">
    <h1 class="info__title">{{product?.title}}</h1>
    <div class="info__description">{{product?.description}}</div>
    <div class="info__bid">
      <span>
        <span class="info__bid-value"
              ngClass.lt-md="info__bid-value--lt-md">
              ➥{{product?.price | currency: 'USD': 'symbol': '.0'}}</span>
        <span class="info__bid-value-decimal"
              ngClass.lt-md="info__bid-value-decimal--lt-md">.00</span>
      </span>
      <span class="info__bid-label">LAST BID</span>
    </div>

    <button class="info__bid-button"
            mat-raised-button
            color="accent">
      PLACE BID {{(product?.price + 5) | currency: 'USD': 'symbol': '.0'}}
    </button>
  </div>
 </div>
</div>
```

Half of the viewport is given to the product image.

Half of the viewport is given to the title, description, and bidding controls.

The last bid amount

Uses the button from Angular Material

The user can place bids in $5 increments.

In this version of the auction, you don't implement the bidding functionality. You'll do that in the hands-on section in chapter 13.

Because you use `mat-raised-button` from the Angular Material library, add the `MatButtonModule` to the product module

Listing 9.14 product.module.ts

```
....
import {MatButtonModule} from '@angular/material/button';
@NgModule({
  imports: [
    ...
    MatButtonModule
  ]
  ...
})
export class ProductModule {}
```

9.3.4 *Implementing the product-suggestion component*

In real-world online stores or auctions, the product-suggestion component shows similar products from the same category that the user may consider buying. In this version of ngAuction, you'll show all your products (you have only a dozen of them) under the caption "More items to consider," as you saw on the left in figure 9.7.

`ProductSuggestionComponent` is the second child of `ProductComponent`, and the content of the product-suggestion.component.ts file is shown in the following listing.

Listing 9.15 product-suggestion.component.ts

```
import { map, startWith } from 'rxjs/operators';
import { Component, Input } from '@angular/core';
import { ObservableMedia } from '@angular/flex-layout';
import { Observable } from 'rxjs';
import { Product } from '../../shared/services';

@Component({
  selector: 'nga-product-suggestion',
  styleUrls: [ './product-suggestion.component.scss' ],
  templateUrl: './product-suggestion.component.html'
})
export class ProductSuggestionComponent {
  @Input() products: Product[];
  readonly columns$: Observable<number>;
  readonly breakpointsToColumnsNumber = new Map([
    [ 'xs', 2 ],
    [ 'sm', 3 ],
    [ 'md', 5 ],
    [ 'lg', 2 ],
    [ 'xl', 3 ],
  ]);

  constructor(private media: ObservableMedia) {
     this.columns$ = this.media.asObservable()
      .pipe(
        map(mc => <number>this.breakpointsToColumnsNumber.get(mc.mqAlias)),
        startWith(3) // bug workaround
      );
  }
}
```

Sets the number of grid columns for different viewport sizes

Injects the ObservableMedia service from the Flex Layout library

Gets the number of grid columns based on the media query alias

The code for `ProductSuggestionComponent` is similar to the code for the `HomeComponent` developed for ngAuction in the hands-on section in chapter 7. In this case, you use different numbers of grid columns based on the viewport size, taking into account that a large portion of the screen will be occupied by the `ProductDetailComponent`. The template of the product suggestions component is shown in the following listing.

Listing 9.16 product-suggestion.component.html

```
<div class="info__title" fxLayout="row">
   More items to consider:
</div>

<mat-grid-list [cols]="columns$ | async" gutterSize="16px">

  <mat-grid-tile class="tile" *ngFor="let product of products">
     <a class="tile__content"
       fxLayout
       fxLayoutAlign="center center"
```

Subscribes to columns$ using the async pipe

For each product, renders a tile that wraps product info in the anchor tag

```
       [routerLink]="['/products', product.id]">
      <div class="tile__thumbnail"
           [ngStyle]="{'background-image':
➥'url(' + product.imageUrl + ')'}"></div>
    </a>
  </mat-grid-tile>
</mat-grid-list>
```

Shows another product info if the user clicks the tile

Because you use `<mat-grid-list>` from the Angular Material library, add the `MatGridListModule` to the product module.

Listing 9.17 product.module.ts

```
....
import { MatGridListModule } from '@angular/material/grid-list';
@NgModule({
  imports: [
    ...
    MatGridListModule
  ]
  ...
})
export class ProductModule {}
```

To run this version of ngAuction that implements routing and the product view, use the following command:

```
ng serve -o
```

Open the Network tab in Chrome Dev Tools and click one of the products. You'll see that the code and resources of the product module were lazy loaded.

In the hands-on section of chapter 11, you'll add search functionality and category tabs for easily filtering products by category.

Summary

- The change detection mechanism automatically monitors changes to components' properties and updates the UI accordingly.
- You can mark selected branches of your app component tree to be excluded from the change detection process.
- Writing the application code in the component lifecycle hook ensures that this code is executed in sync with UI updates.

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.

Listing 10.1 Preparing to use the template-driven Forms API

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

@NgModule({
  imports: [ BrowserModule,
             FormsModule]
  ...
})
class AppModule {}
```

Adds support for the template-driven Forms API

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.

Listing 10.2 Binding `NgForm` to a template variable

```
<form #f="ngForm"></form>
<pre>{{ f.value | json }}</pre>
```

Declares a local template variable f and binds it to ngForm

Displays the values of the form model

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:

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.value)"></form>
```

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 of chapter 2 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.

Listing 10.3 Adding the `NgModel` directive to an HTML element

```
<form #f="ngForm">
  <input type="text"
         name="username"
          ngModel>
 </form>
```

The name attribute is required so you can access its value in the code.

Ensures that this <input> field is included in the form model object

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.

Listing 10.4 A form with a nested form

```
<form #f="ngForm">
  <div ngModelGroup="passwords">
     <input type="text" name="password" ngModel>
    <input type="text" name="pconfirm" ngModel>
  </div>
</form>

<!-- Access the values from the nested object-->
<pre>Password: {{ f.value.passwords.password }}</pre>
 <pre>Password confirmation: {{ f.value.passwords.pconfirm }}</pre>
```

The ngModelGroup attribute requires a string value, which becomes a property name representing the nested form.

Accesses the values of the password and pconfirm form controls, using the nested object passwords for reference

Table 10.1 contains a summary of directives used in template-driven forms.

Table 10.1 Template-driven forms directives

Directive	Description
`NgForm`	Implicitly created directive that represents the entire form
`ngForm`	Used in templates to bind the template element (for example, `<form>`) to `NgForm`, typically assigned to a local template variable

Table 10.1 Template-driven forms directives *(continued)*

Directive	Description
NgModel	Implicitly created directive that marks the HTML element to be included in the form model
ngForm	Used in templates in form elements (for example, <input>) to be included in the form model
name	Used in templates in form elements to specify its name in the form model
NgModelGroup	Implicitly created directive that represents a part of the form, for example, password and confirm password fields
ngModelGroup	Used in templates to name a part of the form for future reference
ngSubmit	Intercepts the HTML form's submit event

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.

Listing 10.5 Angular-aware form

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)">
        <!-- Form controls will be added here -->
</form>
```

Binds NgForm to a local template variable

Submits the form, passing the form model to the event handler

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).

Listing 10.6 The username and ssn fields

```
<div>Username: <input type="text" name="username" ngModel></div>
 <div>SSN:      <input type="text" name="ssn"      ngModel></div>
```

The ngModel attribute makes this <input> element a part of the NgForm. You also add the name attribute with the value username.

Makes similar changes to the ssn field

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).

Listing 10.7 The password fields

```
<div ngModelGroup="passwordsGroup">
   <div>Password: <input type="password"
                  name="password" ngModel></div>
   <div>Confirm password: <input type="password"
                  name="pconfirm" ngModel></div>
 </div>
```

The ngModelGroup directive instructs NgForm to create a nested object within the form's value object that keeps the child fields.

Changes for the password and pconfirm fields are similar to those for ngModelGroup, but the values of the name attributes differ.

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

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

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

Listing 10.8 A component that uses the template-driven Forms API

```
@Component({
  selector: 'app-root',
  template: `
    <form #f="ngForm" (ngSubmit)="onSubmit(f.value)">
       <div>Username:        <input type="text"     name="username" ngModel>
       </div>
      <div>SSN:             <input type="text"     name="ssn"      ngModel>
      </div>
      <div ngModelGroup="passwordsGroup"> 2((CO8-2))
         <div>Password:        <input type="password" name="password" ngModel>
         </div>
         <div>Confirm password: <input type="password" name="pconfirm"  ngModel>
         </div>
      </div>
```

Binds NgForm to a local template variable and submits the form

Creates a nested group for passwords

```
      <button type="submit">Submit</button>
    </form>
  `
})
export class AppComponent {
  onSubmit(formData) {
    console.log(formData);
  }
}
```

The method handler for the onSubmit event

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:

```
ng serve --app template -o
```

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.

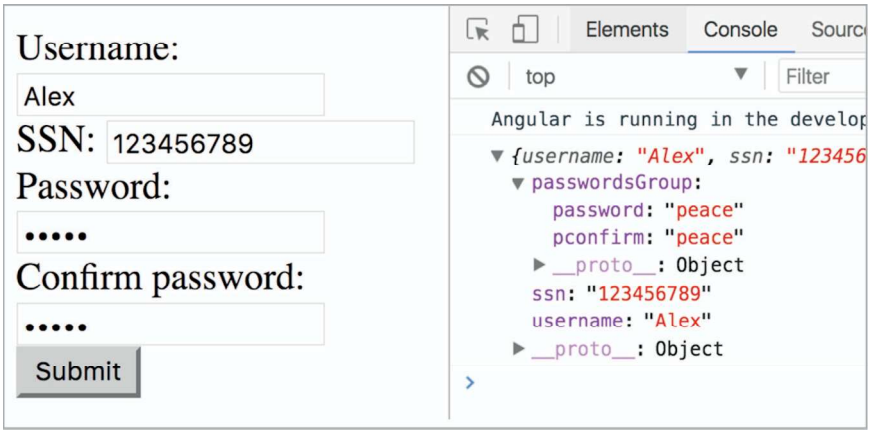


Figure 10.1 Running a template-driven registration form

Figure 10.2 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.

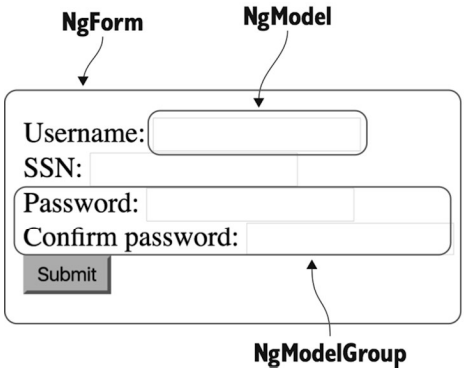


Figure 10.2 Form directives on the registration form

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.

Listing 10.9 Adding support for reactive forms

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

@NgModule({
  ...
  imports: [
    ...
    ReactiveFormsModule
  ],
  ...
})
```

Imports the module that supports reactive forms

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.

Listing 10.10 Creating a form model object

```
myFormModel: FormGroup;

  constructor() {
    this.myFormModel = new FormGroup({
       username: new FormControl(''),
       ssn: new FormControl('')
     });
  }
```

Creates an instance of a form model

Adds form controls to the form model

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:

```
let city = new FormControl('New York');
```

You can also create a `FormControl` attaching one or more built-in or custom validators. Chapter 11 covers form validation, but the following code listing shows how to attach two built-in Angular validators to a form control.

Listing 10.11 Adding validators to a form control

```
let city = new FormControl('New York',
                  [Validators.required,
                   Validators.minLength(2)]);
```

Creates a form control with the initial value New York

Adds a required validator to a form control

Adds a minLength validator 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 in section 6.3.

FORMGROUP

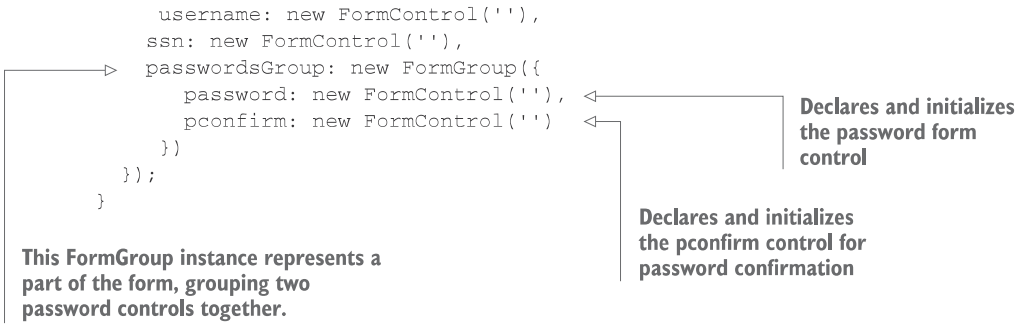
`FormGroup` is a collection of `FormControl` objects and represents either the entire form or its part. `FormGroup` aggregates the values and validity of each `FormControl` in the group. If one of the controls in a group is invalid, the entire group becomes invalid. The following listing shows the use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

Listing 10.12 Creating a form model by instantiating a `FormGroup`

```
myFormModel: FormGroup;

  constructor() {
    this.myFormModel = new FormGroup({
```

This FormGroup instance represents the entire form.

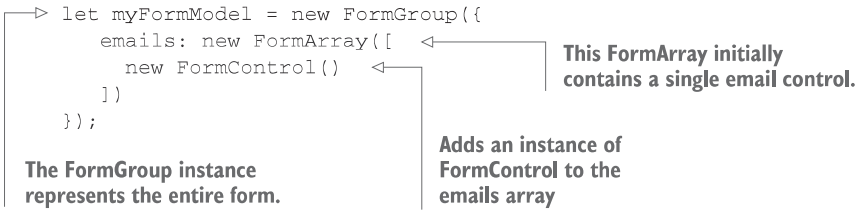


In section 10.3.6, you'll see a simplified syntax for creating form models with nesting.

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.

Listing 10.13 Adding a `FormArray` to a `FormGroup`



In section 10.3.4, 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.

Listing 10.14 Binding `FormGroup` to an HTML form

```
@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myFormModel">
     </form>
  `
})
class AppComponent {
  myFormModel = new FormGroup({
                 // form controls are created here });
}
```

Binds the instance of the form model to the formGroup directive

Creates an instance of the form model

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`.

Listing 10.15 Using `formGroupName`

```
@Component({
  ...
  template: `<form [formGroup]="myFormModel">
               <div formGroupName="dateRange">...</div>
             </form>`
})
class FormComponent {
  myFormModel = new FormGroup({
```

Binds the FormGroup that represents the entire form

Links this to the FormGroup called dateRange, defined in myFormModel

This FormGroup is bound to a DOM element using the formGroup directive in the template.

```
    dateRange: new FormGroup({
      from: new FormControl(),
      to  : new FormControl()
    })
  })
}
```

A child FormGroup named dateRange is bound to a DOM element using the formGroupName directive in the template.

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.

Listing 10.16 Completed form template

```
<form [formGroup]="myFormModel">
  <div formGroupName="dateRange">
    <input type="date" formControlName="from">
    <input type="date" formControlName="to">
  </div>
</form>
```

from is a property name in the model's nested group dateRange.

to is a property name in the model's nested group dateRange.

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 of chapter 6. The following listing shows the essence of that example from the Forms API perspective.

Listing 10.17 `FormControl`

With a standalone FormControl that's not a part of a FormGroup, you can't use the formControlName directive. Use formControl with the property binding.

Instead of defining a form model with FormGroup, create a standalone instance of a FormControl.

```
@Component({
   ...
   template: `<input type="text" [formControl]="weatherControl">`
})
class FormComponent {
  weatherControl: FormControl = new FormControl();
```

```
  constructor() {
    this.weatherControl.valueChanges
          .pipe(
           debounceTime(500),
           switchMap(city => this.getWeather(city))
          )
          .subscribe(weather => console.log(weather));
  }
}
```

Use the valueChanges observable to get the value from the form.

You could use `ngModel` (as section 2.6.2 of chapter 2) 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, 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 to use the reactive Forms API. The following listing uses the reactive Forms API, starting by creating a model object in TypeScript.

Listing 10.18 Creating a form model with the reactive API

```
@Component(...)
class AppComponent {
  myFormModel: FormGroup;

  constructor() {
    this.myFormModel = new FormGroup({
       username: new FormControl(),
      ssn: new FormControl(),
      passwordsGroup: new FormGroup({
         password: new FormControl(),
        pconfirm: new FormControl()
      })
    });
  }

  onSubmit() {
    console.log(this.myFormModel.value);
   }
}
```

Declares a component property myFormModel to hold a reference to the form model

Creates an instance of the form model

Creates a nested group for password fields

Prints the form model's values

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.

Listing 10.19 HTML binding to the model

Binds the <form> element to myFormModel using the formGroup directive

formControlName links input fields to the corresponding FormControl instances defined in the model.

```
<form [formGroup]="myFormModel"
       (ngSubmit)="onSubmit()">
  <div>Username: <input type="text" formControlName="username"></div>
   <div>SSN:      <input type="text" formControlName="ssn"></div>

   <div formGroupName="passwordsGroup">
      <div>Password: <input type="password"
                            formControlName="password"></div>

     <div>Confirm password: <input type="password"
                             formControlName="pconfirm"></div>
    </div>
  <button type="submit">Submit</button>
</form>
```

Links the model's nested FormGroup to the DOM element using formGroupName

Links the password input field and pconfirm using the formControlName directive

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:

```
ng serve --app reactive -o
```

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.

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.

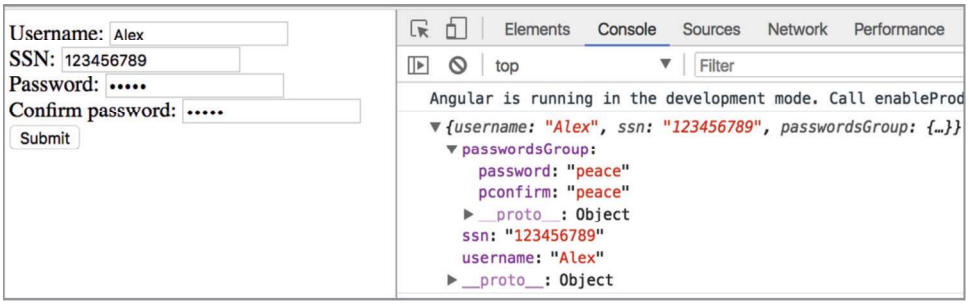


Figure 10.3 Running reactive registration form

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.

Listing 10.20 Using `FormArray` in the form model

```
@Component(...)
class AppComponent {
  formModel: FormGroup = new FormGroup({
    emails: new FormArray([
      new FormControl()
    ])
  });
  ...
```

Creates a FormGroup that will represent the form

Creates a FormArray for the emails form controls

Adds a single form control to the emails array

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.

Listing 10.21 Iterating through the `FormArray` in a template

Links the emails array to the <ul> element

Iterates through the emails array and creates a <li> with an <input> field for each array element

```
<ul formArrayName="emails">
   <li *ngFor="let e of formModel.get('emails').controls; let i = index">
      <input [formControlName]="i">
   </li>
</ul>
<button type="button" (click)="addEmail()">Add Email</button>
```

Defines the click event handler

Uses an emails array element index as a name of the corresponding <input> element

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`:

```
this.formModel.get('emails').push(new FormControl());
```

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:

```
ng serve --app dynamic -o
```

Figure 10.4 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.

Figure 10.4 Form with growable email controls

10.4 Forms API directives summary

You've used+ many different Forms API directives in both template-driven and reactive forms. Table 10.2 lists what they are for.

Table 10.2 Forms API directives

Directive	Description
Directives for reactive forms	
`FormGroup`	A class that represents the entire form or a subform; create its instance in the TypeScript code of the component. Its API is described at https://angular.io/api/forms/FormGroup.
`formGroup`	Used in templates to bind the template element (for example, `<form>`) to the explicitly created `FormGroup`; typically it's assigned to a variable declared in the component.
`formGroupName`	Used in templates to bind a group of the template elements (for example, `<div>`) to the explicitly created `FormGroup`.
`FormControl`	Represents the value, validators, and validity status of an individual form control; create its instance in the TypeScript code of the component. Its API is described at https://angular.io/api/forms/FormControl.
`formControl`	Used in a template to bind an individual HTML element to the instance of `FormControl`.
`formControlName`	Used in templates in form elements to link an individual `FormControl` instance to an HTML element.
`FormArray`	Allows you to create a group of form controls dynamically and use the array indexes as control names. You create an instance of `FormArray` in TypeScript code. Its API is described at https://angular.io/api/forms/FormArray.
`formArrayName`	Used in a template as a reference to the instance of `FormArray`.
Directives for template-driven forms	
`NgForm`	Implicitly created directive that represents the entire form; it creates an instance of `FormGroup`. Its API is described at https://angular.io/api/forms/NgForm.
`ngForm`	Used in templates to bind the template element (for example, `<form>`) to `NgForm`; typically, it's assigned to a local template variable.
`NgModel`	Implicitly created directive that marks the HTML element to be included in the form model. Its API is described at https://angular.io/api/forms/NgModel.
`ngModel`	Used in templates in form elements (for example, `<input>`) to be included in the form model.

Table 10.2 Forms API directives *(continued)*

Directive	Description
Directives for template-driven forms (continued)	
`name`	Used in templates in form elements to specify its name in the form model.
`NgModelGroup`	Implicitly created directive that represents a part of the form, such as password and confirm password fields. Its API is described at https://angular.io/api/forms/NgModelGroup.
`ngModelGroup`	Used in templates to name a part of the form for future reference.
Both template-driven and reactive forms	
`ngSubmit`	Intercepts the HTML form's submit event.

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.

Listing 10.22 Creating a form model

```
this.myFormModel = new FormGroup({
      id: new FormControl(''),
      description: new FormControl(''),
      seller: new FormControl('')
    });
```

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.

Listing 10.23 The data for populating the form

```
{
  id: 123,
  description: 'A great product',
  seller: 'XYZ Corp'
}
```

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

Listing 10.24 The data for updating the description

```
{
  description: 'The best product'
}
```

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.

Listing 10.25 The AppComponent

```
@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myFormModel">
       <div>Product ID: <input type="text" formControlName="id"></div>
      <div>Description: <input type="text" formControlName="description"></div>
      <div>Seller:      <input type="text" formControlName="seller"></div>
    </form>
    <button (click)="updateEntireForm()">Populate</button>
     <button (click)="updatePartOfTheForm()">Update Description</button>
     <button (click)="myFormModel.reset()">Reset</button>
  `
})
export class AppComponent {
  myFormModel: FormGroup;

  constructor() {
    this.myFormModel = new FormGroup({
       id: new FormControl(''),
      description: new FormControl(''),
      seller: new FormControl('')
    });
  }

  updateEntireForm() {
```

Binds the form model to formGroup

A button to invoke setValue()

A button to invoke reset()

A button to invoke patchValue()

Creates a form model object

```
    this.myFormModel.setValue({
       id: 123,
      description: 'A great product',
      seller: 'XYZ Corp'
    });
  }
  updatePartOfTheForm() {
    this.myFormModel.patchValue({
       description: 'The best product'
    });
  }
}
```

Invokes setValue() on the form model

Invokes patchValue() on the form model

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

```
ng serve --app populate -o
```

> **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. The template will remain exactly the same, but the following listing uses `FormBuilder` to construct the form model.

Listing 10.26 Creating a `formModel` with `FormBuilder`

```
constructor(fb: FormBuilder) {
   this.myFormModel = fb.group({
     username: [''],
     ssn: [''],
    passwordsGroup: fb.group({
       password: [''],
      pconfirm': ['']
    })
  });
}
```

Injects the FormBuilder service.

FormBuilder.group() creates a FormGroup using a configuration object passed to it.

Each FormControl is instantiated using the array that may contain an initial control's value and its validators.

Like FormGroup, FormBuilder allows you to create nested groups.

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.

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.

Listing 11.1 The `ValidatorFn` interface

```
interface ValidatorFn {
  (c: AbstractControl): ValidationErrors | null;
}
```

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`.

Listing 11.2 Attaching the `required` validator

```
import { FormControl, Validators } from '@angular/forms';
...
let username = new FormControl('', Validators.required);
```

Attaches the required validator to a FormControl

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.

Listing 11.3 Attaching two validators

```
let username = new FormControl('',
           [Validators.required, Validators.minLength(5)]);
```

Attaches required and minLength validators to a FormControl

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

```
let isValid: boolean = username.valid;
```

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.

Listing 11.4 Getting validators' errors

```
let errors: {[key: string]: any} = username.errors;
```

Gets all errors reported by validators

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 if the user enters an invalid phone number.

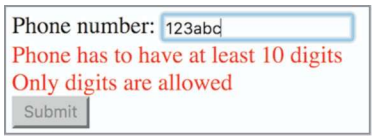


Figure 11.1 Showing validation errors

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.

Listing 11.5 Conditionally showing and hiding errors

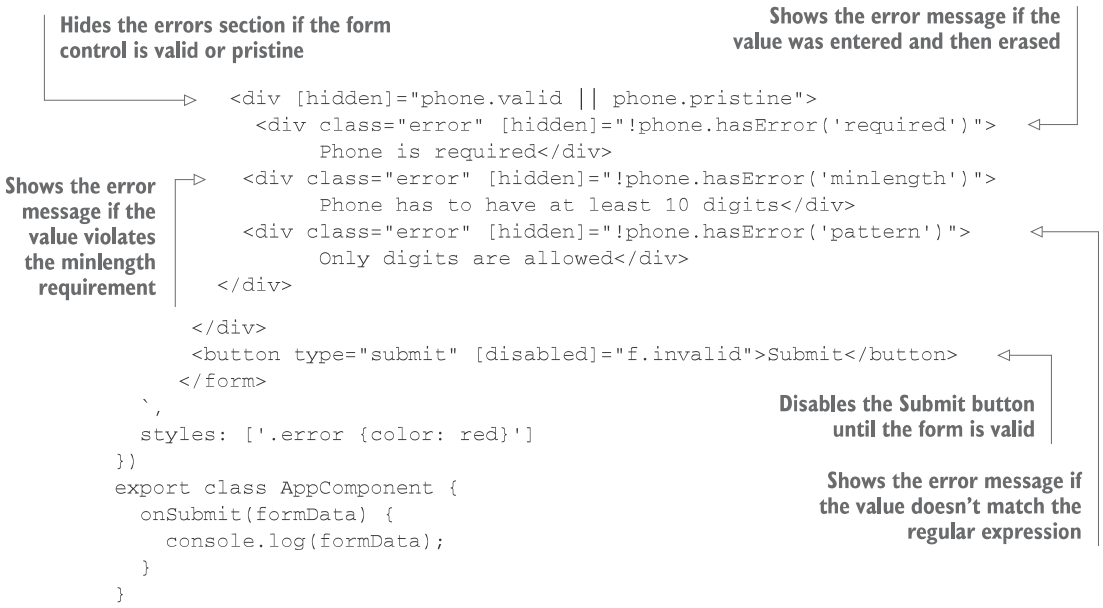
```
@Component({
  selector: 'app-root',
  template: `
    <form #f="ngForm" (ngSubmit)="onSubmit(f.value)" >
      <div>
        Phone Number:
        <input type="text" name="telephone" ngModel
               required
               pattern="[0-9]*"
               minlength="10"
               #phone="ngModel">
```

Adds the required validator

Adds the pattern validator to allow only digits

Adds the minlength validator

The local variable #phone gives access to the value of this control's model.




```
    <div [hidden]="phone.valid || phone.pristine">
      <div class="error" [hidden]="!phone.hasError('required')">
           Phone is required</div>
      <div class="error" [hidden]="!phone.hasError('minlength')">
           Phone has to have at least 10 digits</div>
      <div class="error" [hidden]="!phone.hasError('pattern')">
           Only digits are allowed</div>
    </div>
   </div>
   <button type="submit" [disabled]="f.invalid">Submit</button>
  </form>
`,
styles: ['.error {color: red}']
})
export class AppComponent {
  onSubmit(formData) {
    console.log(formData);
  }
}
```

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

```
ng serve --app threevalidators -o
```

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:

```
{
  minlength: {
    requiredLength: 7,
    actualLength: 5
  }
}
```

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`.

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

```
{
  required: true
}
```

In section 11.6, you'll find an example of how to extract the error description from the `ValidationErrors` object.

Table 11.1 provides a brief description of Angular built-in validators offered by the `Validators` class.

Table 11.1 Built-in validators

Validator	Description
`min`	A value can't be less than the specified number; it can be used only with reactive forms.
`max`	A value can't be greater than the specified number; it can be used only with reactive forms.
`required`	The form control must have a non-empty value.
`requiredTrue`	The form control must have the value `true`.
`email`	The form control value must be a valid email.
`minLength`	The form control must have a value of a minimum length.
`maxLength`	The form control can't have more than the specified number of characters.
`pattern`	The form control's value must match the specified regular expression.

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, 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 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.

Listing 11.6 Applying validators on blur using the reactive API

```
let telephone = new FormControl('',
    [{validators: Validators.minLength(10),
      updateOn:'blur'});
```

Attaches the minLength validator to a FormControl

Validates the value when focus moves out of the FormControl

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

Listing 11.7 Applying validators on blur using template-driven API

```
<form #f="ngForm"
      (ngSubmit)="onSubmit(f.value)"
      [ngFormOptions]="{updateOn: 'blur'}">
   ...
</form>
```

Each form control is validated when focus moves out of it.

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.

Listing 11.8 The interface custom validators must conform to

```
interface ValidatorFn {
  (c: AbstractControl): ValidationErrors | null;
 }
```

In case of errors, returns the ValidationErrors object; otherwise, null

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).

Listing 11.9 A sample custom validator

Validates the FormControl and returns either an error object or null

Gets the control's value if available, or uses an empty string otherwise

```
function ssnValidator(control: FormControl): ValidationErrors | null {
   const value = control.value || '';
   const valid = value.match(/^\d{9}$/);
   return valid ? null : { ssn: true };
 }
```

Matches the value against a regular expression that represents the SSN nine-digit format

If the value is an invalid SSN, returns the error object; the error name is 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.

Listing 11.10 Attaching custom validators to form controls

```
@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myForm">
      SSN: <input type="text" formControlName="socialSecurity">
         <span [hidden]="!myForm.hasError('ssn', 'socialSecurity')">
             SSN is invalid
         </span>
    </form>
  `
})
export class AppComponent {
  myForm: FormGroup;

  constructor() {
    this.myForm = new FormGroup({
      socialSecurity: new FormControl('', ssnValidator)
     });
  }
}
```

Shows an error message if the socialSecurity form control has the error named ssn

Attaches your custom ssnValidator

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.

Listing 11.11 Adding the error description in a custom validator

```
function ssnValidator(control: FormControl): {[key: string]: any} {
  const value: string = control.value || '';
  const valid = value.match(/^\d{9}$/);
  return valid ? null : {ssn: {description: 'SSN is invalid'}};
 }

@Component({
  selector: 'app',
  template: `
    <form [formGroup]="myForm">
      SSN: <input type="text" formControlName="socialSecurity">
           <span [hidden]="!myForm.hasError('ssn', 'socialSecurity')">
              {{myForm.getError('ssn', 'socialSecurity')?.description}}
            </span>
    </form>
  `
})
class AppComponent {
  myForm: FormGroup;

  constructor() {
    this.form = new FormGroup({
      'socialSecurity': new FormControl('', ssnValidator)
    });
  }
}
```

Creates a specific object with the description property that contains the description of the error

Shows an error message if the socialSecurity form control got the error named ssn

Gets the error message from the description property of the error

NOTE In listing 11.10, 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.

Listing 11.12 Using the `dirty` flag

```
<span [hidden]="!(myForm.get('socialSecurity').dirty
                  && myForm.hasError('ssn', 'socialSecurity'))">
        {{myForm.getError('ssn', 'socialSecurity')?.description}}
 </span>
```

Checks whether the form control has been modified

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.

Listing 11.13 Adding styles to the input field

```
@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myForm">
      SSN: <input type="text" formControlName="socialSecurity"
                  class="social">
       <span [hidden]="!(myForm.get('socialSecurity').dirty
                         && myForm.hasError('ssn', 'socialSecurity'))">
             {{myForm.getError('ssn', 'socialSecurity')?.description}}
      </span>
    </form>
  `,
  styles:[`.social.ng-dirty.ng-invalid {
              background-color: lightpink;
            }`]
})
```

Adds the CSS selector social

Is the field dirty and invalid?

Changes the background to light pink

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:

```
ng serve --app reactive-validator -o
```

Figure 11.2 shows how the browser will render this app if the value isn't valid.

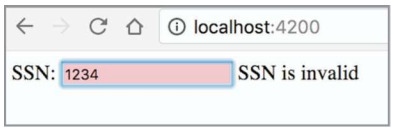


Figure 11.2 Showing the validation error and changing the background color

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.

Listing 11.14 A sample validator for a `FormGroup`

```
function equalValidator({value}: FormGroup): {[key: string]: any} {
  const [first, ...rest] = Object.keys(value || {});
   const valid = rest.every(v => value[v] === value[first]);
   return valid ? null : {equal: true};
}
```

Using rest parameters, gets the names of all properties of FormGroup.value

If equal, returns null; otherwise, returns an error object with the error named equal

Iterates through the properties' values to check if they're equal

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.

Listing 11.15 A modified form model for a user registration form

```
export class AppComponent {
  formModel: FormGroup;

  constructor(fb: FormBuilder) {
     this.formModel = fb.group({
       username: ['', Validators.required],
       socialSecurity: ['', ssnValidator],
       passwordsGroup: fb.group({
       password: ['', Validators.minLength(5)],
```

Injects the FormBuilder service

Creates the form model object

Creates the username control and attaches the required validator

Creates the socialSecurity control and attaches the ssnValidator

Creates the subgroup passwordsGroup for password and password confirmation controls

Creates the password control, applying the minLength validator

```
      pconfirm: ['']
    }, {validator: equalValidator})
  });
}

onSubmit() {
    console.log(this.formModel.value);
}
}
```

Creates the pconfirm control for confirming the password

Attaches the equalValidator to passwordsGroup to ensure that both entered passwords are the same

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.

Listing 11.16 The template and styles of the user registration component

```
template: `
    <form [formGroup]="formModel" (ngSubmit)="onSubmit()">
      <div>
        Username: <input type="text" formControlName="username">
        <span class="error"
              [hidden]="!formModel.hasError('required', 'username')">
               Username is required</span>
      </div>

      <div>
        SSN: <input type="text" formControlName="socialSecurity">
        <span class="error"
              [hidden]="!formModel.hasError('ssn', 'socialSecurity')">
               SSN is invalid</span>
      </div>

      <div formGroupName="passwordsGroup">
        <div>
          Password: <input type="password" formControlName="password">
          <span class="error"
                [hidden]="!formModel.hasError('minlength',
                ➥['passwordsGroup', 'password'])">
                 Password is too short</span>
        </div>

        <div>
          Confirm password: <input type="password" formControlName="pconfirm">
          <span class="error"
                [hidden]="!formModel.hasError('equal', 'passwordsGroup')">
                 Passwords must be the same</span>
        </div>
      </div>

      <button type="submit" [disabled]="formModel.invalid">Submit</button>
    </form>
  `,
  styles: ['.error {color: red;} ']
```

If the value of the username control is invalid, shows the error message

If the value of the socialSecurity control is invalid, shows the error message

If the value of the password control is invalid, shows the error message

If passwords are not the same, shows the error message

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:

```
ng serve --app groupvalidators -o
```

Figure 11.3 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.

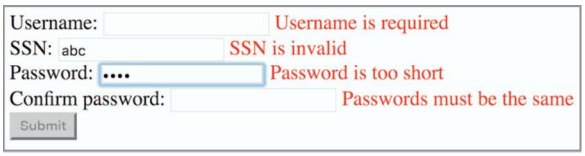


Figure 11.3 Showing multiple validation errors

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

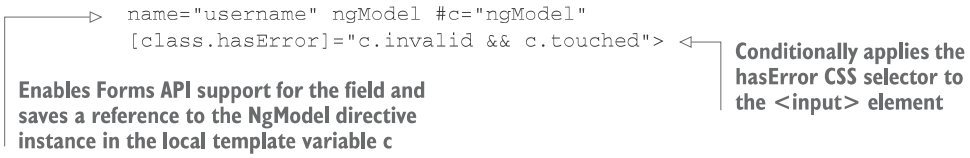
In addition to checking a control's validity, you can also use the `touched` and `untouched` properties to check whether a form control was visited by the user. If the user puts the focus into a form control using the keyboard or mouse and then moves the focus out, this control becomes `touched`; while 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.

Listing 11.17 Using the touched property

Defines a CSS selector that highlights the border of the invalid form control with red

```
<style>.hasError {border: 1px solid red;}</style>
<input type="text" required
```

Adds the required validator for the username field



> **NOTE** All the properties discussed in section 11.5.1 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 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.

Listing 11.18 Dynamically changing validators

```
@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myFormModel">
      Country: <input type="text" formControlName="country">
      <br>
      Phone: <input type="text" formControlName="phone">

      <span class="error" *ngIf="myFormModel.controls['phone'].invalid &&
                                 myFormModel.controls['phone'].dirty">
            Min length: {{this.myFormModel.controls['phone']
          ➥.getError('minlength')?.requiredLength}}
          </span>
    </form>
  `,
  styles: ['.error {color: red;}']
})
export class AppComponent implements OnInit{
  myFormModel: FormGroup;

  countryCtrl: FormControl;
  phoneCtrl: FormControl;

    constructor(fb: FormBuilder) {
      this.myFormModel = fb.group({
        country: [''],
       phone: ['']
      });
    }

    ngOnInit(){
      this.countryCtrl = this.myFormModel.get('country') as FormControl;
       this.phoneCtrl = this.myFormModel.get('phone') as FormControl;

      this.countryCtrl.valueChanges.subscribe( country => {
           if ('USA' === country){
            this.phoneCtrl.setValidators([Validators.minLength(10)]);
           }else{
            this.phoneCtrl.setValidators([Validators.minLength(11)]);
           }
          this.phoneCtrl.updateValueAndValidity();
        }
      );
    }
}
```

Displays the error message only if the phone was modified and is invalid

Subscribes to the changes in the country control

Gets the reference to the instance of the phone control

Creates the form model using FormBuilder

Gets the reference to the instance of the country control

Sets the phone validator for the USA

Sets the phone validator for other countries

Emits the updated validator to the subscribers of valueChanges

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

```
ng serve --app dynamicvalidator -o
```

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 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`.

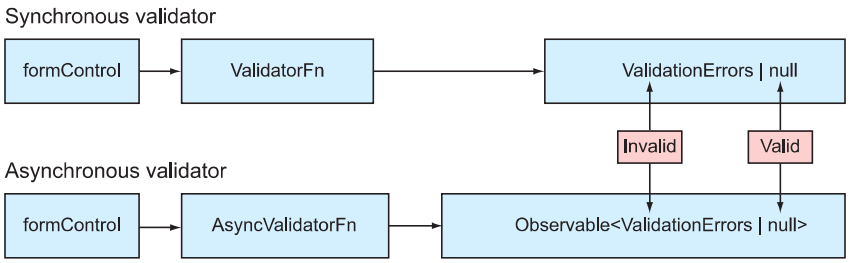


Figure 11.4 Comparing sync and async validators

> **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. 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.

Listing 11.19 A service with an async validator

```
@Injectable()
export class SsnValidatorService {

  checkWorkAuthorization(field: AbstractControl):
                         Observable<ValidationErrors | null> {

    // In the real-world app you'd make an HTTP call to server
```

This function properly implements the async validator interface.

```
    // to check if the value is valid
    return Observable.of(field.value.indexOf('123') >=0 ? null
              : {work: " You're not authorized to work"});
  }
}
```

Returns an observable of null—the validation passed

Returns an observable of the ValidationErrors object—the validation failed

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

Listing 11.20 async-validator/app.component.ts

```
function ssnValidator(control: FormControl): {[key: string]: any} {
   const value: string = control.value || '';
  const valid = value.match(/^\d{9}$/);
  return valid ? null : {ssn: true};
}

@Component({
  selector: 'app-root',
  template: `
    <form [formGroup]="myForm">
      <h2>Sync and async validation demo </h2>

      Enter your SSN: <input type="text" formControlName="ssnControl">
       <span *ngIf ="myForm.hasError('ssn', 'ssnControl');
       ➥else validSSN"> SSN is invalid.</span>

       <ng-template #validSSN> SSN is valid</ng-template>

       <span *ngIf ="myForm.hasError('work', 'ssnControl')">
       ➥{{myForm.get('ssnControl').errors.work}}</span>
    </form>
  `
})
export class AppComponent{

  myForm: FormGroup;

  constructor(private ssnValidatorService: SsnValidatorService) {
    this.myForm = new FormGroup({
      ssnControl: new FormControl('',
                    ssnValidator,
                    ssnValidatorService.checkWorkAuthorization.bind
                    ➥(ssnValidatorService))
    });
  }
}
```

Synchronous validator

In case of an error, shows the text from ; otherwise, from the template validSSN

Defines the template validSSN

Extracts the description of the error named work

Attaches the sync validator to ssnControl

Attaches the async validator to ssnControl

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:

```
ng serve --app async-validator -o
```

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. 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.

Listing 11.21 SsnValidatorDirective

```
@Directive({
   selector: '[ssn]',
   providers: [{
    provide: NG_VALIDATORS,
     useValue: ssnValidator,
    multi: true
  }]
})
class SsnValidatorDirective {}
```

Declares a directive using the @Directive decorator

Defines the directive's selector to be used as an HTML attribute

Registers ssnValidator as an NG_VALIDATORS provider

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, 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:

```
<input type="text" name="my-ssn" ngModel ssn>
```

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:

```
ng serve --app template-validator -o
```

Chapter 10 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.

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.

Figure 11.5 The new search icon and product category tabs

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.

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

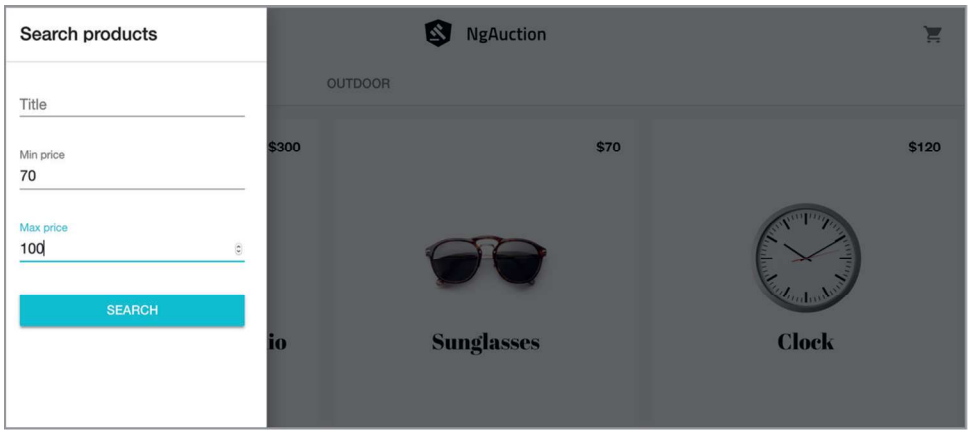


Figure 11.6 The search-form component

figure 11.7. That's because products from different categories can meet the search criteria—for example, a price between $70 and $100.

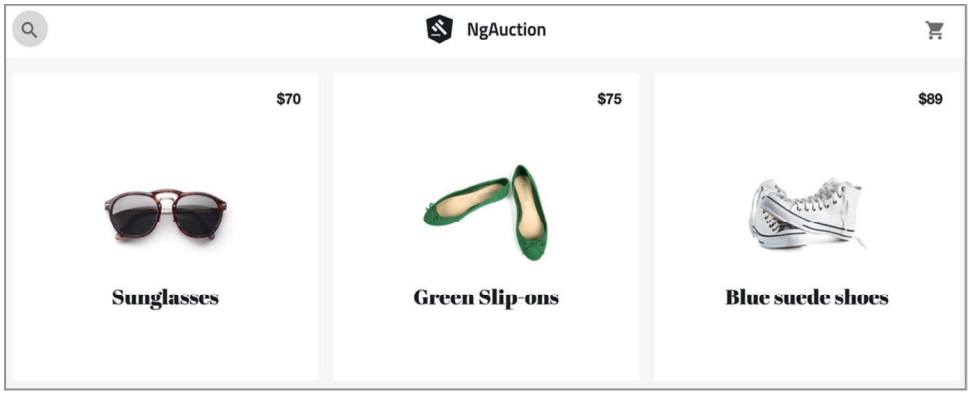


Figure 11.7 The search results view

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.

Listing 11.22 search-form.component.html

```
<h1 class="title">Search products</h1>

<form class="form" [formGroup]="searchForm" (ngSubmit)="onSearch()">
  <mat-form-field class="form__field">
    <input matInput type="text" placeholder="Title"
           formControlName="title">
    <mat-error>Title is too short</mat-error>
   </mat-form-field>

  <mat-form-field class="form__field">
    <input matInput type="number" placeholder="Min price"
                   formControlName="minPrice">
    <mat-error>Cannot be less than 0</mat-error>
   </mat-form-field>

  <mat-form-field class="form__field">
    <input matInput type="number" placeholder="Max price"
    formControlName="maxPrice" [errorStateMatcher]="matcher">
     <mat-error *ngIf="searchForm.controls['maxPrice'].hasError('min')">
           Cannot be less than 0</mat-error>
     <mat-error *ngIf="searchForm.controls['maxPrice'].hasError('max')">
           Cannot be more than 10000</mat-error>
    <mat-error *ngIf="searchForm.hasError('minLessThanMax')">
           Should be larger than min price</mat-error>
  </mat-form-field>

  <button class="form__submit" color="primary"
          mat-raised-button>SEARCH</button>
</form>
```

Title form control

Validation error message for Title

Min price control

Validation error message for negative values

Max price control

Shows an error message if the value is negative

Shows an error message if the entered max price is less than the min price

Shows an error message if the value is greater than max price

The matcher controls when to display validation errors.

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.

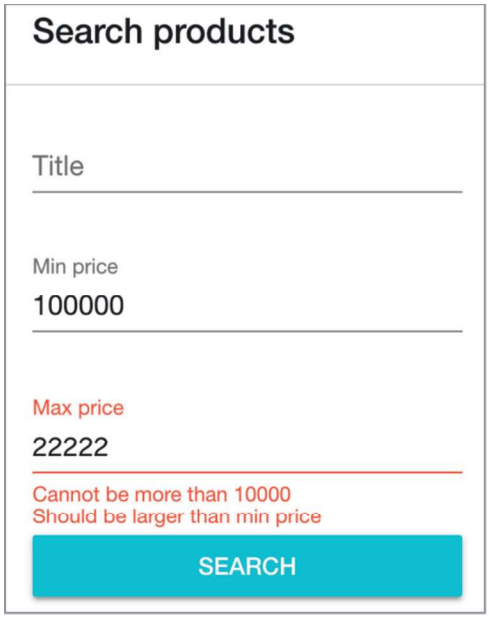


Figure 11.8 Showing two validation errors

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 in chapter 3). 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.

Listing 11.23 search-form.component.ts

```
@Component({
  selector: 'nga-search-form',
  styleUrls: [ './search-form.component.scss' ],
  templateUrl: './search-form.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class SearchFormComponent {
  @Output() search = new EventEmitter();
  readonly matcher = new ShowOnFormInvalidStateMatcher();
   readonly searchForm: FormGroup;

  constructor(fb: FormBuilder, private router: Router) {
    this.searchForm = fb.group({
       title    : [, Validators.minLength(2)],
      minPrice: [, Validators.min(0)],
      maxPrice: [, [Validators.min(0), Validators.max(10000)]]
    }, {
      validator: [ minLessThanMaxValidator ]
    });
  }

  onSearch(): void {
     if (this.searchForm.valid) {
      this.search.emit();
       this.router.navigate([ '/search-results' ],
         queryParams: withoutEmptyValues(this.searchForm.value)
       });
    }
  }
}

export class ShowOnFormInvalidStateMatcher implements
➥ErrorStateMatcher {
   isErrorState(control: FormControl | null,
  ➥form: FormGroupDirective | null): boolean {
    return !!((control && control.invalid) ||
    ➥(form && form.hasError('minLessThanMax')));
  }
}

function withoutEmptyValues(object: any): any {
   return Object.keys(object).reduce((queryParams: any, key) => {
    if (object[key]) { queryParams[key] = object[key]; }
    return queryParams;
  }, {});
}

function minLessThanMaxValidator(group: FormGroup):
                                  ValidationErrors | null {
  const minPrice = group.controls['minPrice'].value;
  const maxPrice = group.controls['maxPrice'].value;

  if (minPrice && maxPrice) {
    return minPrice <= maxPrice ? null : { minLessThanMax: true };
  } else {
```

The object that controls when to show the validation error for Max price

Creates a form model with validators

The user clicked the Search button.

Sends an event to app component to close the search-form component

Navigates to search-results, passing the search criteria

Doesn't send empty values in query parameters

Reports an error when either the form or a control is invalid

Creates a queryParams object that contains only the properties with values

A custom validator for comparing min and max prices

```
    return null;
  }
}
```

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** In ngAuction from chapter 10, the home component included `<mat-grid-list>` for rendering the list of products. In this version of ngAuction, we extracted the grid list into a separate product-grid component that now is reused by two components: categories and search-results (both belong to the 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.

Listing 11.24 search-results.component.ts

```
@Component({
  selector: 'nga-search',
  styleUrls: [ './search-results.component.scss' ],
  templateUrl: './search-results.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class SearchResultsComponent {
  readonly products$: Observable<Product[]>;

  constructor(
    private productService: ProductService,
    private route: ActivatedRoute
  ) {
    this.products$ = this.route.queryParams.pipe(
       switchMap(queryParams => this.productService.search(queryParams))
     );
  }
}
```

Declares an observable for products

Wraps the RxJS pipeable operator into the pipe() function

Passes the received parameters to the search() method

If you need a refresher on the `switchMap` operator, see section D.8 of appendix D. You can read about pipeable operators in section D.4.1.

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.

Listing 11.25 search-results.component.html

```
<div class="grid-list-container">
  <nga-product-grid [products]="products$ | async"></nga-product-grid>
</div>
```

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

```
@Input() products: Product[];
```

Then it renders the grid with products as described in section 9.3.4 in chapter 9.

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, 3, and 4, 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.

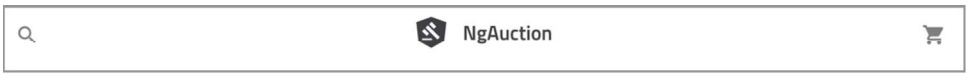


Figure 11.9 The search icon in a toolbar

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.

Listing 11.26 A fragment of app.component.html

Wraps the sidenav and the toolbar into <mat-sidenav-container>

Wraps the search-form component into <mat-sidenav>

On search event, closes the sidenav with the search-form component

```
<mat-sidenav-container>
  <mat-sidenav #sidenav>
    <nga-search-form (search)="sidenav.close()"></nga-search-form>
  </mat-sidenav>

  <mat-toolbar class="toolbar">
    <button mat-icon-button
            class="toolbar__icon-button"
            (click)="sidenav.toggle()">
      <mat-icon>search</mat-icon>
    </button>

    <!-- The markup for the logo and shopping cart is omitted -->

  </mat-toolbar>

  <router-outlet></router-outlet>
</mat-sidenav-container>
```

Declares a button with an icon

Clicking the icon button toggles the sidenav (opening it, in this case).

Uses the icon named search, offered by Google Material icons

REFACTORING THE HOME MODULE

In chapter 9, ngAuction had a home module with a home component. The home module still exists, but there's no home component anymore. You split its functionality into three components: categories, search-results, and product-grid components. The categories component is rendered below the navbar. Below the categories component, the browser renders the product-grid component that encapsulates the search-results component.

The routing has also changed, and the route configuration looks like the following listing now.

Listing 11.27 The modified configuration of the routes

By default, redirects to the componentless categories route

Adds the routing for the search-results component

```
const routes: Route[] = [
  { path: '', pathMatch: 'full', redirectTo: 'categories' },
  { path: 'search-results', component: SearchResultsComponent },
  { path: 'categories',
    children: [
      { path: '', pathMatch: 'full', redirectTo: 'all' },
      { path: ':category', component: CategoriesComponent },
    ]
  }
];
```

Adds the routing for the categories component with a parameter

By default, redirects to the categories/all route

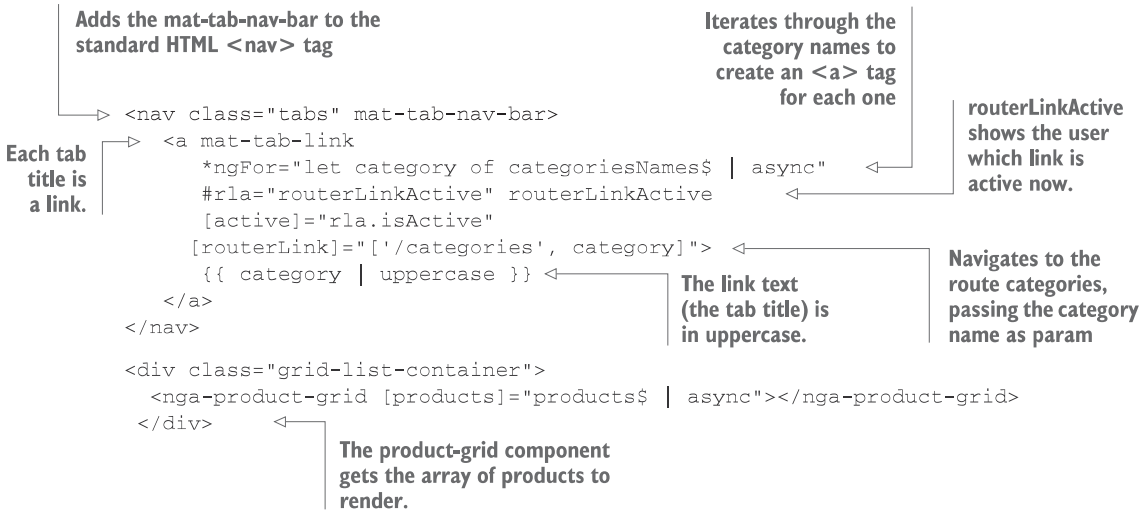
In this code, you use the so-called *componentless route* `categories`, which doesn't have a specific component mapped to the path. It consumes the URL fragment, providing it to its children. By default, the fragment *categories* and *all* will be combined into categories/all.

The parameters passed to the componentless route are passed further down to the child routes. In your case, if there were a parameter after the URL fragment *categories*, it would be passed to the `CategoriesComponent` via the `:category` path.

THE CATEGORIES COMPONENT

The categories component is a part of the home module. It uses the standard HTML `<nav>` element that's meant to hold a set of links for navigation. To make these links fancy, you add the Angular Material `mat-tab-nav-bar` directive to the `<nav>` element. In chapter 9, the home component rendered the grid of products, but now the user will see tabs with product category names. The user can click tabs to select all products or those that belong to a particular category, and the product-grid component will render them. The template of the categories component is shown in the following listing.

Listing 11.28 categories.component.html




```
<nav class="tabs" mat-tab-nav-bar>
  <a mat-tab-link
     *ngFor="let category of categoriesNames$ | async"
     #rla="routerLinkActive" routerLinkActive
     [active]="rla.isActive"
    [routerLink]="['/categories', category]">
     {{ category | uppercase }}
  </a>
</nav>

<div class="grid-list-container">
  <nga-product-grid [products]="products$ | async"></nga-product-grid>
 </div>
```

The TypeScript code of the categories component comes next. It uses `ProductService` to retrieve the distinct names of categories to be used as tab titles. It also uses the same service to retrieve the products of all categories or from the selected one.

Listing 11.29 categories.component.ts

```
@Component({
  selector: 'nga-categories',
  styleUrls: [ './categories.component.scss' ],
  templateUrl: './categories.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
```

```
})
export class CategoriesComponent {
  readonly categoriesNames$: Observable<string[]>;
   readonly products$: Observable<Product[]>;

  constructor(
    private productService: ProductService,
    private route: ActivatedRoute
  ) {
    this.categoriesNames$ =
         this.productService.getDistinctCategories().pipe(
             map(categories => ['all', ...categories]));

    this.products$ = this.route.params.pipe(
      switchMap(({ category }) => this.getCategory(category)));
    }

  private getCategory(category: string): Observable<Product[]> {
    return category.toLowerCase() === 'all'
      ? this.productService.getAll()
       : this.productService.getByCategory(category.toLowerCase());
   }
}
```

Category names to be used as tab titles

Creates an array of category names where the first one is "all"

Gets all products because the user clicked All

Gets corresponding products because the user clicked a tab with a specific category name

Gets products that belong to the selected category

This concludes the code review of ngAuction. To see it in action, run `npm install` in the project directory, and then run `ng serve -o`.

Summary

- Angular comes with several built-in validators, and you can create as many custom ones as you like.
- You can validate user input with synchronous and asynchronous validators.
- You can control when validation happens.