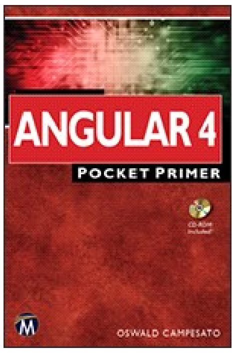


Angular 4 Pocket Primer

by Oswald Campesato

Chapter 5: Forms, Pipes, and Services

Overview

This chapter shows you how to create Angular applications that use Angular `Forms`, `Pipes`, and Services. The code samples rely on an understanding of functionality that is discussed in earlier chapters, such as how to make HTTP requests in Angular.

The first section in this chapter contains Angular applications that use Angular `Controls` and `Control Groups`. This section also provides an example of an Angular application that contains a form that makes `HTTP GET` requests, which enhances the example involving `HTTP`-related functionality in Chapter 4.

The second part of this chapter discusses Angular `Pipes` (the counterpart to `Filters` in Angular 1.x). You will also learn about `async` pipes in this section, which can eliminate the need for defining instance variables and also reduce the likelihood of memory leaks in Angular applications.

Overview of Angular Forms

An Angular `FormControl` represents a single input field, a `FormGroup` consists of multiple logically related fields, and an `NgForm` component represents a `<form>` element in an HTML Web page. The `ngSubmit` action for submitting a form has this syntax: `(ngSubmit)="onSubmit(myForm.value)"`.

Note that `NgForm` provides the `ngSubmit` event, whereas you must define the `onSubmit()` method in the component class. The expression `myForm.value` consists of the key/value pairs in the form. Later in the chapter you will see examples involving these controls, as well as `FormBuilder`, which supports additional useful functionality.

Angular also supports template-driven forms (with a `FormsModule`) and reactive forms (with a `ReactiveFormsModule`), both of which belong to `@angular/forms`. However, Reactive Forms are synchronous whereas template-driven forms are asynchronous.

Reactive forms

Reactive forms involve explicit management of the data flowing between a non-user interface (UI) data model and a UI-oriented form model that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

Reactive Forms involve the creation of a tree of Angular form control objects in the component class `app.component.ts`, which are also bound to native form control elements in the component template `app.component.html`.

The component class has access to the data model and the form control structure, which enables you to propagate data model values into the form controls and retrieve user-supplied values in the HTML controls. The component can observe changes in form control state and react to those changes. One advantage of working with form control objects directly is that value and validity updates are always synchronous and under your control. You won't encounter the timing issues that sometimes plague a template-driven form and reactive forms can be easier to unit test. Since reactive forms are created directly via code, they are always available, which enables you to immediately update values and "drill down" to descendant elements.

Template-Driven Forms

Template-driven forms involve placing HTML form controls (such as `<input>`, `<select>`, and so forth) in the component template. In addition, the form controls are bound to data model properties in the component via directives such as `ngModel`.

Note that Angular directives create Angular form objects based on the information in the provided data bindings. Angular uses `ngModel` to handle the transfer of data values, and also updates the mutable data model with user changes as they happen. Consequently, the `ngModel` directive does not belong to the `ReactiveFormsModule`.

DVD Before delving into the material in this section, the companion disc contains the Angular application `MasterForm`, which has form-related code. Although this code sample does not use Angular FormGroups, you might find some useful features in the code.

The next section shows you how to use the Angular `ngForm` component to create a form "the Angular way." Then you will see an example that shows you how to use an Angular `FormGroup` in an Angular Application.

An Angular Form Example

DVD Copy the directory `NGForm` from the companion disc into a convenient location. Listing 5.1 displays the contents of `app.component.ts` that illustrates how to use `<input>` elements with an `ngModel` attribute in an Angular application.

Listing 5.1: app.component.ts

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

```
@Component({
  selector: 'app-root',
  template: '
    <div>
      <h2>A Sample Form</h2>
      <form #f="ngForm"
            (ngSubmit)="onSubmit(f.value)"
            class="ui form">
        <div class="field">
          <label for="fname">fname</label>
          <input type="text"
                 id="fname"
                 placeholder="fname"
                 name="fname" ngModel>

          <label for="lname">lname</label>
          <input type="text"
                 id="lname"
                 placeholder="lname"
                 name="lname" ngModel>
        </div>

        <button type="submit">Submit</button>
      </form>
    </div>
   '
})
export class AppComponent {
  myForm: any;

  onSubmit(form: any): void {
    console.log('you submitted value:', form);
  }
}
```

Listing 5.1 defines a template property that contains a `<form>` element with two `<div>` elements, each of which contains an `<input>` element. The first `<input>` element is for the first name and the second `<input>` element is for the last name of a new user.

Angular provides the `NgModel` directive, which enables you to use the instance variable `myForm` in an Angular form. For example, the following code snippet specifies `myForm` as the control group for the given form:

```
<form [ngModel]="myForm"
   (ngSubmit)="onSubmit(myForm.value)"
```

Notice that `onSubmit` specifies `myForm` and that a `Control` is bound to the input element.

Note Add the attribute `novalidate` to the `<form>` element to disable browser validation.

Listing 5.2 displays the contents of `app.module.ts` that imports a `FormsModule` and includes it in the `imports` property.

Listing 5.2: app.module.ts

```
import { NgModule }       from '@angular/core';
import { FormsModule }    from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Listing 5.2 is straightforward: It contains two lines (shown in bold) involving the `FormsModule` that are required for this code sample.

Data Binding and ngModel

Angular supports three types of binding in a form: no binding, one-way binding, and two-way binding. Here are some examples:

```
<!-- no binding -->
<input name="fname" ngModel>

<!-- one-way binding -->
<input name="fname" [ngModel]="fname">

<!-- two-way binding -->
<input name="fname" [ngModel]="fname"
       (ngModelChange)="fname=$event">

<!-- two-way binding -->
<input name="fname" [(ngModel)]="fname">
```

The one-way binding example will look for the `fname` property in the associated component and initialize the `<input>` field with the value of the `fname` property.

The two-way binding example fires the `ngModelChange` event when users alter the value of the `<input>` field, which causes an update to the `fname` property in the component, thereby ensuring that the input value and its associated component value are the same. You can also replace the value of `ngModelChange` with the output of a function (e.g., capitalizing the text string that users enter in the input field).

The second example of two-way data binding uses the "banana in a box" syntax, which is a shorthand way of achieving the same result as the first two-way data binding example. However, this syntax does not support the use of a function that is possible with the longer syntax for two-way data binding.

Third-Party UI Components

Valor provides Bootstrap components for Angular, and its home page is located here:

https://valor-software.com/ng2-bootstrap/#/

This module provides Twitter Bootstrap components and a date picker, time picker, rating, and typeahead (among other components), and also works with Bootstrap version 3 and version 4.

Alternatively, you can use `ng2-bootstrap` in Angular applications. There are three steps required to install and use `ng2-bootstrap` in an Angular application. First, install `ng2-bootstrap` with this command:

```
npm install ng2-bootstrap --save
```

Second, add this snippet to `index.html` to reference Bootstrap styles:

```
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/
css/bootstrap.min.css" rel="stylesheet">
```

Third, insert a new entry into the `styles` array in `angular-cli.json`:

```
"styles": [
   "../node_modules/bootstrap/dist/css/bootstrap.min.css",
   "styles.css",
],
```

For additional instructions and code samples, navigate to this URL:

https://github.com/valor-software/ng2-bootstrap

The next portion of this chapter shows you how to work with forms in "the Angular way."

Angular Forms with FormBuilder

The `FormBuilder` class and the `FormGroup` class are built-in Angular classes for creating forms. `FormBuilder` supports the `control()` function for creating a `FormControl` and the `group()` function for creating a `FormGroup`.

DVD Copy the directory `FormBuilder` from the companion disc to a convenient location. Listing 5.3 displays the contents of `app.component.ts` that illustrates how to use an Angular form in an Angular application.

Listing 5.3: app.component.ts

```
import { Component }   from '@angular/core';
import { FormBuilder } from '@angular/forms';
import { FormGroup }   from '@angular/forms';
```

```
@Component({
  selector: 'app-root',
  template: '
      <div>
        <h2>A FormBuilder Form</h2>

        <form [formGroup]="myForm"
              (ngSubmit)="onSubmit(myForm.value)"
              class="ui form">

          <div class="field">
            <label for="fname">fname</label>
            <input type="text"
                   id="fname"
                   placeholder="fname"
                   [formControl]="myForm.controls['fname']">
          </div>

          <div class="field">
            <label for="lname">lname</label>
            <input type="text"
                   id="lname"
                   placeholder="lname"
                   [formControl]="myForm.controls['lname']">
          </div>

         <button type="submit">Submit</button>
       </form>
     </div>
    '
 })
 export class AppComponent {
 //myForm: any;
   myForm: FormGroup;

   constructor(fb: FormBuilder) {
     this.myForm = fb.group({
       'fname': ['John'],
       'lname': ['Smith']
     });
   }

   onSubmit(value: string): void {
     console.log('you submitted value:', value);
   }
 }
```

Listing 5.3 contains a `<form>` element with two `<div>` elements, each of which contains an `<input>` element. The first `<input>` element is for the first name and the second `<input>` element is for the last name of a new user.

In Listing 5.3, `FormBuilder` is injected into the constructor, which creates an instance of `FormBuilder` that is assigned to the `fb` variable in the constructor. Next, `myForm` is initialized by invoking the `group()` method that takes an object of key/value pairs. In this case, `fname` and `lname` are keys, and both of them appear as `<input>` elements in the `template` property. The values of these keys are optional initial values.

Obviously you can add many other properties inside the `group()` method (such as address-related fields). Moreover, you can add a different form for each new entity. For example, you could create separate forms for a `Customer`, `PurchaseOrder`, and `LineItems`.

Angular Reactive Forms

DVD Copy the directory `ReactiveForm` from the companion disc to a convenient location. Listing 5.4 displays the contents of `app.component.ts` that illustrates how to define a reactive Angular form in an Angular application.

Listing 5.4: app.component.ts

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

@Component({
  selector:    'app-root',
  templateUrl: './app.component.html',
  styleUrls:   ['./app.component.css']
})
export class AppComponent {
   userForm: FormGroup;
   disabled:boolean;

   constructor(fb: FormBuilder) {
     this.userForm = fb.group({
        name:    'Jane',
        email:   'jsmith@yahoo.com',
        address: fb.group({
          city:  'San   Francisco',
          state: 'California'
        })
     });
   }
   onFormSubmitted(theForm : FormGroup) {
      console.log("name  = "+theForm.controls['name'].value);
      console.log("email = "+theForm.controls['email'].value);
      console.log("city  = "+theForm.get('address.city').
                                                   value);

      console.log("city  = "+theForm.get('address.state').
                                                   value);
   }
}
```

Listing 5.4 contains the usual `import` statements. Notice how the variable `userForm`, which has type `FormBuilder`, is initialized in the constructor. In addition to two text fields, `userForm` contains the `address` element, which also has type `FormBuilder`.

Listing 5.5 displays the contents of `app.module.html` with an Angular form that contains `<input>` elements that correspond to the fields in the `userForm` variable.

Listing 5.5: app.component.html

```
<form [formGroup]="userForm"
                        (ngSubmit)="onFormSubmitted(userForm)">
  <label>
    <span>Name</span>
    <input type="text" formControlName="name"
placeholder="Name" required>
  </label>

  <div>
    <label>
      <span>Email</span>
      <input type="email" formControlName="email"
placeholder="Email" required>
    </label>
  </div>

  <div formGroupName="address">
    <div>
      <label>
        <span>City</span>
        <input type="text" formControlName="city"
                                  placeholder="City"   required>
      </label>
    </div>
    <label>
      <span>Country</span>
      <input type="text" formControlName="state"
                                 placeholder="State"  required>
    </label>
    </div>
      <br />
    <input type="submit" [disabled]="userForm.invalid">
</form>
```

Listing 5.5 contains very simple HTML markup that enables users to change the default values for each of the input fields.

Listing 5.6 displays the updated contents (shown in bold) of `app.module.ts`.

Listing 5.6: app.module.ts

```
 import { BrowserModule } from '@angular/platform-browser';
 import { NgModule }            from '@angular/core';
 import { FormsModule }         from '@angular/forms';
 import { HttpModule }          from '@angular/http';
 import { ReactiveFormsModule } from '@angular/forms';
 import { AppComponent }        from './app.component';

 @NgModule({
   declarations: [
     AppComponent
  ],
  imports:  [
    BrowserModule,
    FormsModule,
    HttpModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 5.6 contains one new line of code: an `import` statement for `ReactiveFormsModule` (which can be combined with the `import` statement for `FormsModule`), which is also referenced in the `imports` property.

FormGroup versus FormArray

As you now know, a `FormGroup` aggregates the values of `FormControl` elements into one object, where the control name is the key. Angular also supports `FormArray` (a "variation" of `FormGroup`), which aggregates the values of `FormControl` elements into an array.

`FormGroup` data is serialized as an array, whereas `FormArray` data is serialized as an object). If you do not know how many controls are in a given group, consider using a `FormArray` (otherwise use a `FormGroup`). The following link contains an example of using a `FormArray`:

https://alligator.io/angular/reactive-forms-formarray- dynamic-fields/

Other Form Features in Angular

The preceding section gave you a glimpse into the modularized style of Angular forms, and this brief section highlights some additional form-related features in Angular, such as the following:

- Form validation
- Custom validators
- Nested forms
- Dynamic forms
- Template-driven forms

Validators enable you to perform validation on form fields, such as specifying mandatory fields and the minimum and maximum lengths of fields. You can also specify a regular expression that a field must match, which is very useful for zip codes, email addresses, and so forth. Alternatively, you can specify validators programmatically.

Angular forms also provide event listeners that detect various events pertaining to the state of a form, as shown in the following code snippets:

```
{{myform.form.touched}}
```

```
{{myform.form.untouched}}
{{myform.form.pristine}}
{{myform.form.dirty}}
{{myform.form.valid}}
{{myform.form.invalid}}
```

For example, the following `<span>` element is displayed if one or more form fields is invalid:

```
<span *ngIf="!myform.form.valid">The Form is Invalid</span>
```

You can also display error messages using the `*ngIf` directive to display the status of a specific field, as shown here:

```
<label>
  <span>First Name</span>
<input type="text" formControlName="fname"
                                   placeholder="First Name">
  <p *ngIf="userForm.controls.fname.errors">
    This value is invalid
  </p>
</label>
```

You can find an example of a dynamic Angular form is here:

https://angular.io/docs/ts/latest/cookbook/dynamic-form. html

You can find an example of a template-driven Angular form here:

https://toddmotto.com/angular-2-forms-template-driven

Instead of using plain Cascading Style Sheets (CSS) for styling effects for field-related error messages, consider using something like ng2-bootstrap or Bootstrap (discussed briefly in the next section).

Angular Forms and Bootstrap 4 (optional)

Although Bootstrap is external to Angular, you probably want to add some styling effects to your Angular applications, especially if they contain an Angular form with many controls. For instance, the `ReactiveForm` application in a previous section provides a very plain-looking UI, and Bootstrap can help you greatly improve its appearance.

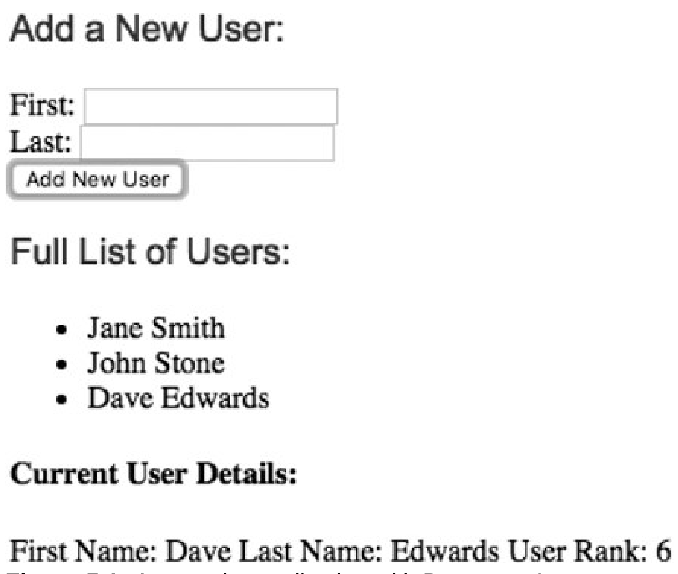
DVD However, you are not bound to Bootstrap, so this section only shows you some visual effects that are very easy to create with Bootstrap. If you want to see the corresponding code, copy the `FormBootstrap4` directory from the companion disc to a convenient location and look at the contents of `app.component.ts`.

Figure 5.1 displays the output from launching the `FormBootstrap4` application, which displays a form and buttons with an assortment of colors.

This concludes the portion of the chapter regarding `forms` in Angular. The next section discusses Angular Pipes, which provide useful functionality in Angular applications.

Working with Pipes in Angular

Angular supports something called a `pipe` (somewhat analogous to the Unix pipe "|" command), which enables you to filter data based on conditional logic (specified by you). Angular supports built-in pipes, asynchronous pipes, and support for custom pipes. The next two sections show you some examples of built-in pipes, followed by a description of asynchronous pipes. A separate section shows you how to define a custom Angular pipe.


Add a New User:

First:
Last:

Full List of Users:

- Jane Smith
- John Stone
- Dave Edwards

Current User Details:

First Name: Dave Last Name: Edwards User Rank: 6

Figure 5.1: An angular application with Bootstrap 4.

Working with Built-In Pipes

Angular supports various built-in pipes, such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. Each of these intuitively named pipes provides the functionality that you would expect: The `DatePipe` supports date values, the `UpperCasePipe` converts strings to uppercase, and so forth.

As a simple example, suppose that the variable food has the value `pizza`. Then the following code snippet displays the string `PIZZA`:

```
<p>I eat too much {{ food | UppercasePipe }} </p>
```

You can also parameterize some Angular pipes, an example of which is shown here:

```
<p>My brother's birthday is {{ birthday | date:"MM/dd/yy" }}
                                                          </p>
```

In fact, you can even chain pipes, as shown here:

```
My brother's birthday is {{ birthday | date | uppercase}}
```

In the preceding code snippet, birthday is a custom pipe (written by you). As another example, suppose that an Angular application contains the variable `employees`, which is an array of JavaScript Object Notation (JSON)-based data. You can display the contents of the array with this code snippet:

```
<div>{{employees | json }}</div>
```

The AsyncPipe

The Angular `AsyncPipe` accepts a `Promise` or `Observable` as input and subscribes to the input automatically, eventually returning the emitted values. Moreover, `AsyncPipe` is stateful: The pipe maintains a subscription to the input `Observable` and keeps delivering values from that Observable as they arrive.

The following code block gives you an idea of how to display stock quotes, where the variable `quotes$` is an `Observable`:

```
@Component({
  selector: 'stock-quotes',
  template: '
    <h2>Your Stock Quotes</h2>
    <p>Message: {{ quotes$ | async }}</p>
  '
})
```

Keep in mind that the `AsyncPipe` provides two advantages. First, `AsyncPipe` reduces boilerplate code. Second, there is no need to subscribe or to unsubscribe from an `Observable` (and the latter can help avoid memory leaks).

One other point: Angular does not provide pipes for filtering or sorting lists (i.e., there is no `FilterPipe` or `OrderByPipe`) because both can be compute intensive, which would adversely affect the perceived performance of an application.

The code sample in the next section shows you how to create a custom pipe that displays a filtered list of users based on conditional logic that is defined in custom code.

Creating a Custom Angular Pipe

DVD Copy the directory `SimplePipe` from the companion disc into a convenient location. Listing 5.7 displays the contents of `app.component.ts` that illustrates how to define and use a custom pipe in an Angular application that displays a subset of a hard-coded list of users.

Listing 5.7: app.component.ts

```
import { Component } from '@angular/core';
import {User}        from './user.component';
import {MyPipe}      from './pipe.component';

@Component({
  selector: 'app-root',
  template: '
    <div>
      <h2>Complete List of Users:</h2>
      <ul>
       <li
       *ngFor="let user of userList"
         (mouseover)='mouseEvent(user)'
         [class.chosen]="isSelected(user)">
         {{user.fname}}-{{user.lname}}<br/>
       </li>
      </ul>

      <h2>Filtered List of Users:</h2>
      <ul>
       <li
       *ngFor="let user of userList|MyPipe"
         (mouseover)='mouseEvent(user)'
         [class.chosen]="isSelected(user)">
         {{user.fname}}-{{user.lname}}<br/>
       </li>
      </ul>
    </div>
   `
})
export class AppComponent {
  user:User;
  currentUser:User;
  userList:User[];

  mouseEvent(user:User) {
     console.log("current user: "+user.fname+" "+user.lname);
     this.currentUser = user;
  }
  isSelected(user: User): boolean {
    if (!user || !this.currentUser) {
      return false;
    }

    return user.lname === this.currentUser.lname;
  //return true;
  }

  constructor() {
     this.userList = [
                  new User('Jane','Smith'),
                  new User('John','Stone'),
                  new User('Dave','Jones'),
                  new User('Rick','Heard'),
                 ]
  }
}
```

Listing 5.7 references a `User` custom component and a `MyPipe` custom component, where the latter is specified in the array of values for the `pipes` property. The `template` property displays two unordered lists of user names. The first list displays the complete list, and when users hover (with their mouse) over a user in the first list, the current user is set equal to that user via the code in the `mouseEvent()` method (defined in the `AppComponent` class). Note that the constructor in the `AppComponent` class initializes the `userList` array with a set of users, each of which is an instance of the `User` custom component.

The second list displays a filtered list of users based on the conditional logic in the custom pipe called `MyPipe`. Listing 5.8 displays the contents of `pipe.component.ts` that defines the pipe `MyPipe` that is referenced in Listing 5.7.

Listing 5.8: pipe.component.ts

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

@Pipe({
  name: "MyPipe"
})
export class MyPipe {
  transform(item) {
    return item.filter((item) => item.fname.startsWith("J"));
  //return item.filter((item) => item.lname.endsWith("th"));
  //return item.filter((item) => item.lname.contains("n"));
  }
}
```

Listing 5.8 contains the `MyPipe` class that contains the `transform()` method. There are three examples of how to define the behavior of the pipe, the first of which returns the users whose first name starts with an uppercase `J` (somewhat contrived, but nevertheless illustrative of pipe-related functionality).

Listing 5.9 displays the contents of the custom component `user.component.ts` for creating `User` instances, which is also referenced via an `import` statement in `app.component.ts`.

Listing 5.9: user.component.ts

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

@Component({
  selector: 'my-user',
  template: '<h1></h1>'
})
export class User {
  fname: string;
  lname: string;

  constructor(fname:string, lname:string) {
     this.fname = fname;
     this.lname = lname;
  }
}
```

The contents of Listing 5.9 are straightforward: there is a `User` class comprising the fields `fname` and `lname` for the first name and last name, respectively, for each new user.

Finally, we need to update the contents of `app.module.ts`, as shown in Listing 5.10, where the modified contents are shown in bold.

Listing 5.10: app.module.ts

```
import { NgModule }        from '@angular/core';
import { BrowserModule }   from '@angular/platform-browser';
import { AppComponent }    from './app.component';
import { MyPipe }          from './pipe.component';
import { User }            from './user.component';
@NgModule({
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, MyPipe, User ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

As you can see, Listing 5.10 contains two new `import` statements so that the custom components `MyPipe` and `User` can be referenced in the declarations property.

Launch the application and after a few moments you will see the following in a browser session:

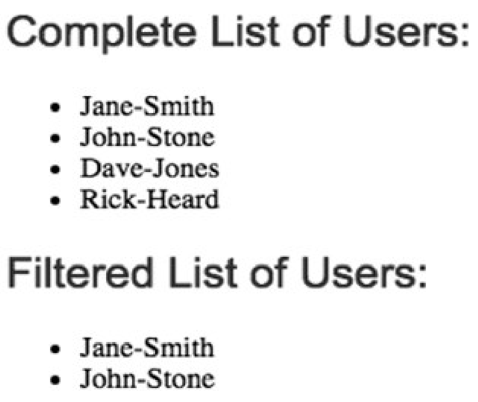
```
Complete List of Users:
```

- `Jane-Smith`
- `John-Stone`
- `Dave-Jones`
- `Rick-Heard`

```
Filtered List of Users:
```

- `Jane-Smith`
- `John-Stone`

Figure 5.2 displays the output from launching this Angular application and filtering based on the users whose first name starts with the capital letter `J`.



Complete List of Users:

- Jane-Smith
- John-Stone
- Dave-Jones
- Rick-Heard

Filtered List of Users:

- Jane-Smith
- John-Stone

Figure 5.2: Filtered user list via an Angular pipe.

Now that you understand how to define a basic `Pipe` in Angular, you can experiment with custom pipes that receive data asynchronously. This type of functionality can be very useful when you need to display data (such as a list or a table) after it's updated, without the need for "polling" the source of the data.

Additional information regarding Angular pipes is located here:

https://angular.io/docs/ts/latest/guide/pipes.html

This concludes the portion of the chapter regarding `Pipes` in Angular. The next section discusses Services in Angular applications.

What Are Angular Services?

Sometimes the front-end of a Web application contains presentation logic and some business logic. Angular components comprise the presentation tier and services belong to the business-logic tier. Define your Angular services in such a way that they are decoupled from the presentation tier.

Angular services are classes that implement some business logic, and are designed so that they can be used by components, models, and other services. In other words, services can be providers for other parts of an application.

Because of the Dependency Injection (DI) mechanism in Angular, services can be invoked in other sections of an Angular application. Moreover, Angular ensures that services are singletons, which means that each service consumer will access the same instance of the service class.

A sample Angular custom service is shown here:

```
@Injectable()
export class UpperCaseService {
  public upper(message: string): string {
    return message.toUpperCase();
  }
}
```

The simple service `UpperCaseService` provides one method that takes a string as an argument and returns the uppercase version of that string. The `@Injectable()` decorator is required so that this class can be injected as a dependency. Although this decorator is not mandatory in all cases, it's a good idea to mark your services in this manner. Use the `@Injectable` decorator only when a service (or class) "receives" an injection.

The following is an example of an `app.component.ts` class that invokes the method in the preceding service:

```
import {UpperCaseService} from "./path/to/service/
UpperCaseService";

@Component({
  selector: "convert",
  template: "<button (click)='greet()'>Greet</button>";
})
export class UpperComponent {
  // inject the custom service in the constructor
  constructor(private upperCaseService: UpperCaseService {
  }

  // invoke the method in the uppercaseService class
  public greet(): void {
    alert(this.upperCaseService.upper("Hello world"));
  }
}
```

The preceding code block imports the `UpperCaseService` class (shown in bold) via an `import` statement and then injects an instance of this class into the constructor of the `UpperComponent` class. Next, the `template` property contains a `<button>` element with a click handler that invokes the `greet()` method defined in the preceding code block. The `greet()` method displays an alert whose contents are the result of invoking the `upper()` method in the custom `UpperCaseService` class.

Built-In Angular Services

Angular supports various built-in services, which are organized in different modules. For example, the `http` module (in `@angular/http`) contains support for HTTP requests that involve typical verbs, such as `GET`, `POST`, `PUT`, and `DELETE`. You saw examples of HTTP-based requests in Chapter 4. The routing module (in `@angular/router`) provides routing support, which includes HTML5 and hash routing. The form module (in `@angular/forms`) provides form-related services. Check the Angular documentation for a complete list of built-in services.

An Angular Service Example

DVD Copy the directory `ServiceExample` from the companion disc into a convenient location. Listing 5.13 displays the contents of `app.component.ts` that contains an example of defining a service.

Listing 5.13: app.component.ts

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

@Injectable()
class Service {
  somedata = ["one", "two", "three"];
  constructor() { }

  getData() { return this.somedata; }
  toString() { return "From toString"; }
}

@Component({
  selector: 'app-root',
  providers: [ Service ],
  template: 'Here is the data: {{ service.getData() }}'
})
export class AppComponent {
  constructor(public service: Service) { }
}
```

Listing 5.13 contains a `Service` class that is preceded by the `@Injectable` decorator, which enables us to inject an instance of the

Service class in the constructor of the AppComponent class in Listing 5.13.

A Service with an EventEmitter

This section contains a code sample that uses EventEmitters for communicating between a component and its child component.

DVD Copy the directory UserServiceEmitter from the companion disc into a convenient location. Listing 5.14 displays the contents of user.component.ts that defines a custom component for an individual user.

Listing 5.14: user.component.ts

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

@Component({
  selector: 'user',
  template: '<h2></h>'
})
export class User {
  fname: string;
  lname: string;
  imageUrl: string;
  constructor(fname:string, lname:string, imageUrl:string) {
     this.fname = fname;
     this.lname = lname;
     this.imageUrl = imageUrl;
  }
}
```

Listing 5.14 is straightforward: the custom User class and a constructor with three arguments that represent the first name, last name, and image URL, respectively, for a user.

Listing 5.15 displays the contents of user.service.ts that creates a list of users, where each user has a first name, last name, and an associated PNG file.

Listing 5.15: user.service.ts

```
import {Component} from '@angular/core';
import {User}      from './user.component';

@Component({
  selector: 'user-comp',
  template: '<h2></h2>'
})
export class UserService {
  userList:User[];

  constructor() {
     this.userList = [
                  new User('Jane','Smith','src/app/sample1.
                                                     png'),
                  new User('John','Stone','src/app/sample2.
                                                     png'),
                  new User('Dave','Jones','src/app/sample3.
                                                     png'),
                ]
  }

  getUserList() {
     return this.userList;
  }
}
```

Listing 5.15 imports the User custom component (displayed in Listing 5.16), and then defines the UserService custom component that uses the userList array to keep track of users. This array is initialized in the constructor, and three new User instances are created and populated with data. The getUserList() method performs the "service" that returns the userList array.

Listing 5.16 displays the contents of app.component.ts that references the two preceding custom components and renders user-related information in an unordered list.

Listing 5.16: app.component.ts

```
import {Component}    from  '@angular/core';
import {EventEmitter} from  '@angular/core';
import {UserService}  from  './user.service';
import {User}         from  './user.component';

@Component({
  selector: 'app-root',
  providers: [User, UserService],
  template: `
     <div class="ui items">
       <user-comp
        *ngFor="let user of userList; let i=index"
          [user]="user"
          (mouseover)='mouseEvent(user)'
          [class.chosen]="isSelected(user)">
          USER {{i+1}}: {{user.fname}}-{{user.lname}}
          <img class="user-image" [src]="user.imageUrl"
               (mouseenter)="mouseEnter(user)"
               height="50">
       </user-comp>
     </div>
    '
})
export class AppComponent {
  user:User;
  currentUser:User;
  userList:User[];
  onUserSelected: EventEmitter<User>;

  mouseEvent(user:User) {
     console.log("current user: "+user.fname+" "+user.lname);
     this.currentUser = user;
     this.onUserSelected.emit(user);
  }

  mouseEnter(user:User) {
     console.log("image name: "+user.imageUrl);
     alert("Image name: "+user.imageUrl);
  }
  isSelected(user: User): boolean {
    if (!user || !this.currentUser) {
      return false;
    }

    return user.lname === this.currentUser.lname;
  //return true;
  }

  constructor(userService:UserService) {
     this.onUserSelected = new EventEmitter();
     this.userList = userService.getUserList();
   }
 }
```

Listing 5.16 contains a `template` property that displays the current list of users (i.e., the three users that are initialized in the constructor in Listing 5.15). Notice the syntax to display information about each user in the list of users:

```
USER {{i+1}}: {{user.fname}}-{{user.lname}}
<img class="user-image" [src]="user.imageUrl"
     (mouseenter)="mouseEnter(user)"
     height="50">
```

When users move their mouse over the displayed list, the `mouseEvent()` method is invoked to set `currentUser` to refer to the current user. In addition, when users move their mouse over one of the images, the `mouseEnter()` method is invoked, which displays a message via `console.log()` and also displays an alert.

Listing 5.17 displays the contents of `app.module.ts` that references the custom component and custom service.

Listing 5.17: app.module.ts

```
import { NgModule }      from '@angular/core';
import {CUSTOM_ELEMENTS_SCHEMA} from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';
import { UserService }   from './user.service';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ UserService ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ],
  schemas:      [CUSTOM_ELEMENTS_SCHEMA]
})
export class AppModule { }
```

Listing 5.17 has essentially the same contents as the example in Chapter 2 that contains the `schemas` property. The lines shown in bold are the modifications that are required for the code sample in this section.

Displaying GitHub User Data

This section shows you how to read a `Github` user name from an input field, search for that user in `Github`, and then append some details about that user in a list.

DVD Copy the directory `GithubUsersForm` from the companion disc into a convenient location. Listing 5.18 displays the contents of `app.component.ts` that illustrates how to make an `HTTP GET` request to retrieve information about `Github` users.

Listing 5.18: app.component.ts

```
import { Component }     from '@angular/core';
import { Inject }        from '@angular/core';
import { Http }          from '@angular/http';
import { UserComponent } from './user.component';

@Component({
  selector: 'app-root',
  template: '
    <div>
      <form>
        <h3>Search Github For User:</h3>
        <div class="field">
          <label for="guser">Github Id</label>
          <input type="text" #guser>
        </div>

        <button (click)="findGithubUser(guser)">
          >>> Find Github User <<<
        </button>
      </form>

      <div id="container">
       <div class="onerow">
        <h3>List of Users:</h3>
        <ul>
          <li *ngFor="let user of users"
              (mouseover)="currUser(user)">
            {{user.field1}} {{user.field2}}</li>
         </ul>
        </div>
       </div>
      </div>
   '
 })
 export class AppComponent {
   currentUser:UserComponent = new UserComponent('ABC',
                                                    'DEF', '');
   users: UserComponent[];
   githubUserInfo:String = "";
   githubUserJSON:JSON;
   user:UserComponent;
   userStr:String = "";
   guserStr:String = "";

   constructor(@Inject(Http) public http:Http) {
     this.users = [
       new UserComponent('Jane', 'Smith', ''),
       new UserComponent('John', 'Stone', ''),
     ];
   }

   currUser(user) {
     console.log("fname: "+user.field1+" lname: "+user.field2);
     this.currentUser = new UserComponent(user.field1,
                                          user.field2,
                                          user.field3);
   }

   findGithubUser(guser: HTMLInputElement): boolean {
      if((guser.value == undefined) || (guser.value == "")) {
         alert("Please enter a user name");
         return;
      }

      // guser.value is not available in the 'subscribe' method
      this.guserStr = guser.value;

          this.http.get('https://api.github.com/
      users/'+guser.value)

          .map(res => res.json())
          .subscribe(data => {
                 //console.log("user = "+JSON.stringify(data));
                 this.githubUserInfo = data;
                 this.user = new UserComponent(data.name,
                                             this.guserStr,
                                             data.created_at);
                  this.users.push(this.user); },
          err => {
             console.log("Lookup error: "+err);
             alert("Lookup error: "+err);
          }
        );

     // reset the input field to an empty string
     guser.value = "";

     // prevent a page reload:
     return false;
   }
 }
```

Listing 5.18 contains the usual `import` statements, followed by an `@Component` decorator that contains the usual selector property and an extensive template property.

The template property consists of a top-level `<div>` element that contains a `<form>` element and another `<div>` element. The `<form>` element contains an `<input>` element where users can enter a `Github` user name, whereas the `<div>` element contains a `<ul>` element that in turn renders the list of current users. Notice that each `<li>` element in the `<ul>` element handles a `mouseover` event by setting the current user to the element that users have highlighted with their mouse.

The next portion of Listing 5.18 is the definition of the exported class `AppComponent`, which initializes some instance variables, followed by a constructor that initializes the `users` array with two hard-coded users. Next, the `currUser()` method "points" to the user that users have highlighted with their mouse. This functionality is not essential, but it's available if you need to keep track of the current highlighted user.

The `findGithubUser()` method displays an alert if the `<input>` element is empty (which prevents a redundant invocation of the `http()` method). If a user is entered in the `<input>` element, the code invokes an `HTTP GET` request from the `Github` website and appends the new user (as an instance of the `UserComponent` class) to the users array. In addition, an alert is displayed if there is no `Github` that matches the input string.

Another small but important detail is the following code snippet, which keeps track of the user-specified input string:

```
this.guserStr = guser.value;
```

The preceding snippet is required because of the context change that occurs inside the invocation of the `get()` method, which loses the reference to the `guser` argument.

Listing 5.19 displays the contents of `user.component.ts` that contains three strings for keeping track of three user-related fields.

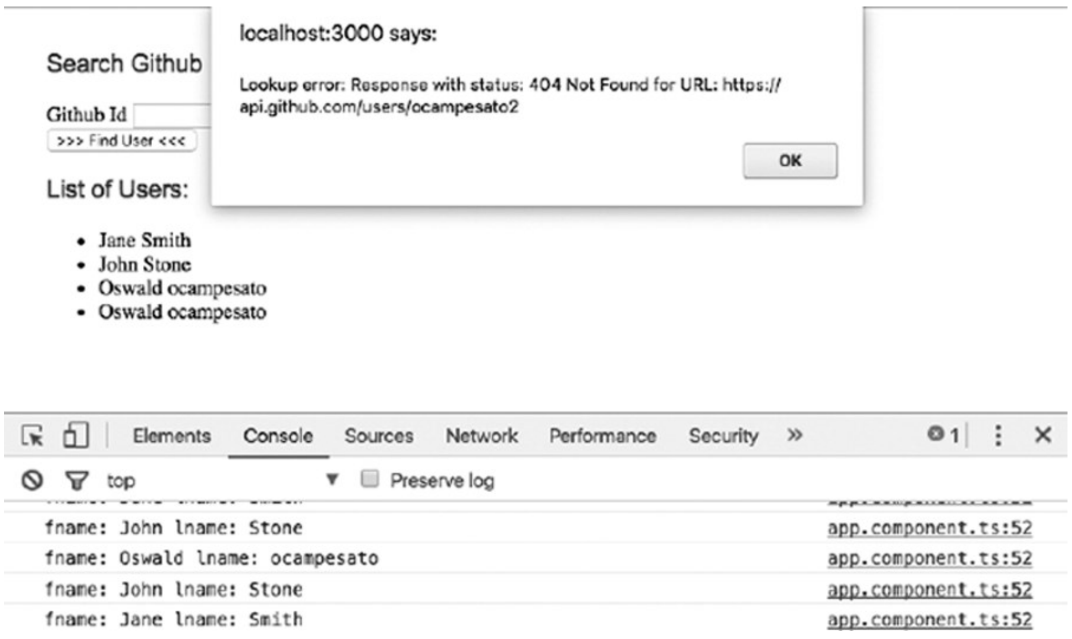
Listing 5.19: user.component.ts

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

@Component({
  selector: 'current-user',
  template: '<h1></h1>'
})
export class UserComponent {
  field1:string = "";
  field2:string = "";
  field3:string = "";

  constructor(field1:string, field2:string, field3:string) {
     this.field1 = field1;
     this.field2 = field2;
     this.field3 = field3;
  }
}
```

Listing 5.19 contains the string properties `field1`, `field2`, and `field3` for keeping track of three attributes from the `JSON`-based string of information for a `Github` user. The property names in the `UserComponent` class are generic so that you can store different properties from the `JSON` string, such as `followers`, `following`, and `created_at`.

You now have a starting point for displaying additional details regarding a user, and you can improve the display by using Bootstrap or some other toolkit for UI-related layouts.

Figure 5.3 displays the output from launching this Angular application and adding information about `Github` users. One thing to notice is that duplicates are allowed in the current sample (the code for preventing duplicates is an exercise for you).



localhost:3000 says:

Lookup error: Response with status: 404 Not Found for URL: https://api.github.com/users/ocampesato2

List of Users:

- Jane Smith
- John Stone
- Oswald ocampesato
- Oswald ocampesato

```
fname: John lname: Stone                                app.component.ts:52
fname: Oswald lname: ocampesato                         app.component.ts:52
fname: John lname: Stone                                app.component.ts:52
fname: Jane lname: Smith                                app.component.ts:52
```

Figure 5.3: Search and display GitHub users in a list.

Other Service-Related Use Cases

As you saw in the previous section, services are useful for retrieving external data. In addition, there are other situations that involve sharing data and services in an Angular application. In particular, one application might need multiple instances of a service class, whereas another application might need to enforce a single instance of a service class. Yet another situation involves sharing data between components in an Angular application.

These three scenarios are discussed briefly in the following subsections, and they are based on a very simple `UserService` class that is defined as follows:

```
export class UserService {
   private users: string[];

   adduser(user: string) {
      this.users.push(user);
   }

   getUsers() {

      return this.users;
   }
}
```

Multiple Service Instances

Suppose that `UserService`, `MyComponent1`, and `MyComponent2` are defined in the TypeScript files `user.service.ts`, `component1.ts`, and `component2.ts`, respectively. If you need a different instance of the `UserService` class in each component, inject this class in their constructors, as shown here:

```
// component1.ts
export class MyComponent1 {
  constructor(private userService: UserService) {
  }
}

// component2.ts
export class MyComponent2 {
  constructor(private userService: UserService) {
  }
}
```

In the preceding code, the instance of the `UserService` class in `MyComponent1` is different from the instance of the `UserService` class in `MyComponent2`.

Single Service Instance

Consider the situation in which two Angular components must share the same instance of the `UserService` class. For simplicity, let's assume that the two components are children of the root component. In this scenario, perform the following sequence of steps:

Create a new service component (`ng g s service).`

Include `UserService` in the `providers` array in `app.module.ts.`

Import `MyComponent1` and `MyComponent2` in service.component.ts.

Remove the `UserService` class from the `providers` array in `MyComponent1.`

Remove the `UserService` class from the `providers` array in `MyComponent2.`

Step 2 ensures that the `UserService` class is available to *all* components in this Angular application, and there is only one instance of the `UserService` class throughout the application.

Services and Intercomponent Communication

There are three steps required to send a new user from `MyComponent1` to `MyComponent2`.

Step 1: Define a variable `sendUser` that is an instance of `EventEmitter` and a `sendNewUser()` method in `UserService`:

```
export class UserService {
   sendUser = new EventEmitter<string>();
   ...
   sendNewUser(user:string) {
      this.sendUser.emit(user);
   }
}
```

Step 2: Define an `onSend()` method in `MyComponent1` to send a new user to `MyComponent2`:

```
onSend(user:string) {
   this.userService.sendNewUser(user);
}
```

Step 3: Define an `Observable` in `MyComponent2` to "listen" for data that is emitted from `MyComponent1`:

```
ngOnInit() {
  this.userService.subscribe(...);
}
```

Another way to summarize the logical flow in the preceding code blocks is shown here:

- Users click a button to add a new user.
- The `UserService` instance sends the data to `Component1.`
- The `Component1` instance "emits" the new user.
- The `Component2` instance "listens" for the new user via an `Observable.`

Injecting Services into Services

You have seen how to use DI to inject a service into a component via its constructor. You can also inject services into other services. To do so, use the `@Injectable` decorator in the "injected service":

```
@Injectable
@Component({
})
export MyService(...)
```

Note DI in Angular only works in classes that have a suitable decorator as part of the class definition.

Summary

This chapter showed you how to create Angular applications with HTML5 `Forms` as well as `Forms` that contain Angular `Controls` and `FormGroups`. You also saw how to save form-based data in local storage. Next you learned about Angular `Pipes`, along with an example that showed you how to implement this functionality.

You also learned about Angular `Services`, and saw an example that illustrated how to use `Services`. Finally, you learned how use the `http()` method (which returns an `Observable`) of the `Http` class to retrieve data for any `Github` user and display portions of that data in a list of users.