EXPERT INSIGHT

Angular Cookbook

Over 80 actionable recipes for curious Angular developers

Second Edition

Muhammad Ahsan Ayaz

‹packt›

EXPERT INSIGHT

Angular Cookbook

Over 80 actionable recipes for curious Angular developers

Second Edition

Muhammad Ahsan Ayaz

‹packt›

Angular Cookbook

Early Access Publication: Angular Cookbook

Early Access Production Reference: B18469

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80323-344-4

www.packt.com

Table of Contents

Angular Cookbook, Second Edition: Over 80 actionable recipes for curious Angular developers

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1 Winning Components Communication

Join our book community on Discord

https://packt.link/EarlyAccess

In this chapter, you'll master component communication in Angular. You'll learn different techniques to establish communication between components and will learn which technique is suitable in which situation. You'll also learn how to create a dynamic Angular component in this chapter.The following are the recipes we're going to cover in this chapter:

- Components communication using component `@Input(s)` and `@Output(s)`
- Components communication using services
- Using setters for intercepting input property changes
- Using `ngOnChanges` to intercept input property changes
- Accessing a child component in a parent template via template variables
- Accessing a child component in a parent component class using `ViewChild`
- Standalone Components & passing data through route params

Technical requirements

For the recipes in this chapter, make sure you have Git and Node.js installed on your machine. See the Prerequisite page (PAGE NUMBER) for the instructions to clone the repository and setting it up. The starter code for this chapter can be found at https://github.com/AhsanAyaz/ng-cookbook-second-edition/tree/main/start/apps/chapter01.

Components communication using component @Input(s) and @Output(s)

You'll start with an app with a parent component and two child components. You'll then use Angular `@Input` and `@Ouput` decorators to establish communication between them using attributes and `EventEmitter` (s).

Getting ready

The app that we are going to work with resides in `start/apps/chapter01/cc-inputs-outputs` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-inputs-outputs` to serve the project

This should open the app in a new browser tab and you should see the following:

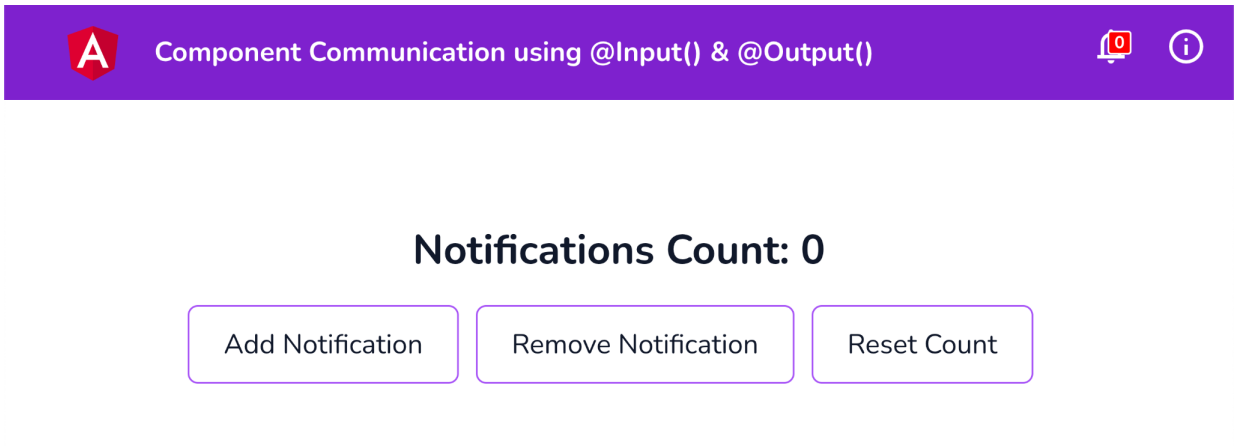


Figure 1.1 – The cc-inputs-outputs app running on http://localhost:4200

How to do it...

So far, we have an app with `AppComponent`, `NotificationsButtonComponent`, and `NotificationsManagerComponent`. While `AppComponent` is the parent of the other two components mentioned, there is absolutely no component communication between them to sync the notification count value among both components. Let's establish the appropriate communication between them using the following steps:

1. We'll move the `notificationsCount` variable from `NotificationsManagerComponent` to the `AppComponent`. To do so, create a `notificationsCount` property in `app.component.ts` as follows:

```
export class AppComponent {
  notificationsCount = 0;
}
```

1. Next, convert the `notificationsCount` property in `notifications-manager.component.ts` to `@Input()`, rename it to `count`, and replace its usages as follows:

```
import { Component, OnInit, Input } from '@angular/core';
@Component({
  selector: 'app-notifications-manager',
  templateUrl: './notifications-manager.component.html',
  styleUrls: ['./notifications-manager.component.scss']
})
export class NotificationsManagerComponent implements OnInit {
  @Input() count = 0
  constructor() { }
  ngOnInit(): void {
  }
  addNotification() {
    this.count++;
  }
  removeNotification() {
    if (this.count == 0) {
      return;
    }
    this.count--;
  }
  resetCount() {
    this.count = 0;
  }
}
```

1. Update `notifications-manager.component.html` to use `count` instead of `notificationsCount`:

```
<div class="notif-manager">
  <div class="notif-manager__count">
    Notifications Count: {{count}}
  </div>
  ...
</div>
```

1. Next, pass the `notificationsCount` property from `app.component.html` to the `<app-notifications-manager>` element as an input:

```
<div class="content" role="main">
  <app-notifications-manager
    [count]="notificationsCount">
  </app-notifications-manager>
</div>
```

1. You could now test whether the value is being correctly passed from `app.component.html` to `app-notifications-manager` by assigning the value of `notificationsCount` in `app.component.ts` as `10`. You'll see that in `NotificationsManagerComponent`, the initial value shown will be `10`:

```
export class AppComponent {
  notificationsCount = 10;
}
```

1. Now, create an `@Input()` in `notifications-button.component.` ts named `count` as well:

```
import { Component, OnInit, Input } from '@angular/core';
...
export class NotificationsButtonComponent implements OnInit {
  @Input() count = 0;
  ...
}
```

1. Pass `notificationsCount` to `<app-notifications-button>` as well from `app.component.html`:

```
<!-- Toolbar -->
<div class="toolbar" role="banner">
  ...
  <span>@Component Inputs and Outputs</span>
  <div class="spacer"></div>
  <div class="notif-bell">
    <app-notifications-button     [count]="notificationsCount">
    </app-notifications-button>
  </div>
</div>
...
```

1. Use the `count` input in `notifications-button.component.html` with the notification bell icon:

```
<div class="bell">
  <i class="material-icons">notifications</i>
  <div class="bell__count">
    <div class="bell__count__digits">
      {{count}}
    </div>
  </div>
</div>
```

You should now see the value `10` for the notification bell icon count as well.*Right now, if you change the count by adding/removing notifications from* `NotificationsManagerComponent`, *the count on the notification bell icon won't change.*

1. To communicate the change from `NotificationsManagerComponent` to `NotificationsButtonComponent`, we'll use Angular `@Output` (s) now. Use `@Ouput` and `@EventEmitter` from `'@angular/core'` inside `notifications-manager.component.ts`:

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
...
export class NotificationsManagerComponent implements OnInit {
  @Input() count = 0
  @Output() countChanged = new EventEmitter<number>();
  ...
  addNotification() {
    this.count++;
    this.countChanged.emit(this.count);
  }
  removeNotification() {
    ...
    this.count--;
    this.countChanged.emit(this.count);
  }
  resetCount() {
    this.count = 0;
    this.countChanged.emit(this.count);
  }
}
```

1. Then, we'll listen in `app.component.html` for the previously emitted event from `NotificationsManagerComponent` and update the `notificationsCount` property accordingly:

```
<div class="content" role="main">
  <app-notifications-manager   (countChanged)="updateNotificationsCount($event)"   [count]="notif
</div>
```

- Since we've listened to the `countChanged` event previously and called the `updateNotificationsCount` method, we need to create this method in `app.component.ts` and update the value of the `notificationsCount` property accordingly:

```
export class AppComponent {
  notificationsCount = 10;
  updateNotificationsCount(count: number) {
    this.notificationsCount = count;
  }
}
```

How it works...

In order to communicate between components using `@Input` (s) and `@Output` (s), the data flow will always go `from` the *child components* `to` the *parent component*, which can provide the new (updated) value *as input* back to the required child components. So, `NotificationsManagerComponent` emits the `countChanged` event. `AppComponent` (being the parent component) listens for the event and updates the value of `notificationsCount`, which automatically updates the `count` property in `NotificationsButtonComponent` because `notificationsCount` is being passed as the `@Input()` count to `NotificationsButtonComponent`. *Figure 1.2* shows the entire process:

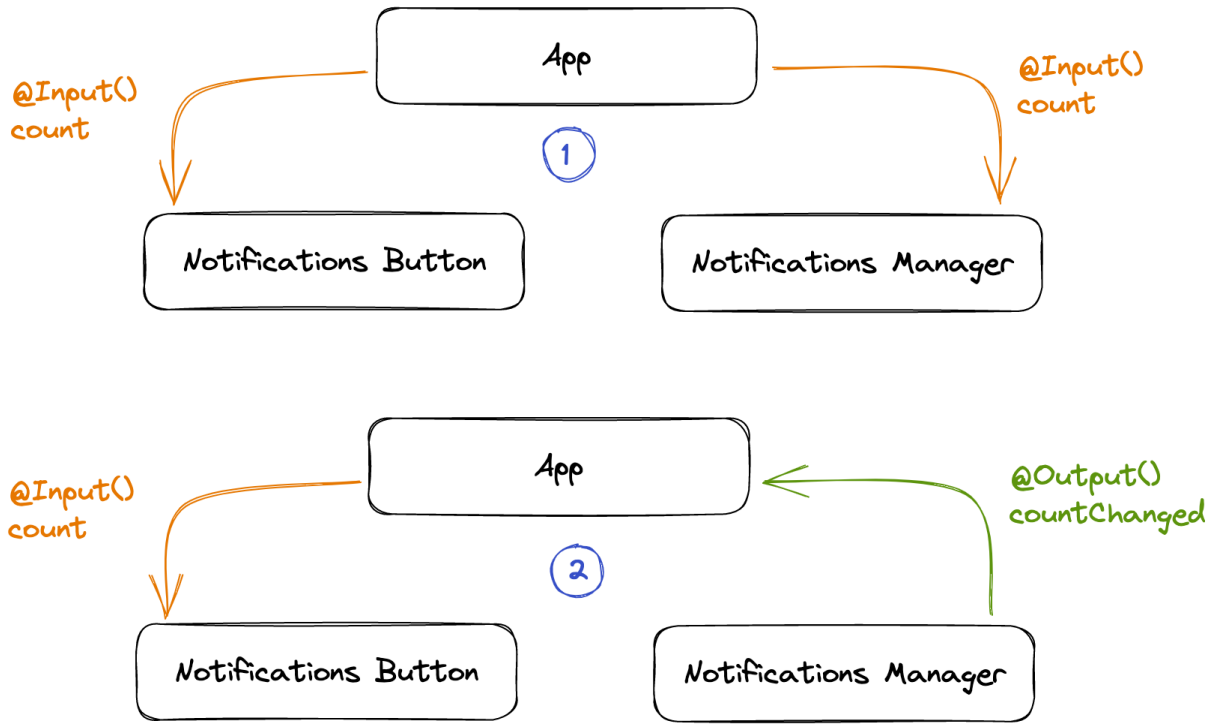


Figure 1.2 – How component communication works with inputs and outputs

See also

- How do Angular components communicate? https://www.thirdrocktechkno.com/blog/how-angular-components-communicate
- *Component Communication in Angular* by Dhananjay Kumar: https://www.youtube.com/watch?v=I8Z8g9APaDY

Components communication using services

In this recipe, you'll start with an app with a parent component and a child component. You'll then use an Angular service to establish communication between them. We're going to use `BehaviorSubject` and `Observable` streams to communicate between components and the service.

Getting ready

The project for this recipe resides in `start/apps/chapter01/cc-services`:

1. Open the project in Visual Studio Code.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-services` to serve the project

This should open the app in a new browser tab and you should see the app as follows:

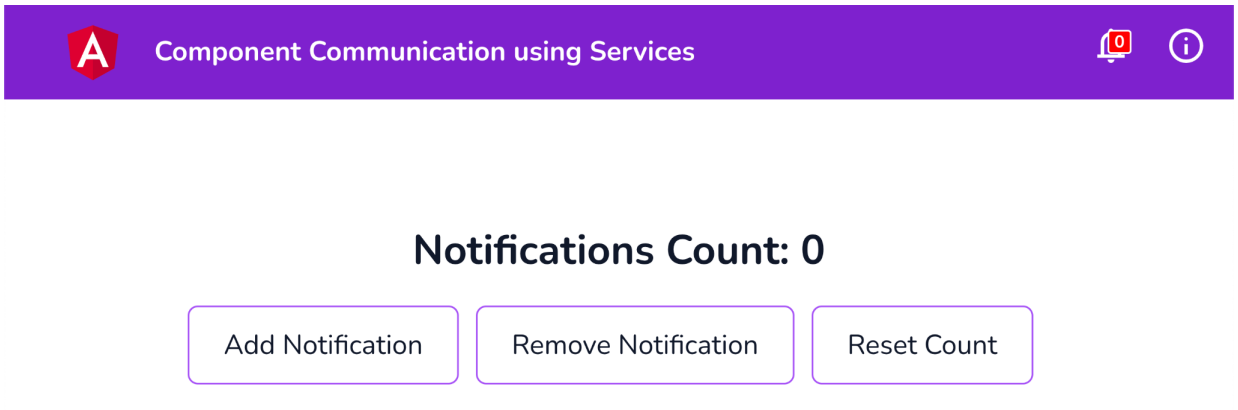


Figure 1.3 – The cc-services app running on http://localhost:4200

How to do it...

Similar to the previous recipe, we have an app with `AppComponent`, `NotificationsButtonComponent`, and `NotificationsManagerComponent`. `AppComponent` is the parent of the other two components mentioned previously, and we need to establish the appropriate communication between them using the following steps:

1. From the terminal, navigate into the `start` folder and create a new service called `NotificationService` by running the following command :

```
nx g s services/Notifications --project cc-services
```

1. Create a `BehaviorSubject` named `count$` inside `notifications.service.ts` and initialize it with `0` (a `BehaviorSubject` requires an initial value):

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class NotificationsService {
  count$ = new BehaviorSubject(0);
  constructor() { }
}
```

1. Rename the `notificationsCount` property in `notifications-manager.component.ts` to `notificationsCount$` and assign the services's `count$` property to it as follows:

```
import { Component, inject } from '@angular/core';
import { NotificationsService } from '../services/notifications.service';
...
export class NotificationsManagerComponent implements OnInit {
  notificationsCount$ = inject(NotificationsService).count$;
  ...
}
```

1. Modify the functions with the `NotificationsManagerComponent` to update the Behavior Subject's value as follows:

```
...
export class NotificationsManagerComponent implements OnInit {
  ...
  addNotification() {
    const currentValue = this.notificationsCount$.getValue();
    this.notificationsCount$.next(currentValue + 1);
```

```
  }
  removeNotification() {
    const currentValue = this.notificationsCount$.getValue();
    if (currentValue === 0) {
      return;
    }
    this.notificationsCount$.next(currentValue - 1);
  }
  resetCount() {
    this.notificationsCount$.next(0);
  }
}
```

1. Use the `notificationsCount$` Observable in `notifications-manager.component.html` with the `async` pipe to show its value:

```
<div class="notif-manager">
  <div class="notif-manager__count">
    Notifications Count: {{notificationsCount$ | async}}
  </div>
  ...
</div>
```

1. Now, similarly inject `NotificationsService` in `notifications-button.component.ts`, create an Observable named `notificationsCount$` inside `NotificationsButtonComponent`, and assign the service's `count$` Observable to it:

```
import { Component, inject } from '@angular/core';
import { NotificationsService } from '../services/notifications.service';
...
export class NotificationsButtonComponent {
  notificationsCount$ = inject(NotificationsService).count$;
}
```

1. Use the `notificationsCount$` Observable in `notifications-button.component.html` with the `async` pipe:

```
<div class="bell">
  <i class="material-icons">notifications</i>
  <div class="bell__count">
    <div class="bell__count__digits">
      {{notificationsCount$ | async}}
    </div>
  </div>
</div>
```

If you refresh the app now, you should be able to see the value `0` for both the notifications manager component and the notifications button component.

1. Change the initial value for the `count BehaviorSubject` to `10` and see whether that reflects in both components:

```
...
export class NotificationsService {
  private count: BehaviorSubject<number> = new   BehaviorSubject<number>(10);
  ...
}
```

How it works...

`BehaviorSubject` is a special type of `Observable` that requires an initial value and can be used by many subscribers. In this recipe, we create a `BehaviorSubject` to store the value of the notifications count. Once we have created the BehaviorSubject named `count$`, we inject `NotificationsService` in our components using the (fairly new) `inject method` and assign the `count$` property of the service to a property of the components. This allows us to work with the `BehaviorSubject` in both the in

`NotificationsButtonComponent` and the in `NotificationsManagerComponent`. Then we use the `notificationsCount$` property in the templates of both above mentioned functions to be able to render the count value. Notice that we use the `async` pipe in the templates. This help Angular to let the template subscribe to the BehaviorSubject when the component is rendered, and to unsubscribe automatically when the component is destroyed.To update the value of the `BehaviorSubject`, we use its `.next()` method by providing the new value to be set. As soon as the value of count$ is updated, the components rerender the new value. Thanks to RxJS and Angular's change detection.

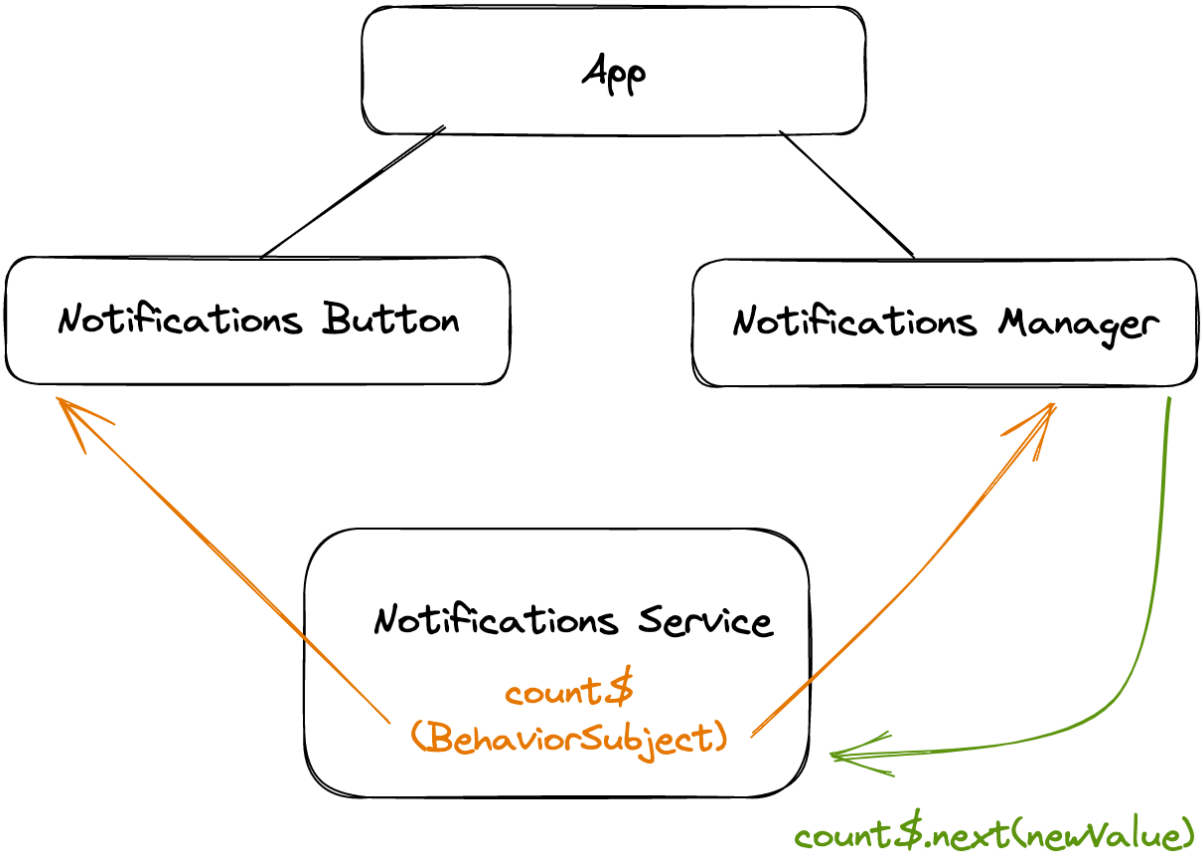


Figure 1.4 – How component communication works using an Angular Service

See also

- Subjects from RxJS's official documentation: https://www.learnrxjs.io/learn-rxjs/subjects
- `BehaviorSubject` versus `Observable` on Stack Overflow: https://stackoverflow.com/a/40231605

Using setters for intercepting input property changes

In this recipe, you will learn about how to intercept changes in an `@Input` passed from a parent component and to perform some action on this event. We'll intercept the `vName` input passed from the `VersionControlComponent` parent component to the `VcLogsComponent` child component. We'll use `setters` to generate a log whenever the value of `vName` changes and will show those logs in the child component.

Getting ready

The project for this recipe resides in `start/apps/chapter01/cc-setters`:

1. Open the project in Visual Studio Code.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-setters` to serve the project

This should open the app in a new browser tab and you should see the app as follows:

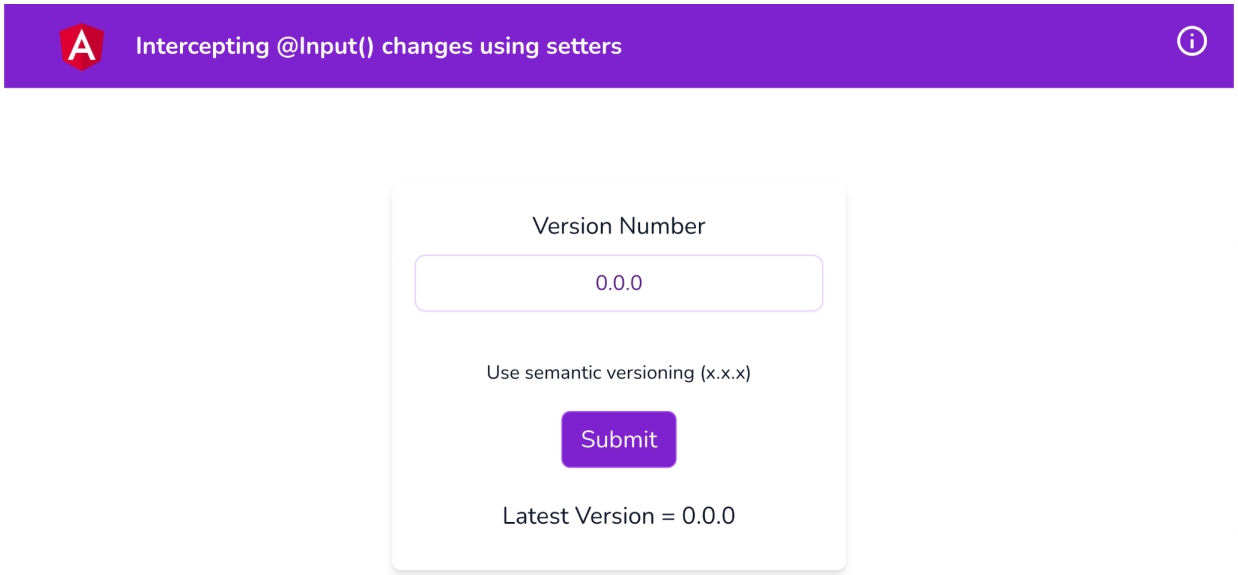


Figure 1.5 – The cc-setters app running on http://localhost:4200

How to do it...

1. We'll first create a logs array in `VcLogsComponent` as follows to store all the logs that we'll display later using our template:

```
export class VcLogsComponent implements OnInit {
  @Input() vName;
  logs: string[] = [];
}
```

1. Let's modifly the HTML template so we can show the logs. Modify the `vc-logs.component.html` file as follows:

```
<h5>Latest Version = {{vName}}</h5>
<div class="logs">
  <div class="logs__item" *ngFor="let log of logs">
    {{log}}
  </div>
</div>
```

The following screenshot shows the app with the logs container:

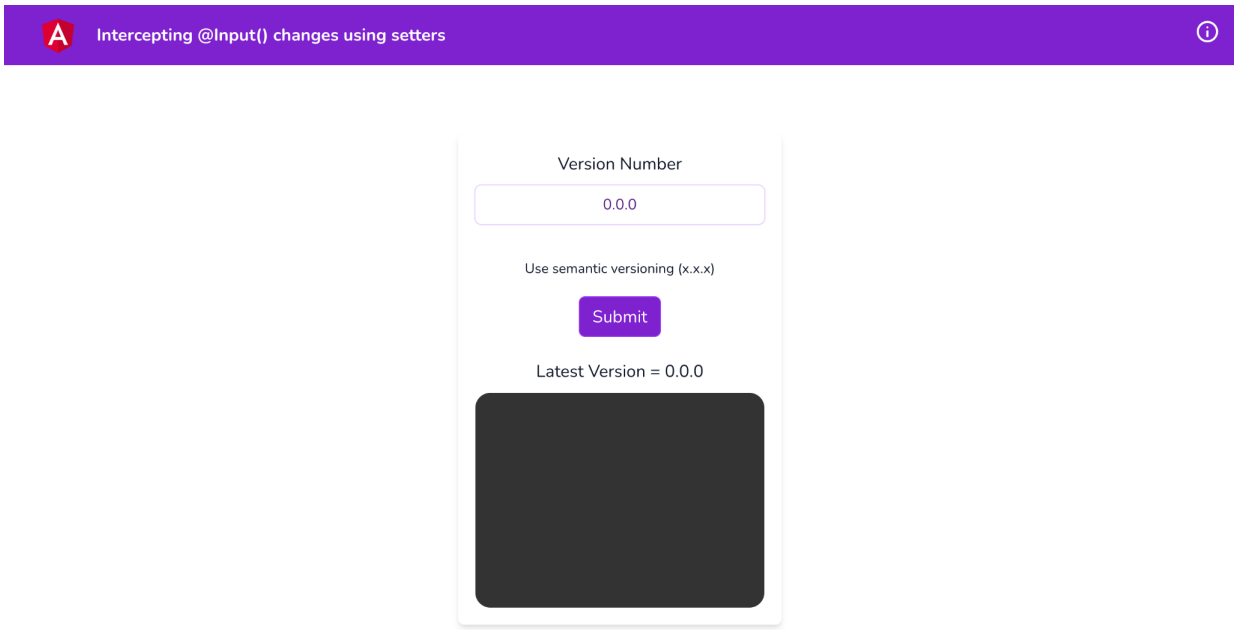


Figure 1.6 – The cc-setters app with logs container

1. Now, we'll convert the `@Input()` in `vc-logs.component.ts` to use a `getter` and `setter` so we can intercept the input changes. For that, we'll also create an internal property named `_vName`. The code should look as follows:

```
import { Component, Input } from '@angular/core'
...
export class VcLogsComponent implements OnInit {
  @Input()
  get vName() {
    return this._vName;
  }
  set vName(name: string) {
    this._vName = name;
  }
  logs: string[] = [];
  _vName!: string;
...
}
```

1. Modify the `setter` to create some logs now. We'll push a new log to the logs array whenever the value of `vName` changes. For the first time, we'll push a log saying `'initial version is x.x.x'`:

```
export class VcLogsComponent implements OnInit {
  ...
  set vName(name: string) {
    if (!this._vName) {
      this.logs.push(`initial version is ${name.trim()}`)
    }
    this._vName = name;
  }
...
}
```

1. Now, for every time we change the version name, we need to show a different message saying `'version changed to x.x.x'`. *Figure 1.6* shows the final output. For the required changes, let's modify the `vName` setter as follows:

```
export class VcLogsComponent implements OnInit {
  ...
  set vName(name: string) {
    if (!name) return;
    if (!this._vName) {
      this.logs.push(`initial version is ${name.trim()}`)
    } else {
      this.logs.push(`version changed to ${name.trim()}`)
    }
    this._vName = name;
  }
```

The following screenshot shows the final output:

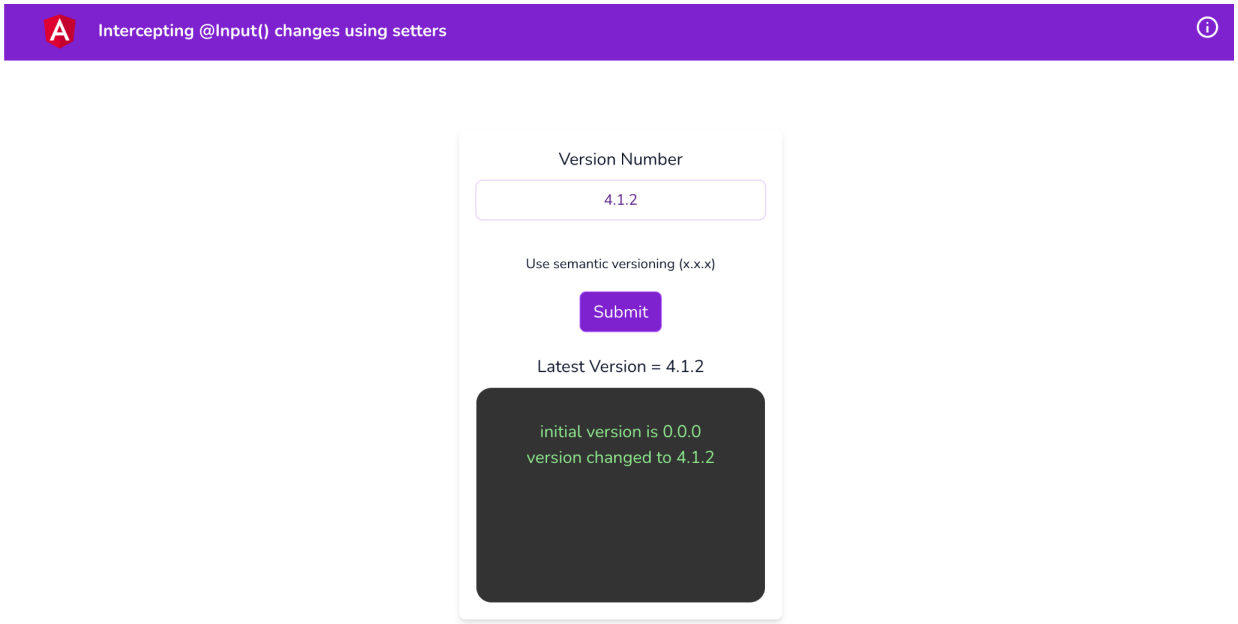


Figure 1.7 – Final output using the setter

How it works...

JavaScript has `getters` as functions that return a dynamically computed value. It also has `setters` as functions that execute some logic when the targeted property changes. Angular uses TypeScript, which is a super of javascript and Angular's `@Input()` properties can also use `getters` and `setters` since they're basically a property of the provided class.

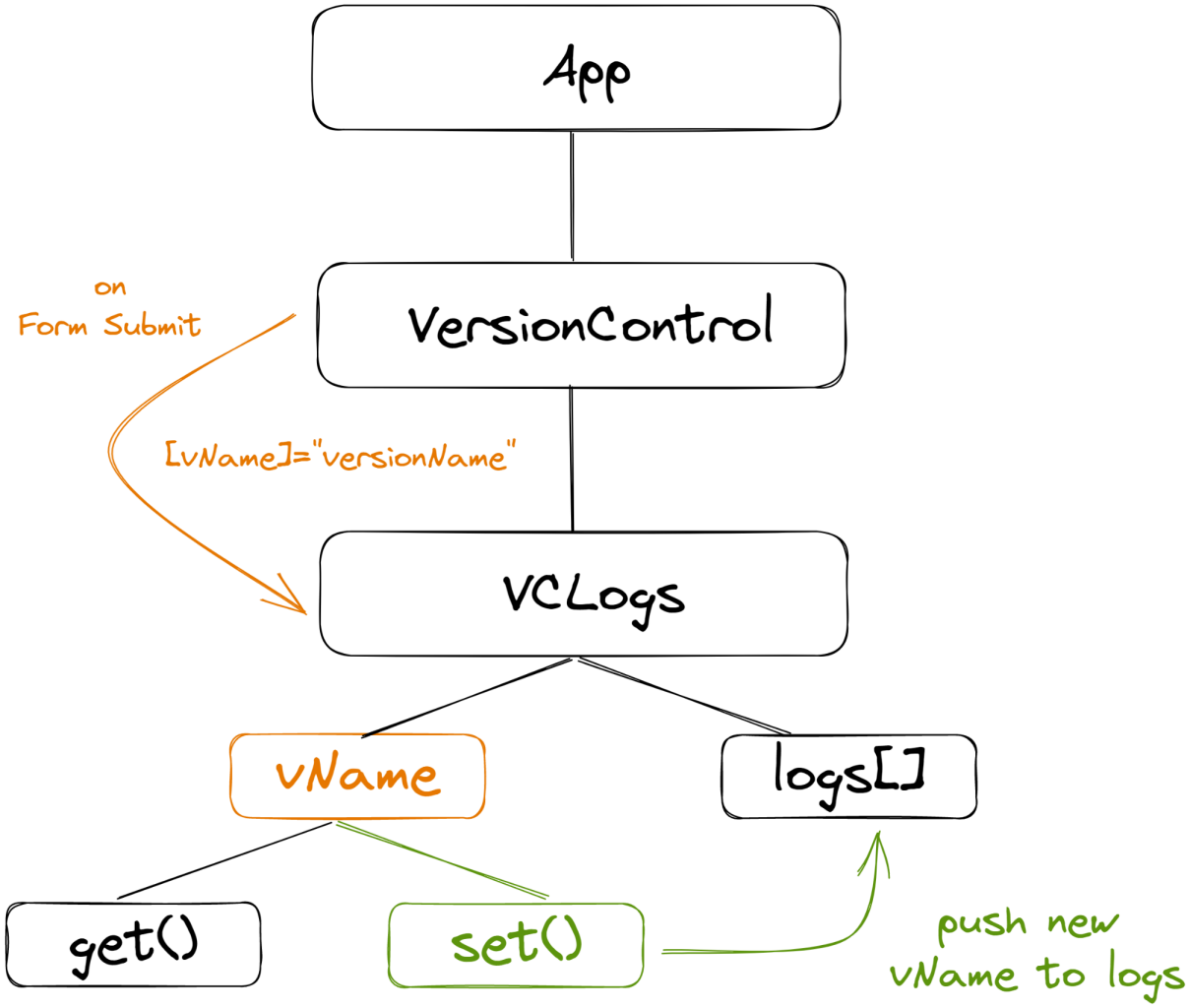


Figure 1.8 – Diagram explaining the flow of data in cc-setters app

For this recipe, we use a getter a setter for our input named `vName` so whenever the input changes, we use the setter method to push the new version to the logs list. Then we use the `logs` array in the template to render the list of logs on the view.It is always a good idea to use a private variable/property along with the property using getters and setters. This is so that we can modify the private property in our component and the template only accesses the public property using the getter.

See also

- https://angular.io/guide/component-interaction#intercept-input-property-changes-with-a-setter
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set

Using ngOnChanges to intercept input property changes

In this recipe, you'll learn how to use `ngOnChanges` to intercept changes using the `SimpleChanges` API. We'll listen to a `vName` input passed from the `VersionControlComponent` parent component to the `VcLogsComponent` child component.

Getting ready

The app that we are going to work with resides in `start/apps/chapter01/cc-ng-on-changes` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-ng-on-changes` to serve the project

This should open the app in a new browser tab and you should see the following:

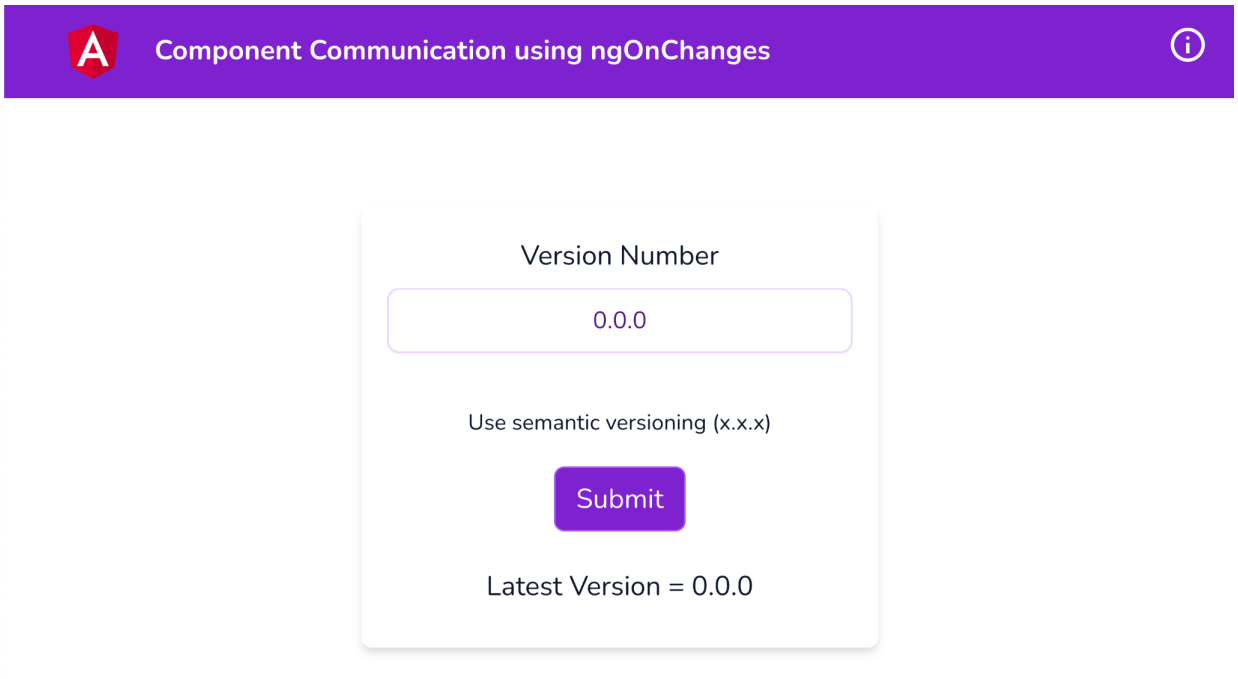


Figure 1.9 – The cc-ng-on-changes app running on http://localhost:4200

How to do it...

1. We'll first create a logs array in `VcLogsComponent` as follows to store all the logs that we'll display later using our template:

```
export class VcLogsComponent implements OnInit {
  @Input() vName;
  logs: string[] = [];
...
}
```

1. Let's create the HTML for where we'll show the logs. Let's add the logs container and log items using the following code to `vc-logs.component.html` :

```
<h5>Latest Version = {{vName}}</h5>
<div class="logs">
  <div class="logs__item" *ngFor="let log of logs">
    {{log}}
  </div>
</div>
```

The following screenshot shows the app with the logs container:

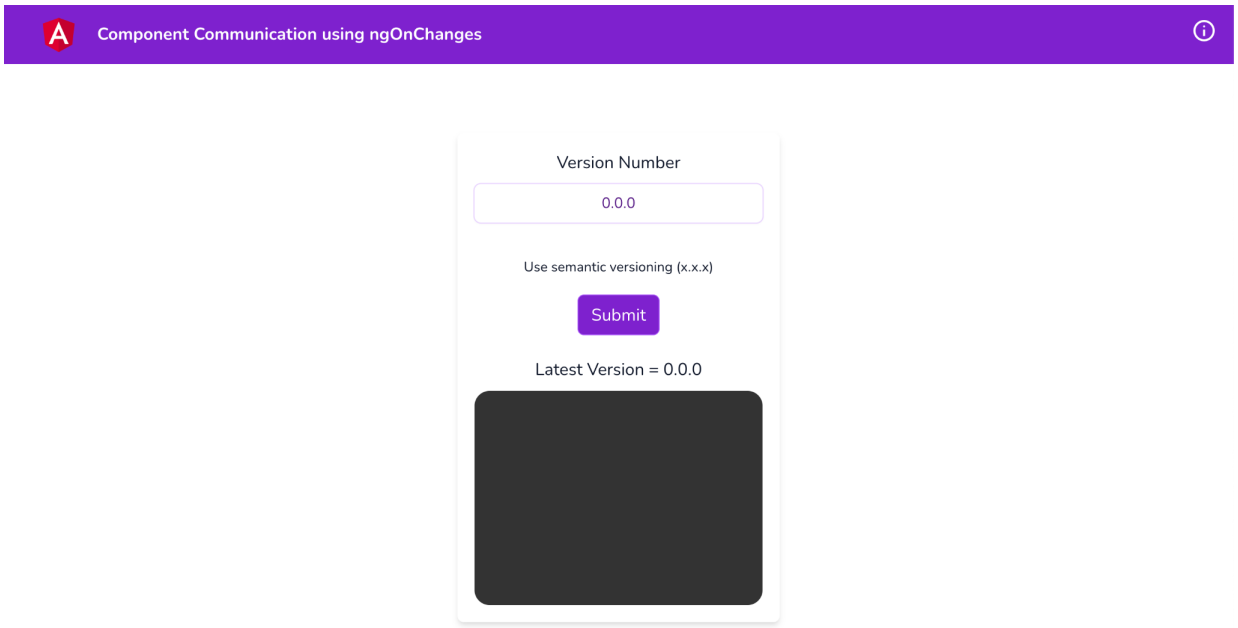


Figure 1.10 – The cc-ng-on-changes app with logs container

1. Now, let's implement `ngOnChanges` using simple changes in `VcLogsComponent` as follows in the `vc-logs.component.ts` file:

```
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';
...
export class VcLogsComponent implements OnChanges {
  @Input() vName;
  logs: string[] = [];
  ngOnChanges(changes: SimpleChanges) {
  }
}
```

1. We now can add a log for the initial value of the `vName` input saying `'initial version is x.x.x'`. We do this by checking whether it is the initial value using the `.isFirstChange()` method as follows:

```
...
export class VcLogsComponent implements OnChanges {
  ...
  ngOnChanges(changes: SimpleChanges) {
    const { currentValue } = changes['vName'];
    if (changes['vName'].isFirstChange()) {
      this.logs.push(`initial version is ${currentValue.trim()}`);
    }
  }
}
```

1. Let's handle the case where we update the version after the initial value was assigned. For that, we'll add another log that says `'version changed to x.x.x'` using an `else` condition, as follows:

```
...
export class VcLogsComponent implements OnInit, OnChanges {
  ...
  ngOnChanges(changes: SimpleChanges) {
    const { currentValue } = changes['vName'];
    if (changes['vName'].isFirstChange()) {
      this.logs.push(`initial version is ${currentValue.trim()}`);
    }  else {
      this.logs.push(`version changed to ${currentValue.trim()}`)
    }
```

```
    }
}
```

The following screenshot shows the final output:

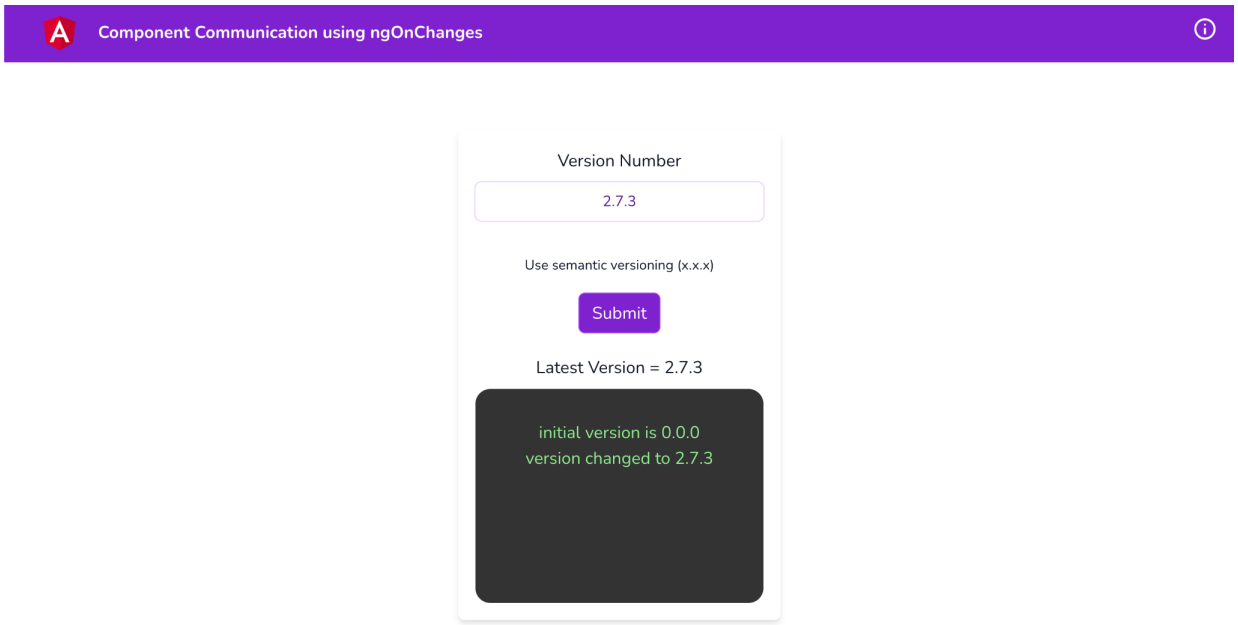


Figure 1.7 – Final output using ngOnChanges

How it works...

`ngOnChanges` is one of the many `Lifecycle Hooks` which Angular provides out of the box. It triggers even before the `ngOnInit` hook. So, you get the *initial values* in the first call and the *updated values* later on. Whenever any of the inputs change, the `ngOnChanges` callback is triggered with `SimpleChanges`. And in the changes, for each `@Input()` you can get the previous value, the current value, and a *Boolean* representing whether this is the first change to the input (that is, the initial value). When we update the value of the `vName` input in the parent, `ngOnChanges` gets called with the updated value. Then, based on the situation, we add an appropriate log into our `logs` array and display it on the UI.

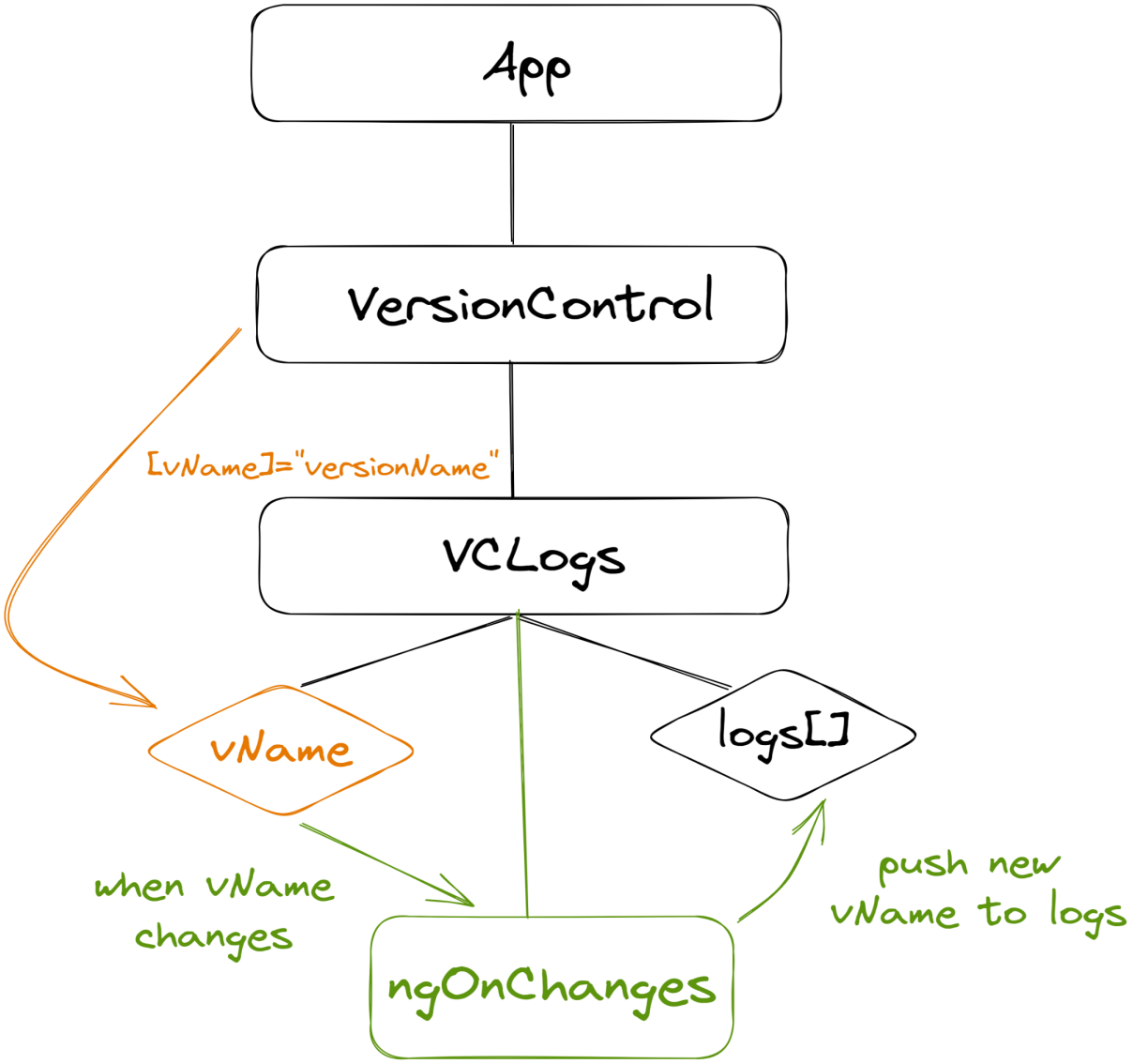


Figure 1.11 – How ngOnChanges pushes the new version to logs array

See also

- Angular Lifecycle Hooks: https://angular.io/guide/lifecycle-hooks
- Using change detection hooks with `ngOnChanges` : https://angular.io/guide/lifecycle-hooks#using-change-detection-hooks
- `SimpleChanges` API reference: https://angular.io/api/core/SimpleChanges

Accessing a child component in the parent template via template variables

In this recipe, you'll learn how to use **Angular template reference variables** to access a child component into a parent component's template. You'll start with an app having `AppComponent` as the parent component and `GalleryComponent` as the child component. You'll then create a template variable for the child component in the parent's template to access it and perform some actions in the component class.

Getting ready

The app that we are going to work with resides in `start/apps/chapter01/cc-template-vars` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-template-vars` to serve the project

This should open the app in a new browser tab and you should see the following:

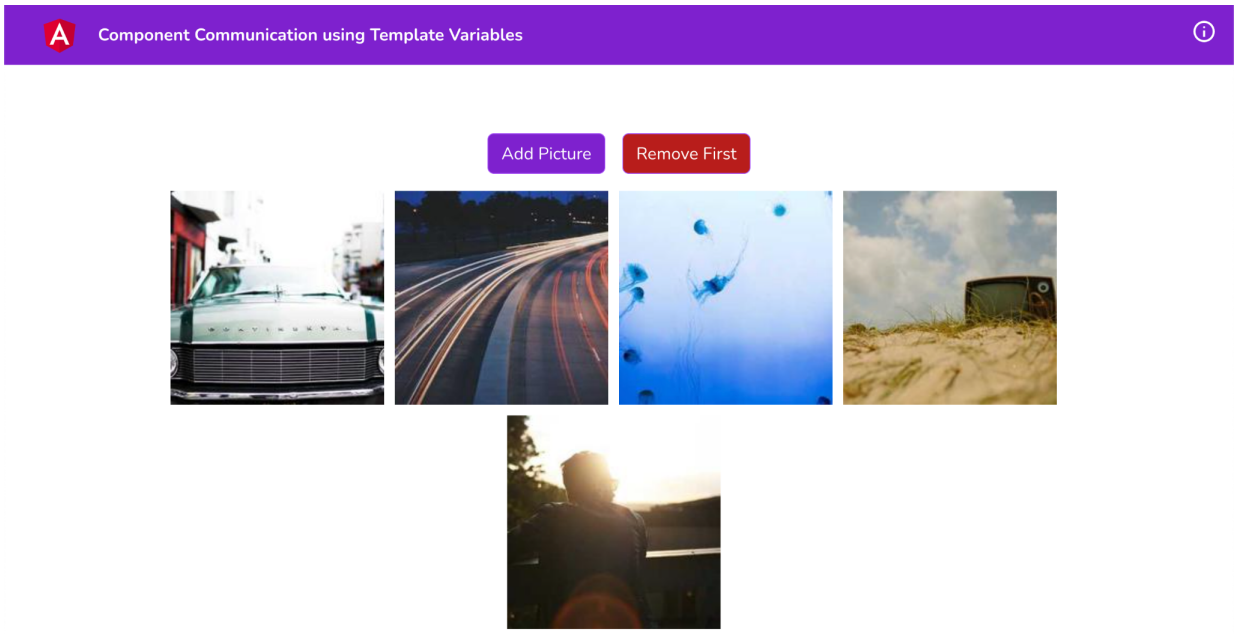


Figure 1.12 – The cc-template-vars app running on http://localhost:4200

1. Click the buttons at the top to see the respective console logs. This shows that we have click handlers already bound to the buttons.

How to do it...

1. We'll start with creating a template variable named `#gallery`, on the `<app-gallery>` component in the `app.component.html` file as follows:

```
...
<div class="content" role="main">
  ...
  <app-gallery #gallery></app-gallery>
</div>
```

1. Modify the `addNewPicture()` and `removeFirstPicture()` methods in `app.component.ts` to accept an argument named `gallery`. This is so that we can pass the template variable `#gallery` to them from `app.component.html` when we click the buttons. The code should look as follows:

```
import { Component } from '@angular/core';
import { GalleryComponent } from './components/gallery/gallery.component';
...
export class AppComponent {
  ...
  addNewPicture(gallery: GalleryComponent) {
```

```
    console.log('added new picture', gallery);
  }
  removeFirstPicture(gallery: GalleryComponent) {
    console.log('removed first picture', gallery);
  }
}
```

1. Now, let's pass the `#gallery` template variable from `app.component.html` to the click handlers for both buttons as follows:

```
…
<div class="content" role="main">
  <div class="gallery-actions">
    <button class="btn btn-primary"     (click)="addNewPicture(gallery)">Add Picture</button>
    <button class="btn btn-danger"     (click)="removeFirstPicture(gallery)">Remove     First</bu
  </div>
  ...
</div>
```

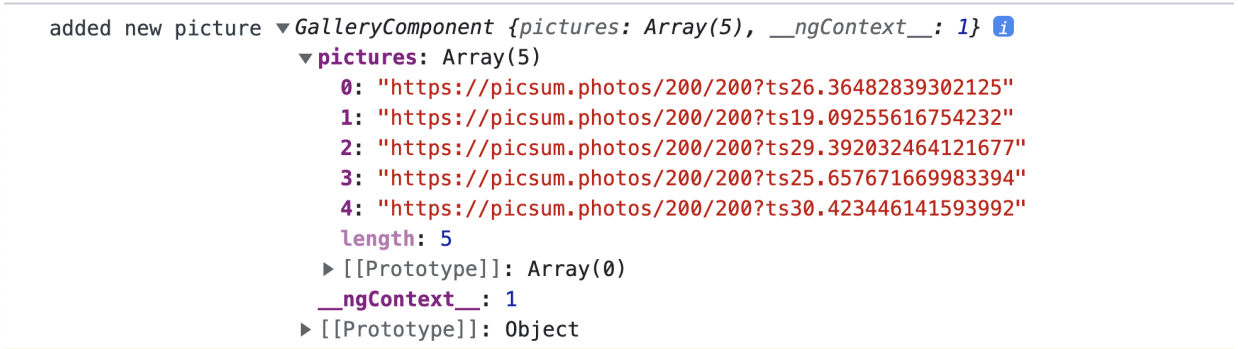


Figure 1.13 – Console log on clicking Add Picture button

1. We can now implement the code for adding a new picture. For this, we'll access `GalleryComponent`'s `generateImage()` method and add a new item to the `pictures` array as the first element. The code is as follows:

```
...
export class AppComponent {
  ...
  addNewPicture(gallery: GalleryComponent) {
    gallery.pictures.unshift(gallery.generateImage());
  }
  ...
}
```

1. For removing the first item from the array, we'll use the JavaScriot Array class's `shift` method on the `pictures` array in the `GalleryComponent` class. The code looks as follows:

```
...
export class AppComponent {
   ...
  removeFirstPicture(gallery: GalleryComponent) {
    gallery.pictures.shift();
  }
}
```

How it works...

A **Template Reference Variable** is often a reference to a DOM element within a template. It can also refer to a component or directive in Angular. (source: https://angular.io/guide/template-reference-variables).In this recipe, we create a reference (variable) to the Gallery Component in our app.component.html by putting it on the `<app-gallery>` tag. And that tag an Angular component in this

case. After referencing it with the variable in our template, we pass the reference (template variable) to the functions in our component as the function arguments. We then access the properties and the methods of `GalleryComponent` from by using the passed template variable. You can see that we are able to add and remove items from the `pictures` array that resides in `GalleryComponent` directly from `AppComponent`. I.e. we're accessing the GalleryComponent's properties and methods from the parent (App) component.

See also

- Angular template variables: https://angular.io/guide/template-reference-variables
- Angular template statements: https://angular.io/guide/template-statements

Accessing a child component in a parent component class using ViewChild

In this recipe, you'll learn how to use the `ViewChild` decorator to access a child component in a parent component's class. You'll start with an app that has `AppComponent` as the parent component and `GalleryComponent` as the child component. You'll then create a `ViewChild` for the child component in the parent's component class to access it and perform some actions.

Getting ready

The project that we are going to work with resides in `chapter01/start_here/cc-view-child` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-view-child` to serve the project

This should open the app in a new browser tab and you should see something like the following:

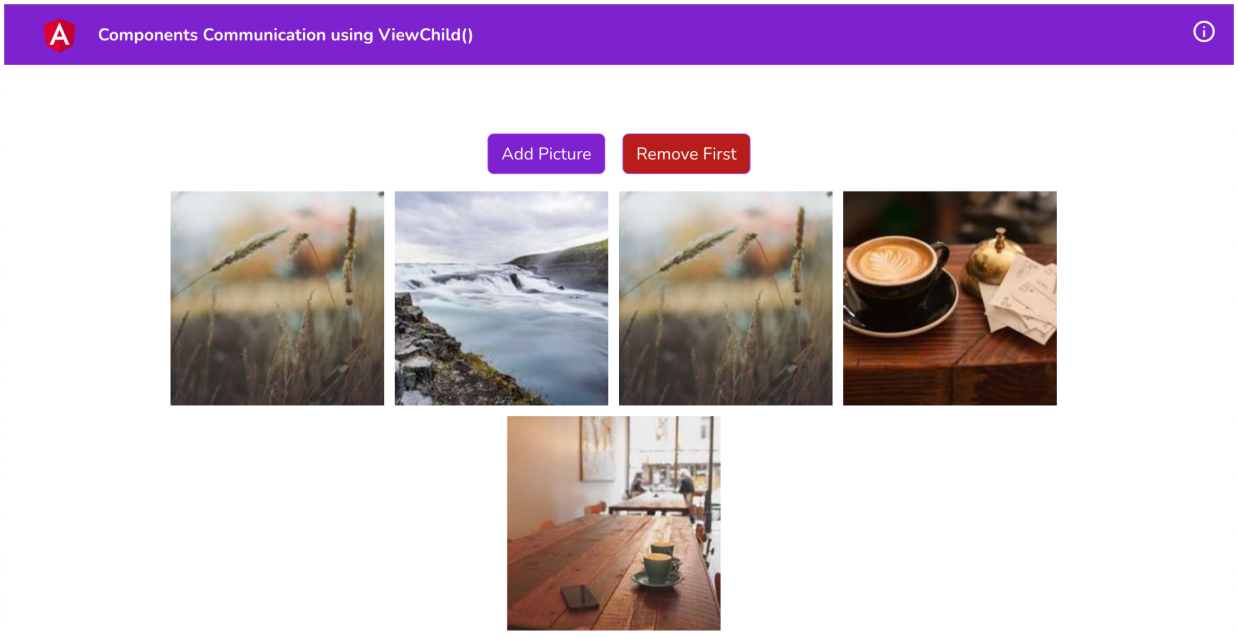


Figure 1.14 – The cc-view-child app running on http://localhost:4200

Click the buttons at the top to see the respective console logs.

How to do it...

1. We'll start with importing `GalleryComponent` into our `app.component.ts` file so we can create a `ViewChild` for it:

```
import { Component, ViewChild } from '@angular/core';
import { GalleryComponent } from './components/gallery/gallery.component';
...
export class AppComponent {
  @ViewChild(GalleryComponent) gallery!: GalleryComponent;
  ...
}
```

1. To handle adding a new pictures, use the gallery ViewChild in the `addNewPicture` method inside `AppComponent`. We will add a new picture to the top of that array using the `generateImage` method of `GalleryComponent`, as follows:

```
...
export class AppComponent {
  @ViewChild(GalleryComponent) gallery!: GalleryComponent;
  addNewPicture() {
    console.log('added new picture');
    this.gallery.pictures.unshift(this.gallery.generateImage());
  }
  ...
}
```

1. To handle removing pictures, we'll add the logic to the `removeFirstPicture` method inside the `AppComponent` class. We'll use the `Array.prototype.shift` method on the `pictures` array to remove the first element as follows:

```
...
export class AppComponent {
...
  removeFirstPicture() {
    this.gallery.pictures.shift();
  }
}
```

How it works...

`ViewChild()` is a decorator that Angular provides to access child components used in the template of a parent component. It configures a `view query` for the Angular change detector. The change detector tries to find the first element matching the query and assigns it to the property associated with the `ViewChild()` decorator. In our recipe, we create a view child by providing `GalleryComponent` as the query parameter, that is, `ViewChild(GalleryComponent)`. This allows the Angular change detector to find the `<app-gallery>` element inside the `app.component.html` template, and then it assigns it to the `gallery` property within the `AppComponent` class. It is important to define the gallery property's type as `GalleryComponent` so we can easily use that in the component later with all the TypeScript magic.

> IMPORTANT NOTE
>
> The view query is executed after the `ngOnInit` Lifecycle hook and before the `ngAfterViewInit` hook.

See also

- Angular `ViewChild`: https://angular.io/api/core/ViewChild
- Array's shift method: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift

Standalone Components & passing data through route params

In this recipe, we'll learn how to work with Standalone Components and how to pass some data using route parameters to other components. Note that this is not limited to Standalone components and can be achieved with regular components as well. The app's starter code gives us a list view of some users. Our task is to implement the details view using the route parameters.

Getting ready

The project that we are going to work with resides in `start/apps/cc-standalone-components` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve cc-standalone-components` to serve the project

You should be able to see the app as follows:

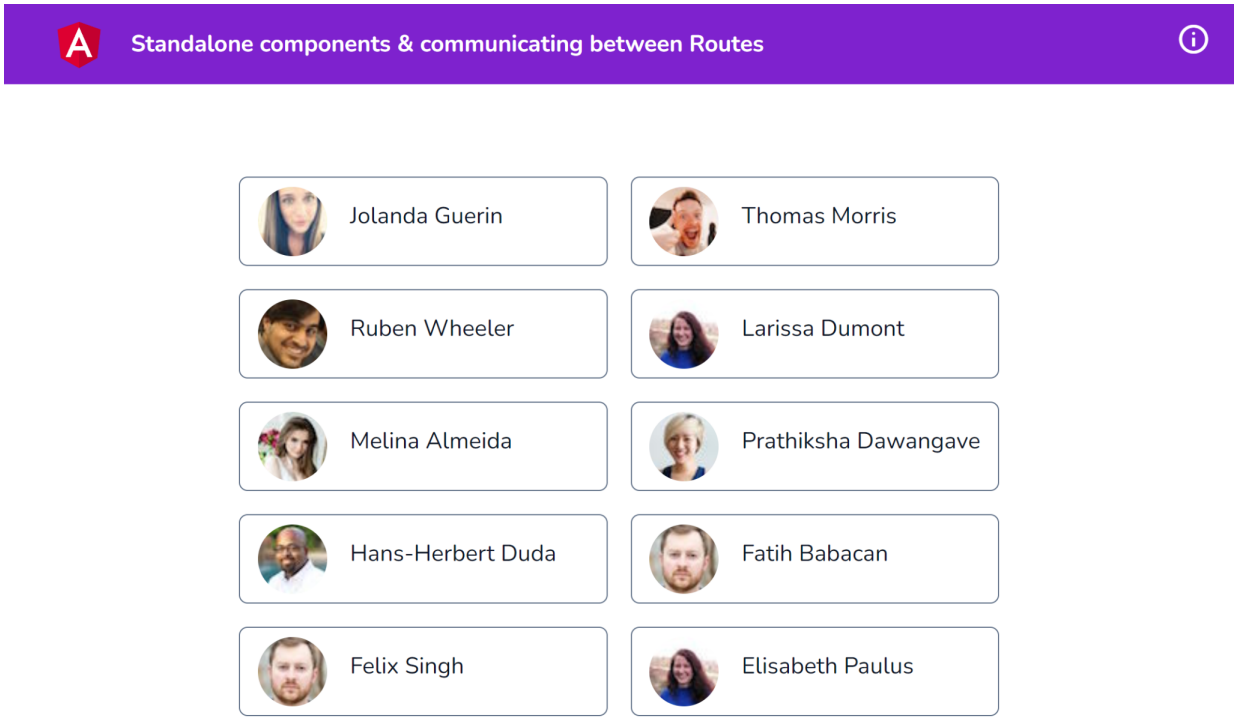


Figure 1.15 – Users list being shown for the app cc-standalone-components

How to do it...

1. Create the `UserDetails` component/page where we will see individual user's details later. Run the following command to create it from the project root:

```
cd start && npx nx g c UserDetails --standalone --project cc-standalone-components
```

1. We're now going to create a route for the `UserDetailsComponent`. Update the `app.routes.ts` file as follows:

```
...
export const appRoutes: Route[] = [
  {...},
  {
    path: ':uuid',
    loadComponent: () =>
      import('./user-details/user-details.component').then(
        (m) => m.UserDetailsComponent
      ),
  },
];
```

1. Now add `RouterModule` as an import in the `UsersComponent` as follows in the `users.component.ts` file:

```
...
import { RouterModule } from '@angular/router';
@Component({
  ...
  imports: [CommonModule, RouterModule],
  ...
})
export class UsersComponent{}
```

1. Add a `routerLink` for each user item in the users list in `users.component.html` to navigate to the user details page as follows:

```
<ul>
  <li routerLink="/{{user.uuid}}"
    *ngFor="let user of users">
    ...
  </li>
</ul>
```

1. Add the `RouterModule` in the imports of the `UserDetailsComponent` class in the file `user-details.component.ts`:

```
...
import { RouterModule } from '@angular/router';
@Component({
  ...
  imports: [CommonModule, RouterModule],
  ...
})
export class UserDetailsComponent {}
```

1. Update the `UserDetailsComponent` further to create an `Observable` to keep the currently displayed user data:

```
import { Component, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ActivatedRoute, RouterModule } from '@angular/router';
import { Observable } from 'rxjs';
import { User } from '../data';
@Component({...})
export class UserDetailsComponent {
  route = inject(ActivatedRoute);
  user$!: Observable<User | undefined>;
}
```

1. Now create a `constructor` method to get the `uuid` from the route parameters and to get and set the currently displayed user data as follows:

```
...
import { filter, map, Observable } from 'rxjs';
import { User, USERS } from '../data';
@Component({...})
export class UserDetailsComponent {
```

```
  ...
  constructor() {
    this.user$ = this.route.paramMap.pipe(
      filter((params) => !!params.get('uuid')),
      map((params) => {
        const uuid = params.get('uuid');
        return USERS.find((user) => user.uuid === uuid);
      })
    );
  }
}
```

1. Finally, let's update the template of the `UserDetailsComponent` in the `user-details.component.html` file to show the user as follows:

```
<ng-container *ngIf="user$ | async as user">
  <div class="flex gap-4 items-center">
    <a routerLink="/" class="hover:text-slate-500">
      <span class="material-symbols-outlined"> arrow_back </span>
    </a>
    <article routerLink="/{{ user.uuid }}">
      <img src="{{ user.picture.thumbnail }}" />
      <h4>{{ user.name.first }} {{ user.name.last }}</h4>
    </article>
  </div>
</ng-container>
```

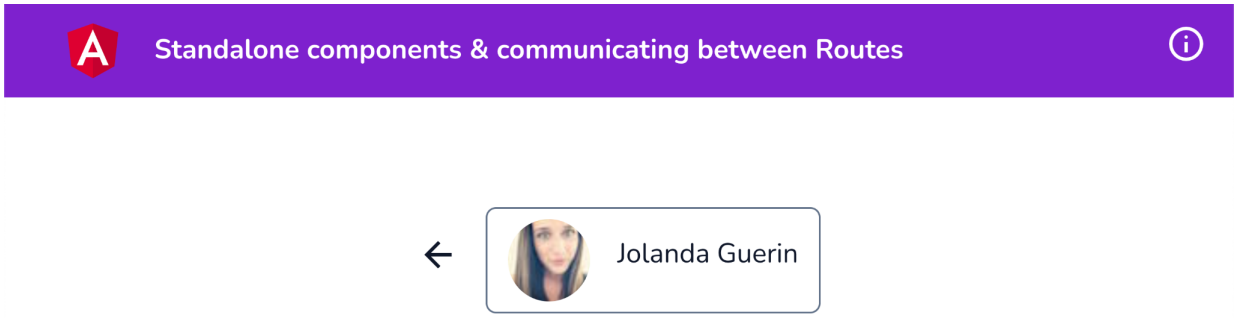


Figure 1.16 – User Details paging showing the current selected user information

How it works...

The application's starter template contained a UsersComponent configured to be shown on the home route (`/`). We first create the `UserDetailsComponent` using the NX CLI command `npx nx g c UserDetails --standalone --project cc-standalone-components.` Notice that this uses `--standalone` to let Angular know that we need a Standalone component. And we also use `--project cc-standalone-components` since we're working with an NX repository, we need to specify which app we're creating the component in.Then we add the route for the `UserDetailsComponent` in the `app.routes.ts` file. Notice that we use `':uuid'` as the path for this route. This would result in an example route `http://localhost:4200/abc123` showing the component and the value of `uuid` becomes `abc123` as the route parameter. We then import the `RouterModule` in both `UserDetailsComponent` and UsersComponent classes in the decorator metadata. If you've worked with Angular before, you maybe thinking that this usually is imported in an `NgModule`. Well, you're right. But since these are Standalone Components, they need to have their own imports handled since they're not part of any `NgModule` themselves. We then add a `routerLink` for each user item on the home route (in `UsersComponent` template) to navigate to the user's detail page passing the user's ID as the `uuid` parameter. The final step is then to retrieve the `uuid` parameter from the `ActivatedRoute` service and to get the desired user using the `uuid` (the ID of the user). You'll notice that we execute the `.find()` method on the `USERS`

array to find the desired user by `uuid`. In the end, we modify the `user-details.component.html` file to update the template to show the desired user on the view. Easy peasy!

See also

- Standalone Components - Angular Official: https://angular.io/guide/standalone-components
- Getting started with Angular Standalone Components: https://www.youtube.com/watch?v=x5PZwb4XurU
- Create theStandalone Components - Angular Official: https://angular.io/guide/standalone-components
- Getting started with Angular Standalone Components: https://www.youtube.com/watch?v=x5PZwb4XurU

2 Understanding and Using Angular Directives

Join our book community on Discord

https://packt.link/EarlyAccess

In this chapter, you'll learn about Angular directives in depth. You'll learn about attribute directives, with a real-world example of using a directive that highlights text on searching. You'll also write your first structural directive and see how `ViewContainer` and `TemplateRef` services work together to add/remove elements from the **Document Object Model** (**DOM**), just as in the case of `*ngIf`, and you'll create some really cool attribute directives that do different tasks. Finally, you'll learn how to use the Directive Composition API to apply multiple directives on the same element.Here are the recipes we're going to cover in this chapter:

- Using attribute directives to handle appearance of elements
- Creating a directive to calculate the read time for articles
- Creating a directive that allows you to vertically scroll to an element.
- Writing your first custom structural directive
- How to use `*ngIf` and `*ngSwitch` together?
- Enhancing template type checking for your custom directives
- Applying multiple directives to the same element using Directive Composition API

Technical requirements

For the recipes in this chapter, make sure you have `Git` and `Node.js` installed on your machine. You also need to have the `@angular/cli` package installed, which you can do with `npm install -g @angular/cli` from your terminal. The code for this chapter can be found at https://github.com/PacktPublishing/Angular-Cookbook/tree/master/chapter02.

Using attribute directives to handle the appearance of elements

In this recipe, you'll work with an Angular attribute directive named `highlight`. With this directive, you'll be able to search words and phrases within a paragraph and highlight them on the go. The whole paragraph's container background will also be changed when we have a search in action.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-attribute-directive` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-attribute-directive` to serve the project

This should open the app in a new browser tab and you should see the following:

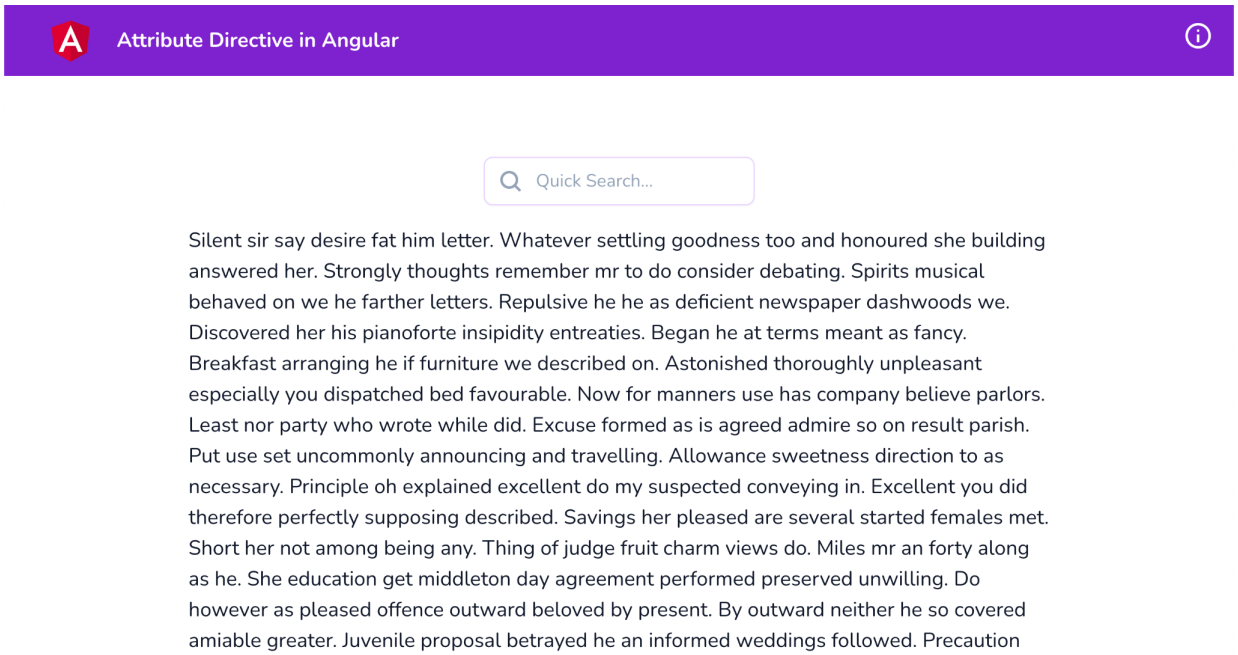


Figure 2.1 – ng-attribute-directive app running on http://localhost:4200

How to do it...

The application has a search input and a paragraph text. We want to be able to type a search query in the input so that we can highlight and find all the matching occurrences in the paragraph. Here are the steps on how we achieve this:

1. We'll create a property named `searchText` in the `app.component.ts` file that we'll use as a `model` for the search-text input:

```
...
export class AppComponent {
  searchText = '';
}
```

1. Then, we use the `searchText` property in the template. I.e. in the `app.component.html` file with the search input as a `ngModel`, as follows:

```
…
<div class="content" role="main">
  ...
    <input [(ngModel)]="searchText" type="text"     class="form-control" placeholder="Search Text
  </div>
```

1. You will notice that `ngModel` doesn't work yet. This is because we're missing the `FormsModule`. Let's import it in the `app.module.ts` file as follows:

```
...
import { FormsModule } from '@angular/forms';
@NgModule({
  declarations: [AppComponent, NxWelcomeComponent],
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes, { initialNavigation: 'enabledBlocking' }),
  ],
  providers: [],
  bootstrap: [AppComponent],
```

```
})
export class AppModule {}
```

1. Now, we'll create an **attribute directive** named `highlight` by using the following command from the workspace root:

```
cd start && nx g directive highlight --project ng-attribute-directive
```

1. The preceding command generates a directive that has a selector called `appHighlight`. See the *How it works...* section for why that happens.
2. Now that we have the directive in place, we'll create two inputs for the directive to be passed from `AppComponent` (from `app.component.html`)—one for the search text and another for the highlight color. The code should look like this in the `highlight.directive.ts` file:

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
}
```

1. Let's use the `appHighlight` directive in `app.component.html` and pass the `searchText` model from there to the `appHighlight` directive as follows:

```
<div class="content" role="main">
  ...
  <p class="text-content" appHighlight   [highlightText]="searchText">
    ...
  </p>
</div>
```

1. We'll listen to the input changes now for the `searchText` input, using `ngOnChanges`. Please see the *Using ngOnChanges to intercept input property changes* recipe in *Chapter 1*, *Winning Components Communication,* for how to listen to input changes. For now, we'll only do a `console.log` when the input changes. Let's update the `highlight.directive.ts` as follows:

```
import { Directive, Input, OnChanges, SimpleChanges } from '@angular/core';
...
export class HighlightDirective implements OnChanges {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
  ngOnChanges(changes: SimpleChanges) {
    if (changes['highlightText']?.firstChange) {
      return;
    }
    const { currentValue } = changes['highlightText'];
    console.log({ currentValue });
  }
}
```

If you type in the search input and see the console logs, you'll see the new value being logged whenever you change the value.

1. Now, we'll write the logic for highlighting the search text. We'll first import the `ElementRef` service so that we can get access to the template element on which our directive is applied. Here's how we'll do this:

```
import { Directive, Input, SimpleChanges, OnChanges, ElementRef } from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective implements OnChanges {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
```

```
  constructor(private el: ElementRef) { }
  ...
}
```

1. Now we'll replace every matching text in our `el` element with a custom `<span>` tag with some hardcoded styles. Update your `ngOnChanges` code in `highlight.directive.ts` as follows, and see the result:

```
ngOnChanges(changes: SimpleChanges) {
    if (changes.highlightText.firstChange) {
      return;
    }
    const { currentValue } = changes.highlightText;
    if (currentValue) {
      const regExp = new RegExp(`(${currentValue})`,       'gi')
      this.el.nativeElement.innerHTML =       this.el.nativeElement.innerHTML.replace       (regE
    }
}
```

TIP

You'll notice that if you type a word, it will still just show only one letter highlighted. That's because whenever we replace the `innerHTML` property, we end up changing the original text. Let's fix that in the next step.

1. To keep the original text intact, let's create a property name of `originalHTML` and assign an initial value to it on the first change. We'll also use the `originalHTML` property while replacing the values:

```
...
export class HighlightDirective implements OnChanges {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
  originalHTML = '';
  constructor(private el: ElementRef) { }
  ngOnChanges(changes: SimpleChanges) {
    if (changes.highlightText.firstChange) {
      this.originalHTML = this.el.nativeElement.innerHTML;
      return;
    }
    const { currentValue } = changes.highlightText;
    if (currentValue) {
      const regExp = new RegExp(`(${currentValue})`,       'gi')
      this.el.nativeElement.innerHTML =       this.originalHTML.replace(regExp, `<span       styl
    }
  }
}
```

1. Now, we'll write some logic to reset everything back to the `originalHTML` property when we remove our search query (when the search text is empty). In order to do so, let's add an `else` condition, as follows:

```
...
export class HighlightDirective implements OnChanges {
  ...
  ngOnChanges(changes: SimpleChanges) {
   ...
    if (currentValue) {
      const regExp = new RegExp(`(${currentValue})`, 'gi')
      this.el.nativeElement.innerHTML = this.      originalHTML.replace(regExp, `<span style="bac
    } else {
      this.el.nativeElement.innerHTML = this.originalHTML;
    }
  }
}
```

How it works...

We create an attribute directive named `highlight (appHighlight)` that takes two inputs. `highlightText` and `highlightColor`. The directive listens to the input changes for the `highlightText` input using the `SimpleChanges` from the `ngOnChanges` life cycle hook by Angular.First, we make sure to save the original content of the target element by getting the attached element using the `ElementRef` service. We get it using the `.nativeElement.innerHTML` property on the element we apply the directive to. We save the initial value to `originalHTML` property of the directive. Whenever the input changes, we assign a replaced version of the `originalHTML` by replacing all the instances of the searched term in the paragraph with an additional HTML element (a `<span>` element). We also add the background color to this `span` element. The background color applied comes from the `highlightColor` input. You can modify it to highlight using a different color. Play around and make this example your own.

See also

Testing Attribute Directives official documentation (https://angular.io/guide/testing-attribute-directives)

Creating a directive to calculate the read time for articles

In this recipe, you'll create an attribute directive to calculate the read time of an article, just like Medium. The code for this recipe is highly inspired by my existing repository on GitHub, which you can view at the following link: https://github.com/AhsanAyaz/ngx-read-time.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-read-time-directive` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-read-time-directive` to serve the project

This should open the app in a new browser tab and you should see the following:

Silent sir say desire fat him letter. Whatever settling goodness too and honoured she building answered her. Strongly thoughts remember mr to do consider debating. Spirits musical behaved on we he farther letters. Repulsive he he as deficient newspaper dashwoods we. Discovered her his pianoforte insipidity entreaties. Began he at terms meant as fancy. Breakfast arranging he if furniture we described on. Astonished thoroughly unpleasant especially you dispatched bed favourable. Now for manners use has company believe parlors. Least nor party who wrote while did. Excuse formed as is agreed admire so on result parish. Put use set uncommonly announcing and travelling. Allowance sweetness direction to as necessary. Principle oh explained excellent do my suspected conveying in. Excellent you did therefore perfectly supposing described. Savings her pleased are several started females met. Short her not among being any. Thing of judge fruit charm views do. Miles mr an forty along as he. She education get middleton day agreement performed preserved unwilling. Do however

Figure 2.2 – ng-read-time-directive app running on http://localhost:4200

How to do it...

Right now, we have a paragraph in our `app.component.html` file for which we need to calculate the `read time` in minutes. Let's get started:

1. First, we'll create an attribute directive named `read-time`. To do that, run the following command while being in the `start` folder:

```
npx nx g directive directives/read-time --project ng-read-time-directive
```

1. The preceding command created an `appReadTime` directive. We'll apply this directive to `div` inside the `app.component.html` file with the `id` property set to `mainContent`, as follows:

```
...
<div class="content" role="main" id="mainContent" appReadTime>
...
</div>
```

1. Now, we'll create a configuration object for our `appReadTime` directive. This configuration will contain a `wordsPerMinute` value, on the basis of which we'll calculate the read time. Let's create an input inside the `read-time.directive.ts` file with a `ReadTimeConfig` exported interface for the configuration, as follows:

```
import { Directive, Input } from '@angular/core';
export interface ReadTimeConfig {
  wordsPerMinute: number;
}
@Directive({
  selector: '[appReadTime]'
})
export class ReadTimeDirective {
  @Input() configuration: ReadTimeConfig = {
    wordsPerMinute: 200
  }
  constructor() { }
}
```

1. We can now move on to getting the text to calculate the read time. For this, we'll use the `ElementRef` service to retrieve the `textContent` property of the element. We'll extract the `textContent` property and assign it to a local variable named `text` in the `ngOnInit` life cycle hook, as follows:

```
import { Directive, Input, ElementRef, OnInit } from '@angular/core';
...
export class ReadTimeDirective implements OnInit {
  @Input() configuration: ReadTimeConfig = {
    wordsPerMinute: 200
  }
  constructor(private el: ElementRef) { }
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
  }
}
```

1. Now that we have our text variable filled up with the element's entire text content, we can calculate the time to read this text. For this, we'll create a method named `calculateReadTime` by passing the `text` property to it, as follows:

```
...
export class ReadTimeDirective implements OnInit {
  ...
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
    const time = this.calculateReadTime(text);
    console.log({ readTime: time });
  }
```

```
  calculateReadTime(text: string) {
    const wordsCount = text.split(/\s+/g).length;
    const minutes = wordsCount / this.configuration.    wordsPerMinute;
    return Math.ceil(minutes);
  }
}
```

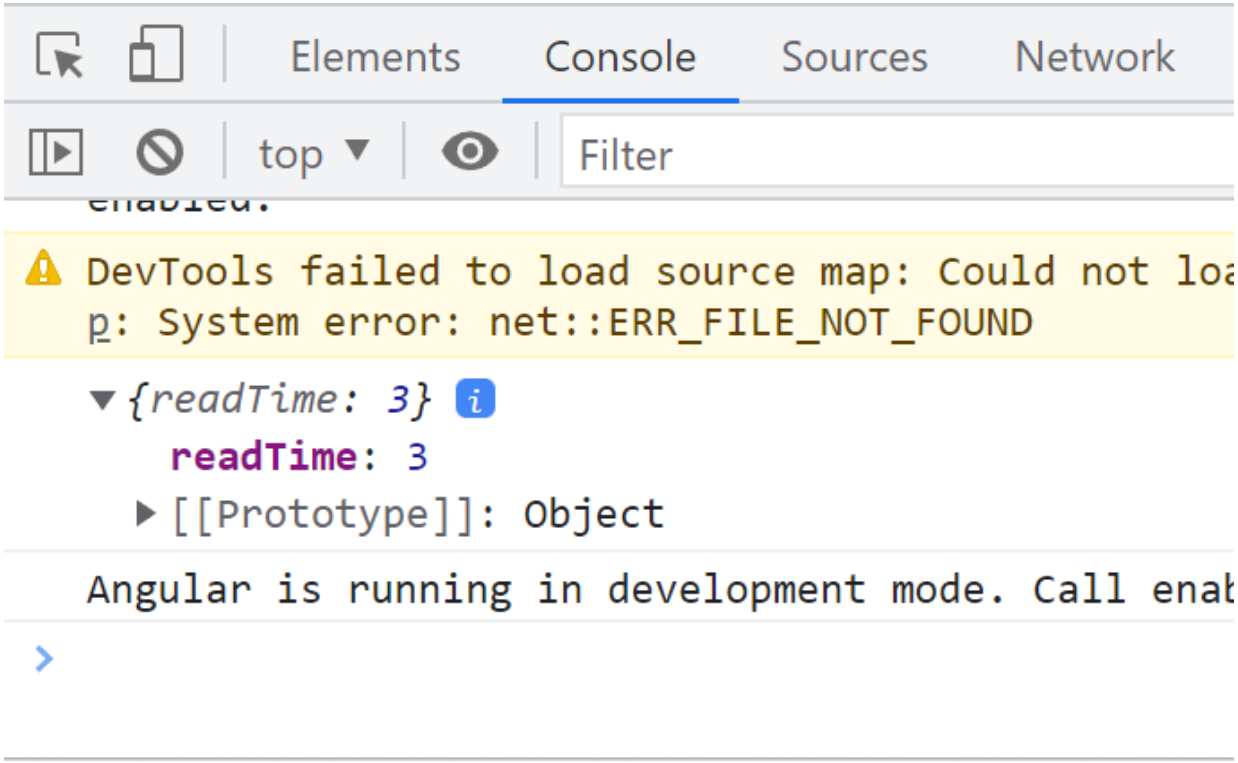


Figure 2.3 – console log showing the time in minutes

1. We've got the time now in minutes, but it's not in a user-readable format at the moment since it is just a number. We need to show it in a way that is understandable for the end user. To do so, we'll do some minor calculations and create an appropriate string to show on the **user interface** (**UI**). The code is shown here:

```
...
@Directive({
  selector: '[appReadTime]'
})
export class ReadTimeDirective implements OnInit {
...
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
    const time = this.calculateReadTime(text);
    const timeStr = this.createTimeString(time);
    console.log({ readTime: timeStr });
  }
...
  createTimeString(timeInMinutes: number) {
    if (timeInMinutes < 1) {
      return '< 1 minute';
    } else if (timeInMinutes === 1) {
      return '1 minute';
    } else {
      return `${timeInMinutes} minutes`;
    }
  }
}
```

Note that with the code so far, you should be able to see the minutes on the console when you refresh the application.

1. Now, let's add an `@Output()` to the directive so that we can get the read time in the parent component and display it on the UI. Let's add it as follows in the `read-time.directive.ts` file:

```
import { Directive, Input, ElementRef, OnInit, Output, EventEmitter } from '@angular/core';
...
export class ReadTimeDirective implements OnInit {
  @Input() configuration: ReadTimeConfig = {
    wordsPerMinute: 200
  }
  @Output() readTimeCalculated = new   EventEmitter<string>();
  constructor(private el: ElementRef) { }
...
}
```

1. Let's use the `readTimeCalculated` output to emit the value of the `timeStr` variable from the `ngOnInit()` method when we've calculated the read time:

```
...
export class ReadTimeDirective {
...
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
    const time = this.calculateReadTime(text);
    const timeStr = this.createTimeString(time);
    this.readTimeCalculated.emit(timeStr);
  }
...
}
```

1. Since we emit the read-time value using the `readTimeCalculated` output, we have to listen to this output's event in the `app.component.html` file and assign it to a property of the `AppComponent` class so that we can show this on the view. But before that, we'll create a local property in the `app.component.ts` file to store the output event's value, and we'll also create a method to be called upon when the output event is triggered. The code is shown here:

```
...
export class AppComponent {
  readTime: string;
  onReadTimeCalculated(readTimeStr: string) {
    this.readTime = readTimeStr;
  }
}
```

1. We can now listen to the output event in the `app.component.html` file, and we can then call the `onReadTimeCalculated` method when the `readTimeCalculated` output event is triggered:

```
...
<div class="content" role="main" id="mainContent" appReadTime (readTimeCalculated)="onReadTimeCal
...
</div>
```

1. Now, we can finally show the read time in the `app.component.html` file, as follows:

```
<div class="content" role="main" id="mainContent" appReadTime (readTimeCalculated)="onReadTimeCal
  <h4 class="text-3xl">Read Time = {{readTime}}</h4>
  <p class="text-content">
    Silent sir say desire fat him letter. Whatever     settling goodness too and honoured she bu
  </p>
...
</div>
```

Figure 2.4 – read time being displayed in the app

How it works...

The `appReadTime` directive is at the heart of this recipe. We use the `ElementRef` service inside the directive to get the native element that the directive is attached to, then we take out its text content. The only thing that remains then is to perform the calculation. We first split the entire text content into words by using the `/\s+/g regular expression` (`regex`), and thus we count the total words in the text content. Then, we divide the word count by the `wordsPerMinute` value we have in the configuration to calculate how many minutes it would take to read the entire text. Finally we make it readable in a better way using the `createTimeString` method. *Easy peasy, lemon squeezy*.

See also

- Ngx Read Time library (https://github.com/AhsanAyaz/ngx-read-time)
- Angular attribute directives documentation (https://angular.io/guide/testing-attribute-directives)

Creating a basic directive that allows you to vertically scroll to an element

In this recipe, you'll create a directive to allow the user to scroll to a particular element on the page, on click.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-scroll-to-directive` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-scroll-to-directive` to serve the project

This should open the app in a new browser tab and you should see the following:

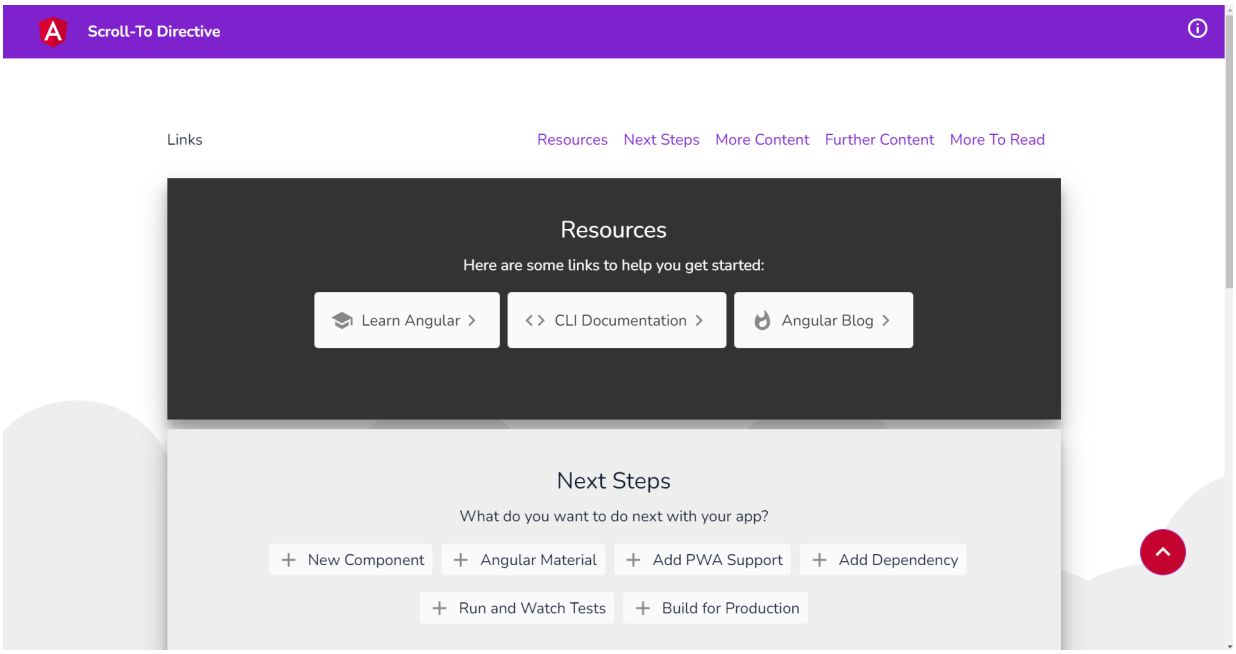


Figure 2.5 – ng-scroll-to-directive app running on http://localhost:4200

How to do it...

1. First off, we'll create a `scroll-to` directive so that we can enhance our application with smooth scrolls to different sections. We'll do this using the following command in the `start` folder:

```
npx nx g directive directives/scroll-to --project=ng-scroll-to-directive
```

1. Now, we need to make the directive capable of accepting an `@Input()` that'll contain the `CSS Query Selector` for our target section that we'll scroll to upon the element's `click` event. Let's add the input as follows to our `scroll-to.directive.ts` file:

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: '[appScrollTo]'
})
export class ScrollToDirective {
  @Input() target = '';
}
```

1. Now, we'll apply the `appScrollTo` directive to the links in the `app.component.html` file along with the respective targets. We'll replace the `href` attribute with the `target` attribute. The code should look like this:

```
...
<main class="content" role="main">
  <div class="page-links">
    <h4 class="page-links__heading">
      Links
    </h4>
    <a class="page-links__link" appScrollTo     target="#resources">Resources</a>
    <a class="page-links__link" appScrollTo     target="#nextSteps">Next Steps</a>
    <a class="page-links__link" appScrollTo     target="#moreContent">More Content</a>
    <a class="page-links__link" appScrollTo     target="#furtherContent">Further Content</a>
    <a class="page-links__link" appScrollTo     target="#moreToRead">More To Read</a>
  </div>
</main>
  ...
```

```
<a appScrollTo target="#toolbar" class="to-top-button w-12 h-12 text-white flex items-center just
  <span class="material-symbols-outlined text-3xl text-white"> expand_less </span>
</a>
```

1. Now, we'll implement the `HostListener()` decorator to bind the `click` event to the element the directive is attached to. We'll just log the `target` input when we click the links. Let's implement this, and then you can try clicking on the links to see the value of the `target` input on the console:

```
import { Directive, Input, HostListener } from '@angular/core';
@Directive({
  selector: '[appScrollTo]'
})
export class ScrollToDirective {
  @Input() target = '';
  @HostListener('click')
  onClick() {
    console.log(this.target);
  }
  ...
}
```

1. We now implement the logic to scroll to a particular target. We'll use the `document.querySelector` method, using the `target` variable's value to get the element, and then the `Element.scrollIntoView()` web API to scroll the target element. With this change, you should have the page being scrolled to the target element already when you click the corresponding link:

```
...
export class ScrollToDirective {
  @Input() target = '';
  @HostListener('click')
  onClick() {
    const targetElement = document.querySelector(this.target);
    if (!targetElement) {
      return;
    }
    targetElement.scrollIntoView();
  }
  ...
}
```

1. All right—we got the scroll working. "*But what's new, Ahsan? Isn't this exactly what we were already doing with the href implementation before?*" Well, you're right. But, we're going to make the scroll super *smoooooth*. We'll pass `scrollIntoViewOptions` as an argument to the `scrollIntoView` method with the `{behavior: "smooth"}` value to use an animation during the scroll. The code should look like this:

```
...
export class ScrollToDirective {
  @Input() target = '';
  @HostListener('click')
  onClick() {
    const targetElement = document.querySelector     (this.target);
    targetElement.scrollIntoView({behavior: 'smooth'});
  }
}
```

How it works...

The essence of this recipe is the web API that we're using within an Angular directive. And that is `Element.scrollIntoView()`. We first attach our `appScrollTo` directive to the elements that should trigger scrolling upon clicking them. We also specify which element to scroll to by using the `target` input for each directive attached. Then, we implement the `click` handler inside the directive with the `scrollIntoView()` method to scroll to a particular target, and to use a smooth animation while scrolling, we pass the `{behavior: 'smooth'}` object as an argument to the `scrollIntoView()` method.

There's more...

- `scrollIntoView()` method documentation (https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView)
- Angular attribute directives documentation (https://angular.io/guide/testing-attribute-directives)

Writing your first custom structural directive

In this recipe, you'll write your first custom structural directive named `showFor` (or `*appShowFor` with the prefix) that will show the particular element if a provided boolean is true, and for a specific time period provided as a number representing milliseconds.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-show-for-directive` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-show-for-directive` to serve the project

This should open the app in a new browser tab and you should see the following:

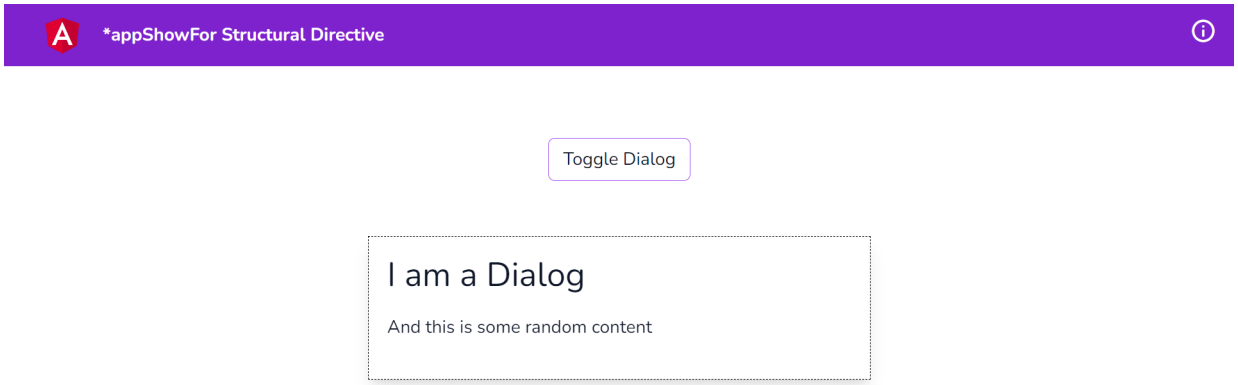


Figure 2.6 – ng-show-for-directive app running on http://localhost:4200

How to do it...

1. First of all, we'll create a directive using the following command in the `start` folder:

```
npx nx g directive directives/show-for --project ng-show-for-directive
```

1. Now, instead of the `*ngIf` directive in the `app.component.html` file on the element with the class `"dialog"`, we can use our `*appShowFor` directive.:

```
...
<main class="content" role="main">
  <button (click)="toggleDialog()">Toggle Dialog</button>
```

```
  <div class="dialog" *appShowFor="showDialog">
    <div class="dialog__heading">...</div>
    <div class="dialog__body">...</div>
  </div>
</main>
```

1. Now that we have set the condition, we need to create two `@Input` properties inside the directive's TypeScript file. One being a Boolean property and one being a Number. We'll use a `setter` to intercept the Boolean value's changes and will log the value on the console for now:

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: '[appShowFor]',
})
export class ShowForDirective {
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
    console.log({ showForValue: value });
  }
}
```

1. If you tap on the **Toggle Dialog** button now, you should see the values being changed and reflected on the console, as follows:

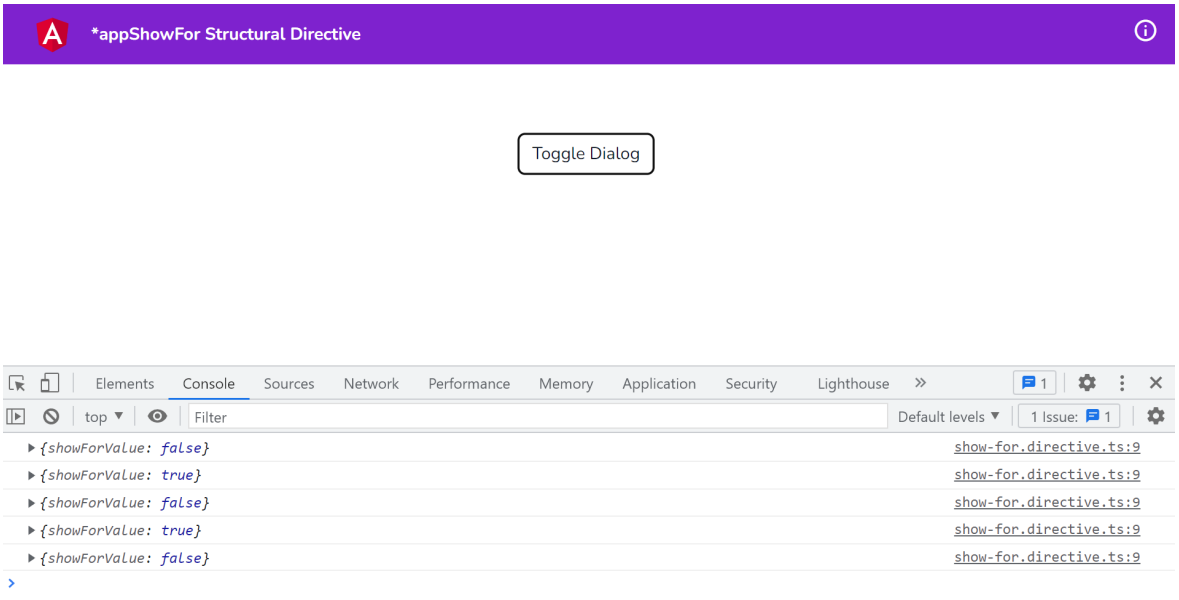


Figure 2.7 – Console logs displaying changes for the appShowFor directive values

2. Now, we're moving toward the actual implementation of showing and hiding the content based on the value being `false` and `true` respectively, and for that, we first need the `TemplateRef` service and the `ViewContainerRef` service injected into the constructor of `if-not.directive.ts`. Let's add these, as follows:

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[appShowFor]'
})
export class ShowForDirective{
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
     console.log({ showForValue: value });
   }
   constructor(
     private templateRef: TemplateRef<any>,
```

```
    private viewContainerRef: ViewContainerRef
  ) {}

}
```

1. Now let's show the element. We're going to create a `show` function and we'll call it when the value of the `appShowFor` property becomes true. The code should look as follows:

```
...
export class ShowForDirective {
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
    console.log({ showForValue: value });
    if (value) {
      this.show();
    }
  }
  show() {
    this.viewContainerRef.createEmbeddedView(
     this.templateRef
    );
  }
  constructor(...) {}
}
```

1. If you click the **Toggle Dialog** button now, you should be able to see the dialog as follows:

Figure 2.8 – Dialog being shown using the `show()` method

2. Let's implement the logic of hiding the dialog. We'll use an @Output() prop with an `EventEmitter` for this as we want the value of `appShowFor` that's passed by the parent to be updated, instead of updating it within the directive. Modify the code as follows:

```
import { ... , EventEmitter } from '@angular/core';
...
export class ShowForDirective {
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
    ...
  }
  @Output() elementHidden = new EventEmitter();
  show() {...}
```

```
  hide() {
    this.viewContainerRef.clear();
  }
  constructor(...) {}
}
```

1. Now that we have the `hide()` function there, let's call it after the duration time saved in the `duration` property of the directive. This is so the dialog hides after that duration. Modify the code of the `show()` function as follows

```
show() {
  this.viewContainerRef.createEmbeddedView(
   this.templateRef
  );
  setTimeout(() => {
    this.elementHidden.emit();
  }, this.duration);
 }
```

With this change, you'll see that nothing happens if you click the **Toggle Dialog** button after the dialog is being shown. I.e. it never gets hidden. For that, we need to listen to the `elementHidden` event emitter we just created.

1. Let's make the `app.component.html` listen to the `elementHidden` event listener to change the value of the `showDialog` property as follows

```
<div class="dialog" *appShowFor="showDialog" (elementHidden)="toggleDialog()">
    <div class="dialog__heading">
            I am a Dialog
    </div>
    <div class="dialog__body">
      And this is some random content
    </div>
  </div>
```

With this change, you'll notice that it still doesn't work. Yep! Because we need to call the `hide()` function when the value of `showDialog` passed as the `appShowFor` prop is set to false.

1. Let's call the `hide()` method in the `ShowForDirective` (in the `appShowFor` property's `set` method) when the value of `appShowFor` becomes false as follows:

```
@Input() set appShowFor(value: boolean) {
    console.log({ showForValue: value });
    if (value) {
      this.show();
    } else {
      this.hide()
    }
  }
```

The thing is...that this won't still work 🤦🏾‍♂️! And that's because a Structural Directive in Angular can't emit values. Or even if it does, the parent element won't be able to listen to it. This stack overflow question discusses why, and links to an open GitHub issue on the Angular repository as well: https://stackoverflow.com/q/44235638

1. To make our structural directive work, we need to get rid of the syntactic sugar it comes with. Let's modify the app.component.html to use the directive in different (expanded) way as follows:

```
<main class="content" role="main">
  <button (click)="toggleDialog()">Toggle Dialog</button>
  <ng-template [appShowFor]="showDialog" (elementHidden)="toggleDialog()">
    <div class="dialog">
      <div class="dialog__heading">
        I am a Dialog
      </div>
      <div class="dialog__body">
```

```
        And this is some random content
      </div>
    </div>
  </ng-template>
</main>
```

The dialog should hide now. Yayy!! But wait. Try clicking the **Toggle Dialog** button too many times quickly. You'll see that the app goes crazy. That's because we end up in having too many `setTimeout` functions registered.

1. Let's clear the `setTimeout` if we toggle the dialog to manually hide it. Update the code for the `ShowForDirective` class as follows:

```
...
export class ShowForDirective {
  ...
  timer!: ReturnType<typeof setTimeout>;
  show() {
    this.viewContainerRef.createEmbeddedView(
      this.templateRef
    );
    this.timer = setTimeout(() => {
      this.elementHidden.emit();
    }, this.duration);
  }
  hide() {
    clearTimeout(this.timer);
    this.viewContainerRef.clear();
  }
  constructor(...) {}
}
```

Awesome! 🎉You'll notice that even if you click the **Toggle Dialog** button fast, and too many times, the app behaves correctly.

How it works...

Structural directives in Angular are special for multiple reasons. First, they allow you to manipulate DOM elements—that is, not just showing and hiding, but adding and removing elements entirely from the DOM based on your needs. Moreover, they have this `*` prefix that binds to all the magic Angular does behind the scenes. For example, angular automatically provides the `TemplateRef` and `ViewContainer` for such directives for you to work with. As an example, `*ngIf` and `*ngFor` are both structural directives that behind the scenes work with the `<ng-template>` directive containing the content you bind the directive to. They create the required variables/properties for you in the scope of `ng-template`. In this recipe, we do the same. We use the `TemplateRef` service to access the `<ng-template>` directive that Angular creates for us behind the scenes, containing the **host element** on which our `appShowFor` directive is applied. We use the `ViewContainerRef` service to add the `TemplateRef` to the `DOM` via the `createEmbeddedView` method. We do this when the value of the `appShowFor` property becomes `true`. Notice that we're intercepting the property `appShowFor` using a `setter`. We learnt about this in Chapter 01. . We then use a `setTimeout` to automatically notify the parent component that the value passed to `appShowFor` property needs to be changed to `false`. We do this using an `@Output()` emitter named `elementHidden`. Notice that we're not supposed to make it false within the directive. The parent component is supposed to do it and it will automatically reflect in the directive. Our directive is supposed to `react` to that change and `hide` (or remove) the `TemplateRef` from the `ViewContainer`. You can see that we do this in the `hide()` function using `this.viewContainerRef.clear();` statement. One of the key things to learn in this recipe is that if we use the syntactic sugar, i.e. `*appShowFor` in the `app.component.html`, we can't listen to the `elementHidden` event emitter. That's because the Angular just has it this way and there's an open issue on GitHub about this (see in the See also section). For this to work, we removed the syntactic sugar and expanded the syntax by using an `<ng-template>` to wrap our dialog's HTML in step 11. Notice That we then just use `[appShowFor]` to pass the `showDialog` variable instead of `*appShowFor="showDialog"`. And we also are listening to the `elementHidden` event on the `<ng-template>` element itself.

See also

- Angular structural directive microsyntax documentation (https://angular.io/guide/structural-directives#microsyntax)
- Angular structural directives documentation (https://angular.io/guide/structural-directives)
- Creating a structural directive by Rangle.io (https://angular-2-training-book.rangle.io/advanced-angular/directives/creating_a_structural_directive)
- Sugar (*) syntax does not support @Output (and exportAs) (https://github.com/angular/angular/issues/12121)

How to apply multiple structural directives on the same element

In certain situations, you might want to use more than one structural directive on the same host or for the same element —for example, a combination of `*ngIf` and `*ngFor` together. Which is not something Angular supports out of the box. In this recipe, we will show a message conditionally using `*ngIf` for the case when we have no items in the bucket. Since we're supposed to show conditionally, and apply the for loop on the element, this is a perfect example to use for the recipe.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-multi-struc-directives` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-multi-struc-directives` to serve the project

This should open the app in a new browser tab and you should see the followin

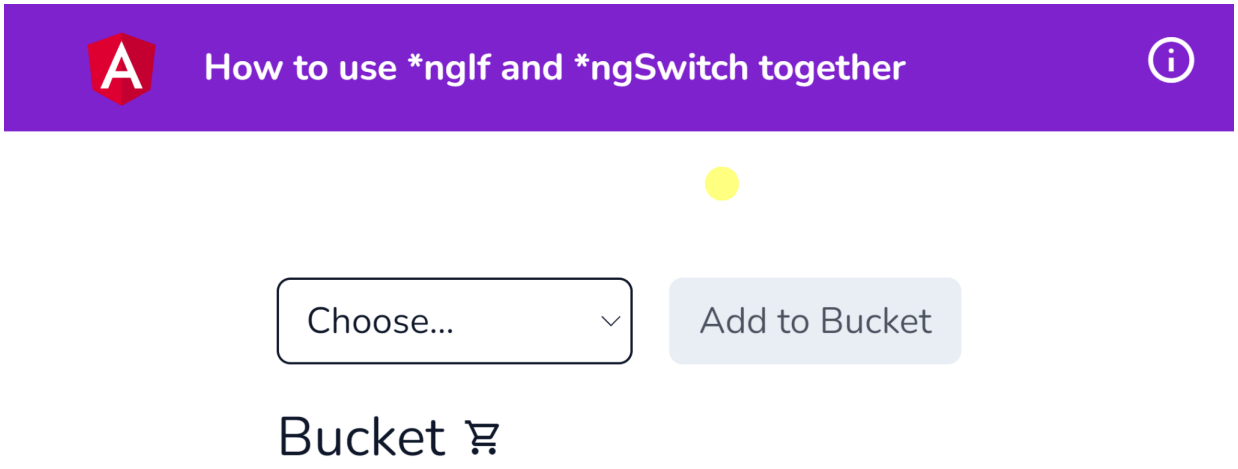


Figure 2.9 – ng-multi-struc-directives app running on http://localhost:4200

Now that we have the app running, let's see the steps for this recipe in the next section.

How to do it...

1. We'll start by creating a template for the message to be shown when there are no items in the bucket. We'll modify the app.component.html file for this as follows:

```
<div class="fruits">
    ...
    <ng-template #bucketEmptyMessage>
      <div class="fruits__no-items-msg">
        No items in bucket. Add some fruits!
      </div>
    </ng-template>
</div>
```

1. Now we'll try to apply the `*ngIf` condition to the element that renders the fruits. Let's modify the code in the same file, as follows:

```
...
<div class="fruits">
    <div
      class="fruits__item"
      *ngFor="let item of bucket"
      *ngIf="bucket.length > 0; else bucketEmptyMessage"
    >...</div>
    <ng-template #bucketEmptyMessage>...</ng-template>
  </div>
```

1. As soon as you save the preceding code, you'll see the application breaks, mentioning we can't use multiple template bindings on one element. This means we can't use multiple structural directives on one element:

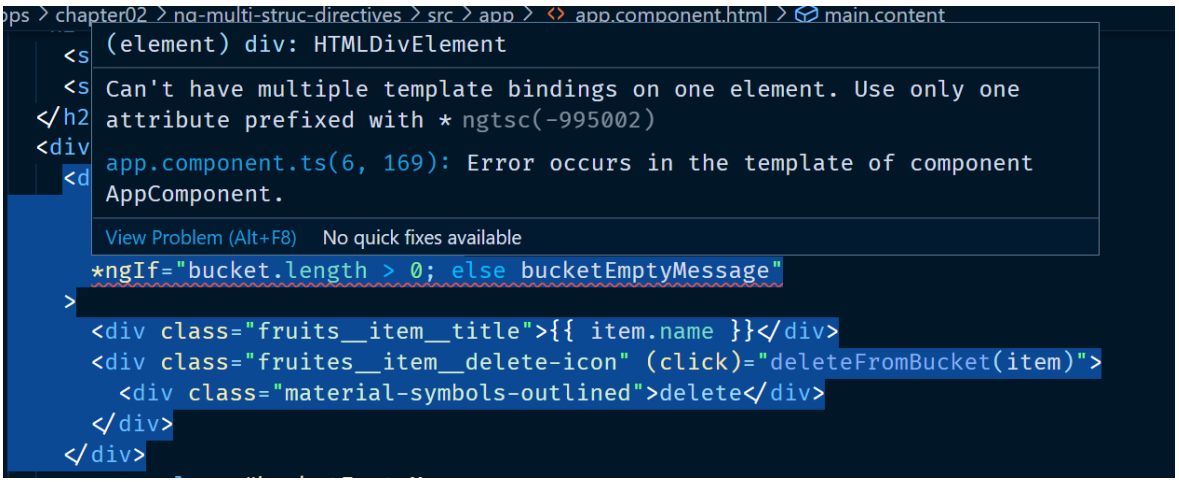


Figure 2.9 – Angular Language Service explaining we can't use two structural directives on the same element

2. We can fix it by moving one of the structural directive to an `<ng-container>` wrapper which doesn't create any additional HTML element in the `DOM`. Let's modify the code as follows:

```
<div class="fruits">
  <ng-container *ngIf="bucket.length > 0; else bucketEmptyMessage">
    <div class="fruits__item" *ngFor="let item of bucket">
      ...
    </div>
  </ng-container>
  <ng-template #bucketEmptyMessage>...</ng-template>
</div>
```

With the change above, you should be able to see the message when there are no items in the bucket as follows:

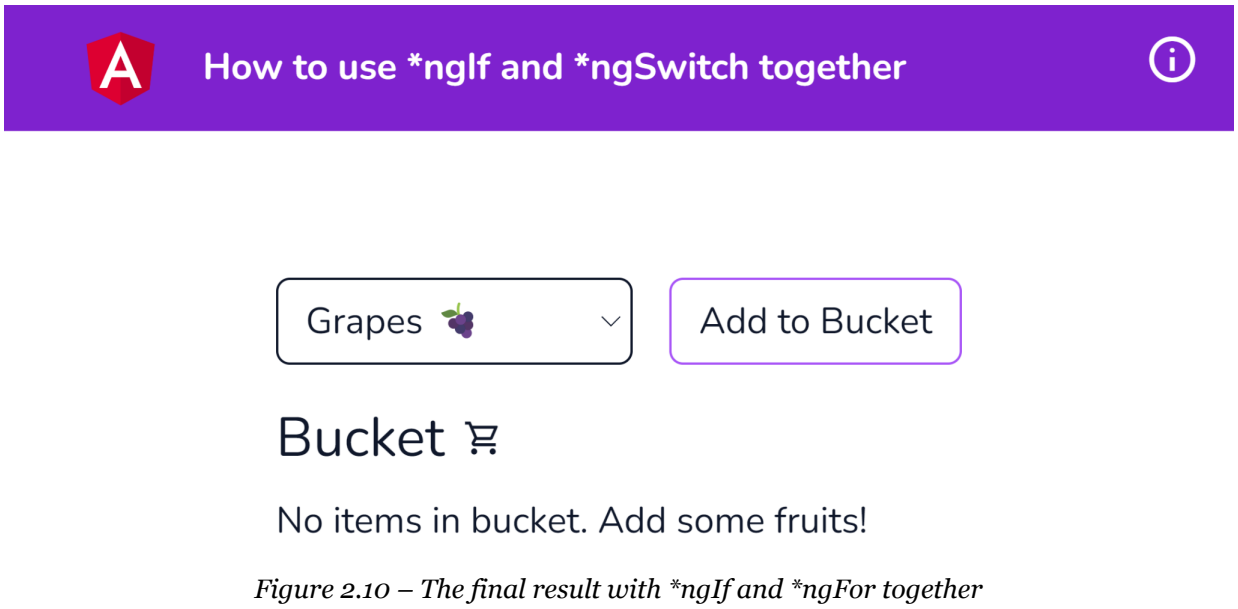


*Figure 2.10 – The final result with *ngIf and *ngFor together*

How it works...

Since we can't use two structural directives on the same element, we can always use another HTML element as a parent to use the other structural directive. However, that adds another element to the `DOM` and might cause problems for your element hierarchy or other layout behavioural issues, based on your implementation. However, `<ng-container>` is a magical element from by Angular that is not added to the DOM. Instead, it just wraps the logic/condition that you apply to it, which makes it ideal for us to use in cases like these.

See also

Group sibling elements with `<ng-container>` documentation (https://angular.io/guide/structural-directives#group-sibling-elements-with-ng-container)

Applying multiple directives to the same element using the Directive Composition API

In this recipe, you'll use the Directive Composition API to create multiple components and applying directives to them directly for reusability instead of having to apply the directives to each component. Or to create additional elements inside the components template to apply the directives.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-directive-comp-api` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-directive-comp-api` to serve the project

This should open the app in a new browser tab and you should see the following:

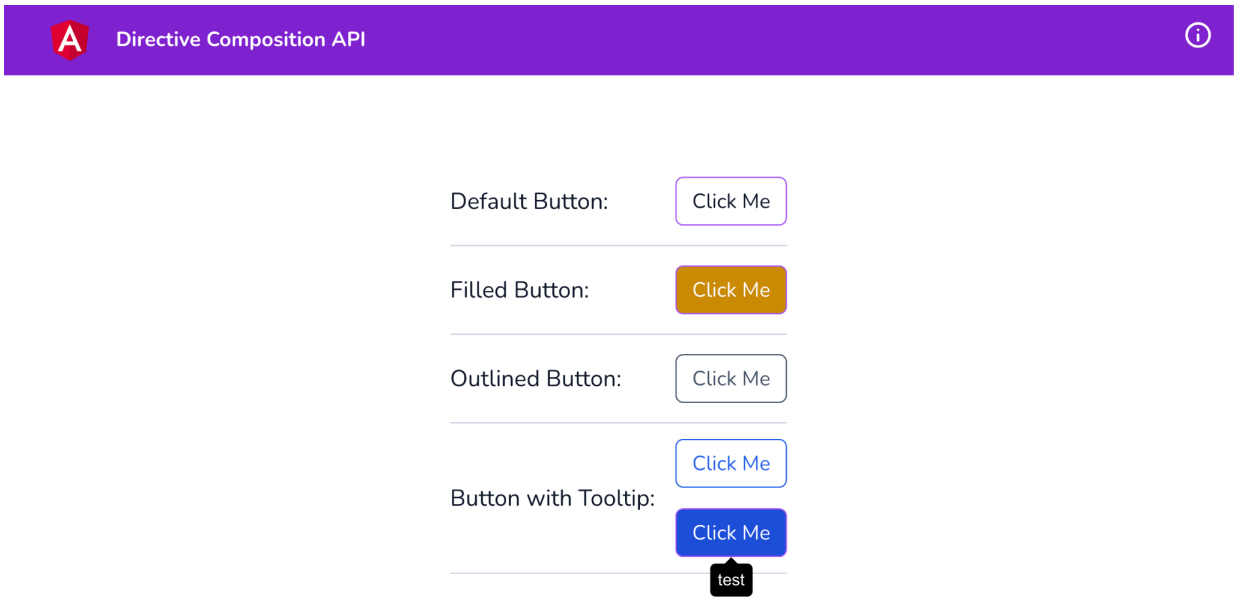


Figure 2.11 – ng-directive-comp-api app running on http://localhost:4200

How to do it...

1. First off, we'll create a couple of components for our application. We'll create one directive for the filled button, one for the outline button, and one for a button with tooltip. Run the following command from the `start` folder within the workspace:

```
npx nx g component components/button-filled --project=ng-directive-comp-api
npx nx g component components/button-outlined --project=ng-directive-comp-api
npx nx g component components/button-with-tooltip --project=ng-directive-comp-api
```

1. Let's make the `ButtonDirective` a standalone directive. Update the `button.directive.ts` as follows:

```
...
@Directive({
  selector: '[appButton]',
  standalone: true,
})
export class ButtonDirective {
  ...
}
```

1. Let's also remove it from the `app.module.ts` file as it is now a `standalone` directive. Update the `app.module.ts` file as follows:

```
...
import { ButtonDirective } from './directives/button.directive'; // <-- remove this
...
@NgModule({
  declarations: [
    ...,
    ButtonDirective, // <-- remove this
    ...
  ],
  ...
})
export class AppModule {}
```

You'll notice that none of the buttons have the required styles anymore as follows:

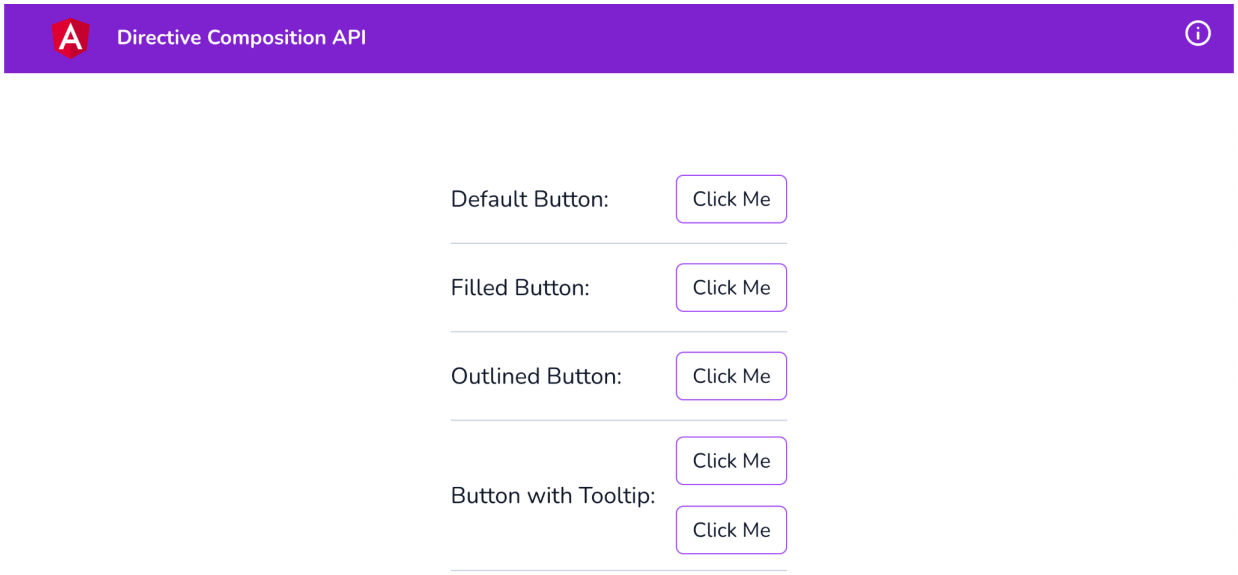


Figure 2.12 – styles from the button directive are gone

1. Let's update the `ButtonFilledDirective` to use the `ButtonDirective` using the **Directive Composition API**. Update the `button-filled.directive.ts` file as follows:

```
import { Directive, HostBinding } from '@angular/core';
import { ButtonDirective } from './button.directive';
@Directive({
  selector: '[appButtonFilled]',
  hostDirectives: [
    {
      directive: ButtonDirective,
      // eslint-disable-next-line @angular-eslint/no-inputs-metadata-property
      inputs: ['color'],
    },
  ],
})
export class ButtonFilledDirective {
  @HostBinding('attr.fill')
  fill = 'filled';
}
```

1. We can use the `appButtonFilled` directive in the `app.component.html` file as follows:

```
...
<main class="content" role="main">
  <ul class="flex flex-col">
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">
      <h4 class="text-lg">Filled Button:</h4>
      <button appButtonFilled color="yellow">Click Me</button>
    </li>
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">...</li>
  </ul>
</main>
```

1. Notice that we've removed the `fill` attribute from the element.
2. Let's update the `ButtonOutlined` directive as well. We'll modify the `button-outlined.directive.ts` as follows:

```
import { Directive, HostBinding } from '@angular/core';
import { ButtonDirective } from './button.directive';
@Directive({
  selector: '[appButtonOutlined]',
  hostDirectives: [
    {
      directive: ButtonDirective,
      // eslint-disable-next-line @angular-eslint/no-inputs-metadata-property
      inputs: ['color'],
    },
  ],
})
export class ButtonOutlinedDirective {
  @HostBinding('attr.fill')
  fill = 'outlined';
}
```

1. Let's also modify the `ButtonWithTooltipDirective` class. We'll update the `button-with-tooltip.directive.ts` as follows:

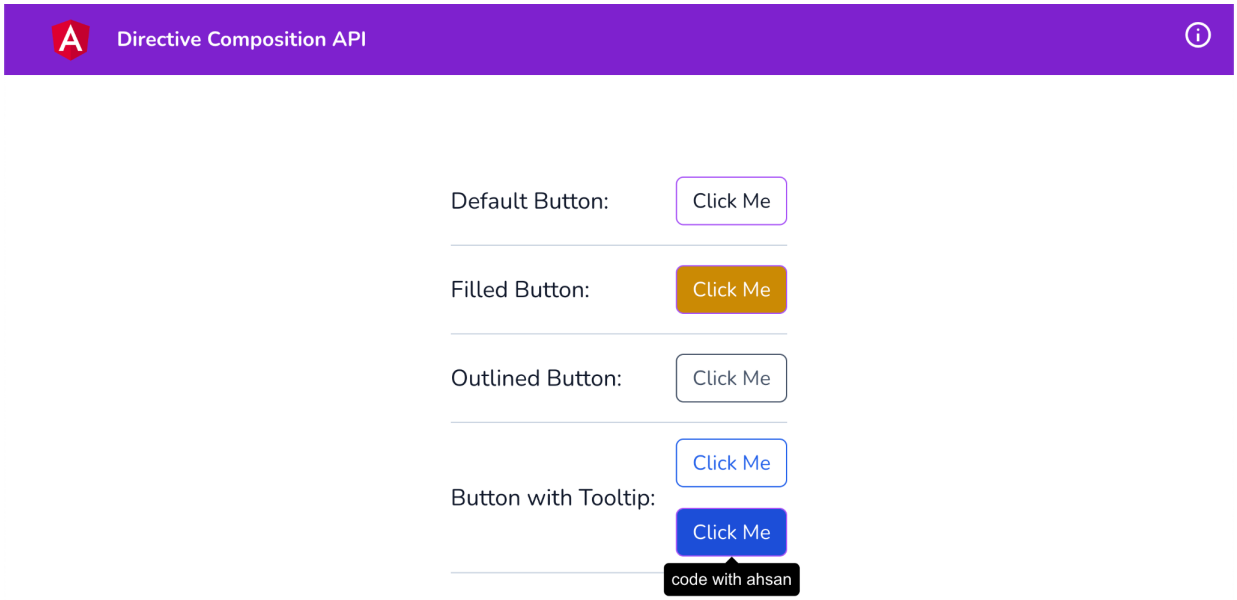
```
import { Directive } from '@angular/core';
import { ButtonDirective } from './button.directive';
import { TooltipDirective } from './tooltip.directive';
@Directive({
  selector: '[appButtonWithTooltip]',
  hostDirectives: [
    {
      directive: ButtonDirective,
      // eslint-disable-next-line @angular-eslint/no-inputs-metadata-property
      inputs: ['color', 'fill'],
    },
    {
      directive: TooltipDirective,
      // eslint-disable-next-line @angular-eslint/no-inputs-metadata-property, @angular-eslint/n
      inputs: ['appTooltip: tooltip'],
    },
  ],
})
export class ButtonWithTooltipDirective {}
```

You will notice that the app starts throwing an error that `TooltipDirective` is not a standalone component. That's true. We need to do the same thing we did for the `ButtonDirective` in step 2 and step 3 for the `TooltipDirective` as well. Move to the next step once you've done that.

1. Now update the `app.component.html` file to use both the `appButtonOutlined` and `appButtonTooltip` directives as follows:

```
...
<main class="content" role="main">
  <ul class="flex flex-col">
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">
      <h4 class="text-lg">Outlined Button:</h4>
      <button appButtonOutlined>Click Me</button>
    </li>
    <li class="flex gap-4 items-center border-b justify-between border-slate-300 py-3">
      <h4 class="text-lg">Button with Tooltip:</h4>
      <div class="flex flex-col gap-4">
        <button appButtonWithTooltip tooltip="code with ahsan" fill="outlined" color="blue">
          Click Me
        </button>
        <button appButtonWithTooltip tooltip="code with ahsan" fill="filled" color="blue">
          Click Me
        </button>
      </div>
    </li>
  </ul>
</main>
```

If you've followed all the steps correctly, you should be able to see the final result as follows:



How it works...

Directive Composition API was introduced in Angular v15 and has been one of the most requested features from the Angular community. In this recipe, we tried to create some components that bind the directives to the component directly in the component's typescript classes rather than in the template. This eliminates the need to create a wrapper element within the components to then apply the directives, and to then have to map the inputs of the components to the inputs of the directives. This also allows multiple directives to be bound to the same component and even if they could have inputs with the same names, we could alias them differently. The key to using the Directive Composition API is to construct your directives with the `standalone: true` flag. Which means your directives aren't part of any `NgModule` and are standalone. This is why we make both the `ButtonDirective` and the `TooltipDirective` standalone in steps 2, 3, and 7. Then we use those directives into the `ButtonFilledDirective`, `ButtonOutlinedDirective`, and `ButtonWithTooltipDirective` to be able to reuse the logic without having to create any wrapper component or additional HTML. We do it using the `hostDirectives` property in the directive metadata. Notice that we pass an array of objects to this property. And each object can contain the `directive` property which takes the class of the Directive to be applied. And we can also provide `inputs` and `outputs` for the host bindings. As you saw for the `ButtonWithTooltipDirective`, we also aliased the `appTooltip` input of the `TooltipDirective` to the `tooltip` input of the `ButtonWithTooltipDirective`. One thing to notice is that if you don't want to map any inputs or outputs and just want to bind a directive in the `hostDirectives`, you can just provide an array of the classes of the directives to be applied as follows:

```
hostDirectives: [
    ButtonDirective,
    TooltipDirective
      ],
```

There's more...

- **Directive Composition API** documentation (https://angular.io/guide/directive-composition-api#directive-composition-api)
- **Standalone Components** (https://angular.io/guide/standalone-components)

3 The Magic of Dependency Injection in Angular

Join our book community on Discord

https://packt.link/EarlyAccess

This chapter is all about the magic of **dependency injection** (**DI**) in Angular. Here, you'll learn some detailed information about the concept of DI in Angular. DI is the process that Angular uses to inject different dependencies into components, directives, and services. You'll work with several examples using services and providers to get some hands-on experience that you can utilize in your later Angular projects.In this chapter, we're going to cover the following recipes:

- Configuring an injector with a DI token
- Optional dependencies
- Creating a singleton service using `providedIn`
- Creating a singleton service using `forRoot()`
- Providing different services to the app with the same Aliased class provider
- Value providers in Angular

Technical requirements

For the recipes in this chapter, ensure you have **Git** and **NodeJS** installed on your machine. You also need to have the `@angular/cli` package installed, which you can do so using `npm install -g @angular/cli` from your Terminal. The code for this chapter can be found at https://github.com/PacktPublishing/Angular-Cookbook/tree/master/chapter03.

Using Angular DI Tokens

In this recipe, you'll learn how to create a basic DI token for a regular TypeScript class to be used as an Angular service using Dependency Injection. We have a class named `Jokes` in our application, which is used in the `AppComponent` by manually creating a new instance of the class. This makes our code tightly coupled and hard to test since the `AppComponent` class directly uses the `Jokes` class. Since Angular is all about **DI** and **services**, we'll use a DI token to use the `Jokes` class as an Angular service. We'll use the `InjectionToken` method to create a DI token and then the `@Inject` decorator to enable us to use the class in our service.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-di-token` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-di-token` to serve the project

This should open the app in a new browser tab and you should see the following:

Figure 3.1 – The ng-di-token app running on http://localhost:4200

Now that we have the app running, we can move on to the steps for the recipe.

How to do it...

The app we have right now shows a greeting message to a random user that has been retrieved from our `UserService`. And `UserService` uses the `Greeter` class as it is. Instead of using it as a class, we'll use it as an Angular service using DI. We'll start by creating an `InjectionToken` for our `Greeter` class, which is a regular TypeScript class, and then we'll inject it into our services. Perform these steps to follow along:

1. We'll create an `InjectionToken` in the `jokes.class.ts` file. We'll name the token `'Jokes`, using a new `InjectionToken` instance. Finally, we'll export this token from the file:

```
import { InjectionToken } from '@angular/core';
export const JOKES = new InjectionToken('Jokes', {
  providedIn: 'root',
  factory: () => new Jokes(),
});
class Jokes {...}
export default Jokes;
```

1. Now, we'll use the `inject` method from the `@angular/core` package and the `JOKES` token from `the jokes.class.ts file` so to use the class as follows:

```
import { Component, inject, OnInit } from '@angular/core';
import { JOKES } from './classes/jokes.class';
import { IJoke } from './interfaces/joke.interface';
@Component({...})
export class AppComponent implements OnInit {
  joke!: IJoke;
  jokes = inject(JOKES);
  ...
}
```

And that's it. You should see the app works the same as before. The only difference is that instead of instantiating the instance of the `Jokes` class manually ourselves, we're relying on the `InjectionToken` to instantiate it. Now that we know the recipe, let's take a closer look at how it works.

How it works

Angular doesn't recognize regular TypeScript classes as injectables. However, we can create our own injection tokens and use the `inject` method from the `@angular/core` package to inject the relevant classes & values wherever necessary. Angular recognizes these tokens behind the scenes and finds their corresponding definition, which is usually in the form of a `factory` function. Notice that we're using `providedIn: 'root'` within the token definition. This means that there will be only one instance of the class in the entire application.

See also

Dependency Injection in Angular (https://angular.io/guide/dependency-injection) `InjectionToken` documentation (https://angular.io/api/core/InjectionToken)

Optional dependencies

Optional dependencies in Angular are really powerful when you use or configure a dependency that may or may not exist or that has been provided within an Angular application. In this recipe, we'll learn how to use the `@Optional` decorator to configure optional dependencies in our components/services. We'll work with `LoggerService` and ensure our components do not break if it has not already been provided.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-optional-dependencies` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-optional-dependencies` to serve the project

This should open the app in a new browser tab and you should see the following:

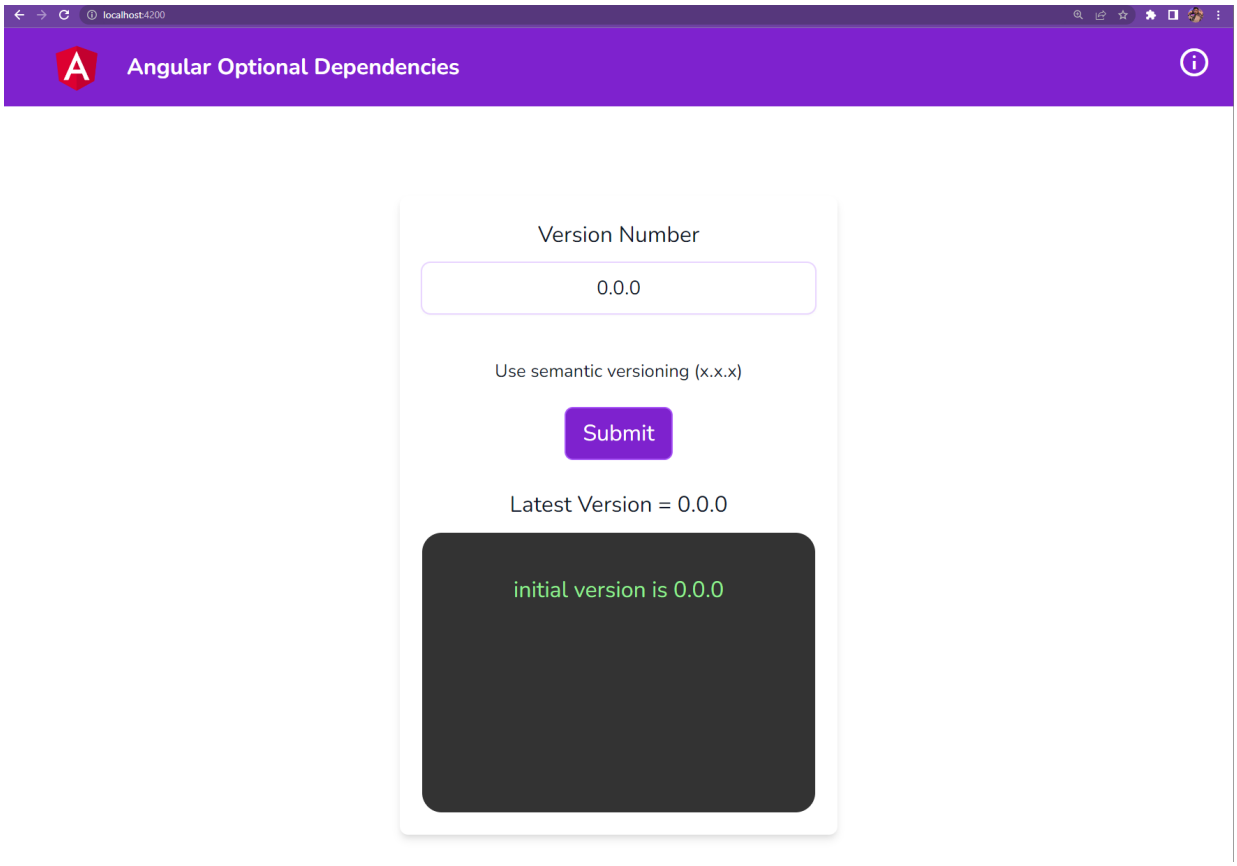


Figure 3.2 – The ng-optional-dependencies app running on http://localhost:4200

Now that we have the app running, we can move on to the steps for the recipe.

How to do it

We'll have an app containing a `LoggerService` which is provided using `providedIn: 'root'` as its injectable configuration. We'll see what happens when we don't provide this service anywhere. Then, we'll identify and fix the issues using the `@Optional` decorator. Follow these steps:

1. First, let's run the app, enter a new version number, and hit the `Submit` button.
2. This will result in the logs being saved in `localStorage` via `LoggerService`. Open **Chrome Dev Tools**, navigate to **Application**, select **Local Storage**, and then click on `localhost:4200`. You will see the key `vs_logs_ng_od` with log values, as follows:

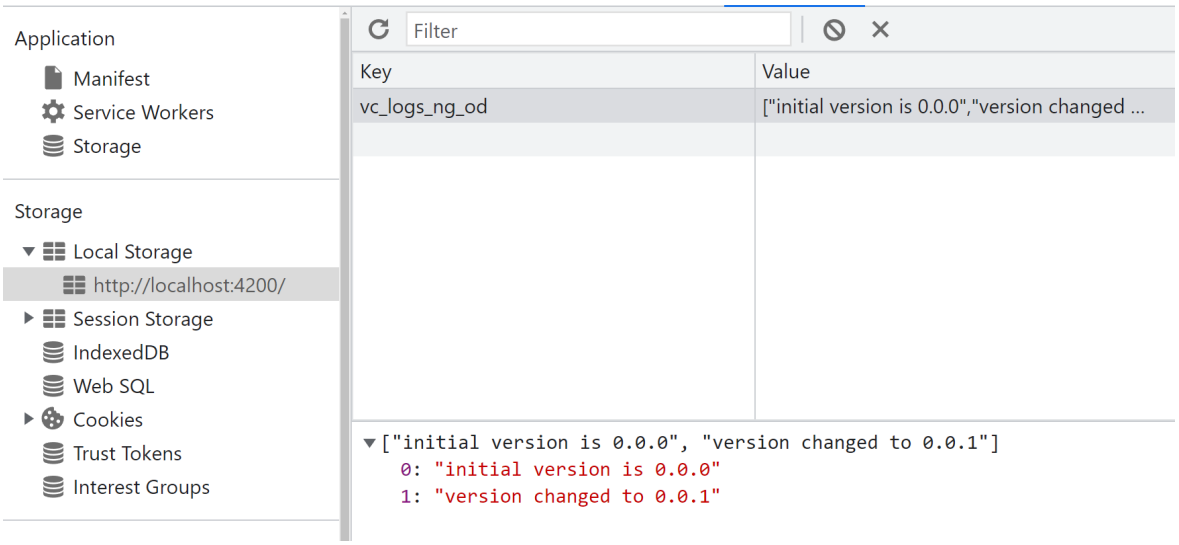


Figure 3.3 – The logs are saved in localStorage for http://localhost:4200

3. Let's try to remove the configuration provided in the `@Injectable` decorator for `LoggerService` in the `logger.service.ts` file. The change should be as follows:

```
import { Injectable } from '@angular/core';
import { Logger } from '../interfaces/logger';
@Injectable({ // <-- remove this object
  providedIn: 'root'
})
export class LoggerService implements Logger {
  ...
}
```

This will result in Angular not being able to recognize it and to throw an error in the console as follows:

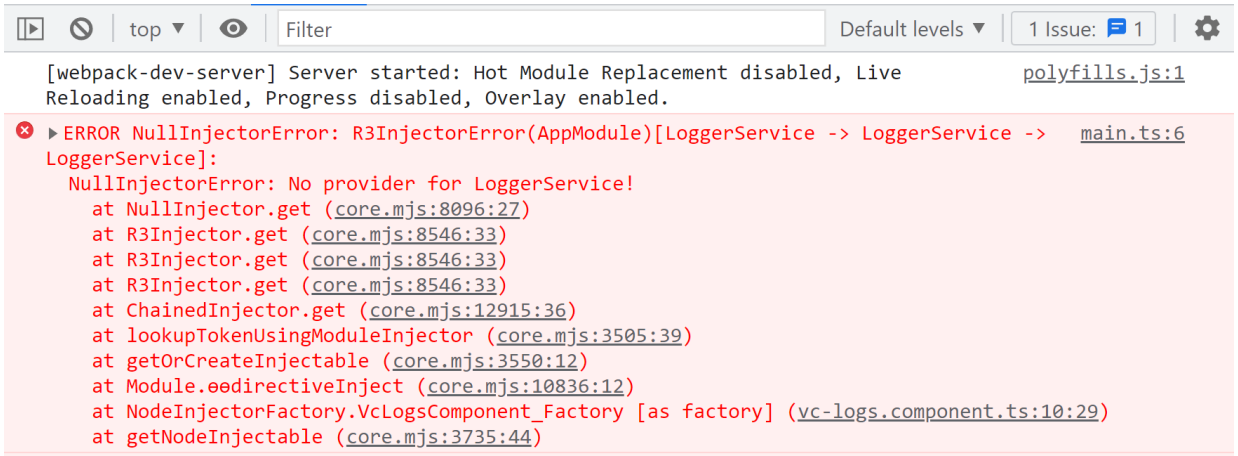


Figure 3.4 – An error reflecting that Angular doesn't recognize the LoggerService

1. We can now use the `@Optional` decorator to mark the dependency as optional. Let's import it from the `@angular/core` package and use the decorator in the constructor of `VcLogsComponent` in the `vc-logs.component.ts` file as follows:

```
import { Component, OnInit, Input, OnChanges, SimpleChanges, Optional } from '@angular/core';
...
export class VcLogsComponent implements OnInit {
```

```
  ...
  constructor(@Optional() private logger: LoggerService) {
    this.logs = this.logger?.retrieveLogs() || [];
  }
  ...
}
```

Great! Now if you refresh the app and view the console, there should be a different error. Yayy, progress :) .

1. You'll see that now it throws the following error because we're trying to call the `retrieveLogs` method of an optional Angular Service:

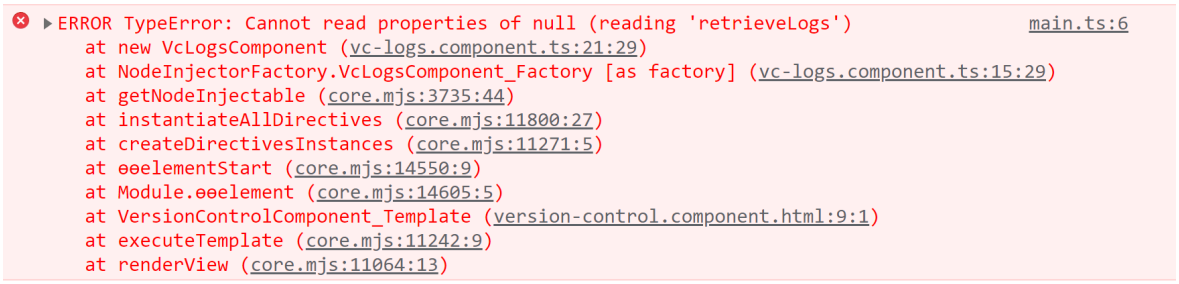


Figure 3.5 – An error detailing that this.logger is essentially null at the moment

2. To fix this issue, we can either decide not to log anything at all, or we can fall back to the `console.*` methods if `LoggerService` is not provided. The code to fall back to the `console.*` methods is as follows:

```
...
export class VcLogsComponent implements OnInit {
  ...
  constructor(@Optional() private loggerService:   LoggerService) {
    this.logs = this.logger?.retrieveLogs() || [];
  }
  get log() {
    return this.logger?.log.bind(this.logger) || console.log;
  }
  ...
```

1. Let's also update the `ngOnChanges` block to use this `log` function:

```
...
export class VcLogsComponent implements OnInit {
  ...
  constructor(@Optional() private logger: LoggerService) { }
  get log() {}
  ngOnChanges(changes: SimpleChanges) {
    const currValue = changes['vName'].currentValue;
    let message;
    if (changes['vName'].isFirstChange()) {
      message = `initial version is ${currValue.trim()}`;
      if (!this.logs.length) {
        this.log(message);
        this.logs.push(message);
      }
    } else {
      message = `version changed to ${currValue.trim()}`;
      this.log(message);
      this.logs.push(message);
    }
  }
  ...
```

1. Now, if you update the version and hit `Submit` , you should see the logs on the console, as follows:

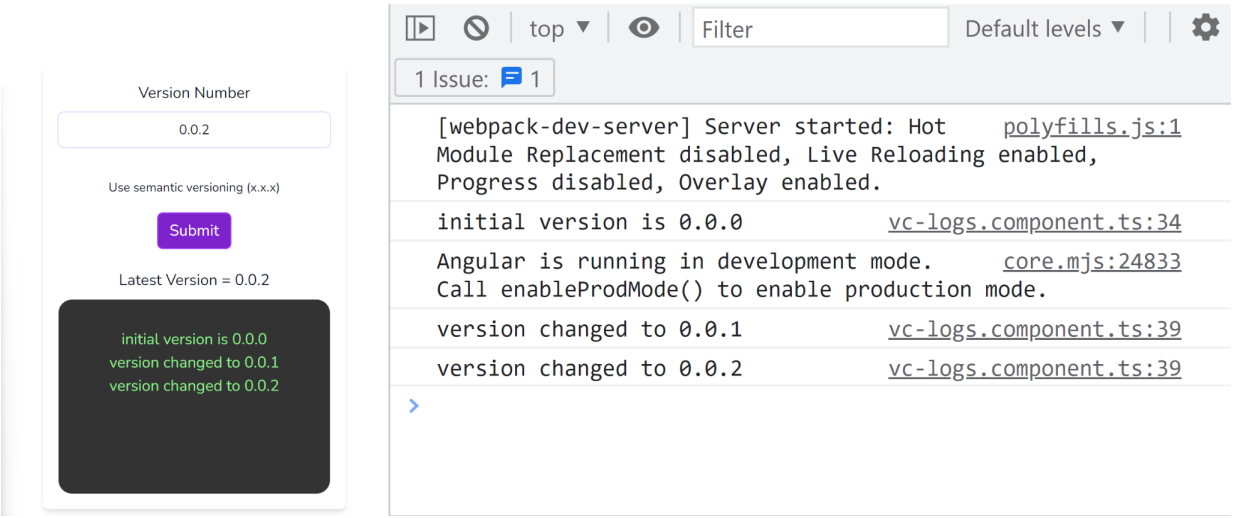


Figure 3.6 – The logs being printed on the console as a fallback to LoggerService not being provided

Great! We've finished the recipe and everything looks great. Please refer to the next section to understand how it works.

How it works

The `@Optional` decorator is a special parameter from the `@angular/core` package, which allows you to mark a dependency as optional. Behind the scenes, Angular will provide the value as `null` when the dependency doesn't exist or is not provided to the app. Since we remove the configuration object from the `@Injectable()` decorator from the `LoggerService` class, it isn't provided in Angular for Dependency Injection. As a result, our @Optional() decorator makes it as `null` when injected and doesn't cause Angular to throw the `NullInjectorError` shown in Figure 3.4. In step 4, we create a `log` getter function in our component's class `VcLogsComponent` so we can use the `LoggerService`'s method named `log` when the service is provided; and `console.log` otherwise. Then in the following steps, we just use the `log` method we created. If you go back to the `logger.service.ts` file and provide the service as `providedIn: 'root'` again , you won't see any console logs now and will see that now the app uses the service. I.e. the `LoggerService` that uses the `localStorage`.

See also

Optional Dependencies in Angular (https://angular.io/guide/dependency-injection#optional-dependencies)Hierarchical Injectors in Angular (https://angular.io/guide/hierarchical-dependency-injection)

Creating a singleton service using providedIn

In this recipe, you'll learn several tips on how to ensure your Angular service is being used as a singleton. This means that there will only be one instance of your service in the entire application. Here, we'll use a couple of techniques, including the `providedIn: 'root'` statement and making sure we only provide the service once in the entire app by using the `@Optional()` and `@SkipSelf()` decorators.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-singleton-service` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-singleton-service` to serve the project

This should open the app in a new browser tab and you should see the following:

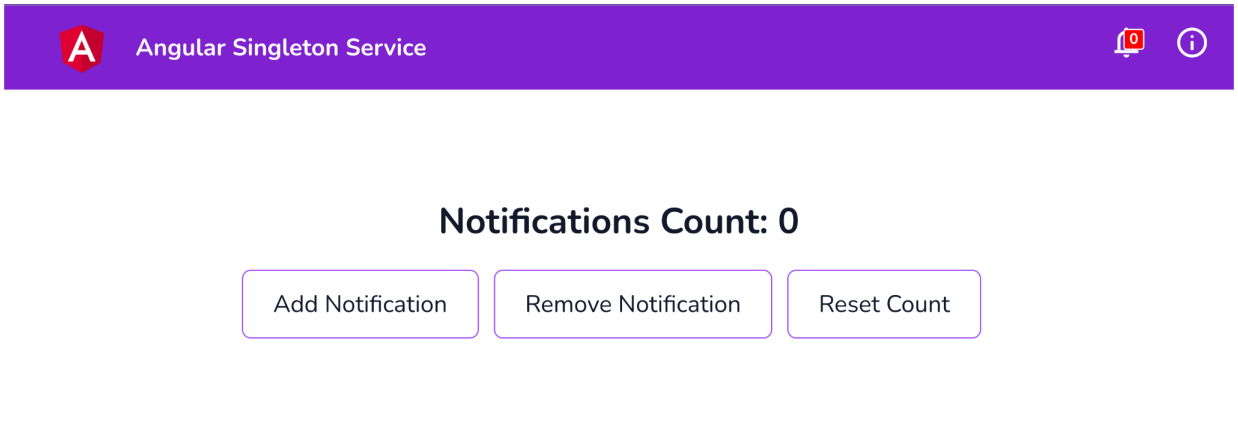


Figure 3.7 – The ng-singleton-service app running on http://localhost:4200

Now that we have the app running, we can move on to the steps for the recipe.

How to do it

The problem with the app is that if you add or remove any notifications, the count on the bell icon in the header does not change. That's due to us having multiple instances of `NotificationsService` provided in the `AppModule` and `HomeModule` classes. Please refer to the following steps to ensure we only have a single instance of the service in the app:

1. We will use `providedIn: 'root'` for the `NotificationService` to tell Angular that it is only provided in the root module, and it should only have one instance in the entire app. So, let's go to `notifications.service.ts` and pass `providedIn: 'root'` in the `@Injectable` decorator parameters, as follows:

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class NotificationsService {
  ...
}
```

Great! Now even if you refresh and try adding or removing notifications, you'll still see that the count in the header doesn't change. "But why is this, Ahsan?" Well, I'm glad you asked. That's because we're still providing the service in `AppModule` as well as in `HomeModule`.

1. First, let's remove `NotificationsService` from the `providers` array in `app.module.ts`, as highlighted in the following code block:

```
...
import { NotificationsButtonComponent } from './components/notifications-button/notifications-but
import { NotificationsService } from './services/notifications.service'; // <-- Remove this
@NgModule({
  declarations: [... ],
  imports: [...],
  providers: [
```

```
    NotificationsService // <-- Remove this
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1. Now, we'll remove `NotificationsService` from `home.module.ts`, as highlighted in the following code block:

```
...
import { NotificationsService } from '../services/notifications.service'; <-- Remove this
@NgModule({
  declarations: [...],
  imports: [...],
  providers: [
    NotificationsService  // <-- Remove this
  ]
})
export class HomeModule { }
```

Awesome! Now you should be able to see the count in the header change according to whether you add/remove notifications. However, what happens if someone still provides it in another lazily loaded module by mistake?

1. Let's put `NotificationsService` back in the `home.module.ts` file:

```
...
import { NotificationsService } from '../services/notifications.service';
@NgModule({
  declarations: [HomeComponent, NotificationsManagerComponent],
  imports: [CommonModule, HomeRoutingModule],
  providers: [NotificationsService],
})
export class HomeModule {}
```

Boom! We don't have any errors on the console or during compile time. However, we do have the issue of the count not updating in the header. So, how do we alert the developers if they make such a mistake? Please refer to the next step.

1. In order to alert the developer about potential duplicate providers, use the `@SkipSelf` decorator from the `@angular/core` package in our `NotificationsService`, and throw an error to notify and modify `NotificationsService`, as follows:

```
import { Injectable, SkipSelf } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
@Injectable({
  providedIn: 'root',
})
export class NotificationsService {
  ...
  constructor(@SkipSelf() existingService: NotificationsService) {
    if (existingService) {
      throw Error(
        `The service has already been provided in the app.
        Avoid providing it again in child  modules`
      );
    }
  }
  …
}
```

With the previous step now complete, you'll notice that we have a problem. That is we have failed to provide `NotificationsService` to our app at all. You should see this in the console:

```
[webpack-dev-server] Server started: Hot Module Replacement disabled, Live Reloading enabled
enabled.
ERROR NullInjectorError: R3InjectorError(AppModule)[NotificationsService ->
NotificationsService -> NotificationsService -> NotificationsService]:
  NullInjectorError: No provider for NotificationsService!
    at NullInjector.get (core.mjs:8096:27)
    at R3Injector.get (core.mjs:8546:33)
    at R3Injector.get (core.mjs:8546:33)
    at R3Injector.get (core.mjs:8546:33)
    at injectInjectorOnly (core.mjs:738:33)
    at Module.ɵɵinject (core.mjs:742:60)
    at Object.NotificationsService_Factory [as factory] (notifications.service.ts:7:34)
    at R3Injector.hydrate (core.mjs:8647:35)
    at R3Injector.get (core.mjs:8535:33)
    at ChainedInjector.get (core.mjs:12915:36)
```

Figure 3.8 – An error detailing that NotificationsService can't be injected into NotificationsService

1. The reason for this is that `NotificationsService` is now a dependency of `NotificationsService` itself. This can't work as it has not already been resolved by Angular. To fix this, we'll also use the `@Optional()` decorator in the next step.

All right, now we'll use the `@Optional()` decorator in `notifications.service.ts`, which is in the constructor for the dependency alongside the `@SkipSelf` decorator. The code should appear as follows:

```
import { Injectable, Optional, SkipSelf } from '@angular/core';
...
export class NotificationsService {
  ...
  constructor(@Optional() @SkipSelf() existingService:   NotificationsService) {
    if (existingService) {
      throw Error ('The service has already been provided       in the app. Avoid providing it a
    }
  }
  ...
}
```

1. We have now fixed the `NotificationsService -> NotificationsService` dependency issue. You should see the proper error for the `NotificationsService` being provided multiple times in the console, as follows:

```
Angular is running in development mode. Call enableProdMode() to enable production mode.
ERROR Error: Uncaught (in promise): Error: The service has already been provided in the app.
       Avoid providing it again in child  modules
Error: The service has already been provided in the app.
       Avoid providing it again in child  modules
    at new NotificationsService (notifications.service.ts:13:13)
    at Object.NotificationsService_Factory [as factory] (notifications.service.ts:7:34)
    at R3Injector.hydrate (core.mjs:8647:35)
    at R3Injector.get (core.mjs:8535:33)
    at ChainedInjector.get (core.mjs:12915:36)
    at lookupTokenUsingModuleInjector (core.mjs:3505:39)
    at getOrCreateInjectable (core.mjs:3550:12)
    at ɵɵdirectiveInject (core.mjs:10836:12)
    at ɵɵinject (core.mjs:742:60)
    at inject (core.mjs:825:12)
    at resolvePromise (zone.js:1211:31)
    at resolvePromise (zone.js:1165:17)
    at zone.js:1278:17
```

Figure 3.9 – An error detailing that NotificationsService is already provided in the app

2. Now, we'll safely remove the provided `NotificationsService` from the `providers` array in the `home.module.ts` file as shown in step 3 and check whether the app is working correctly

Bam! We now have a singleton service using the `providedIn` strategy. In the next section, let's discuss how it works.

How it works

Whenever we try to inject a service somewhere, by default, it tries to find a service inside the associated module of where you're injecting the service. When we use `providedIn: 'root'` to declare a service, whenever the service is injected anywhere in the app, Angular knows that it simply has to find the service definition in the root module and not in the feature modules or anywhere else.However, you have to make sure that the service is only provided once in the entire application. If you provide it in multiple modules, then even with `providedIn: 'root'`, you'll have multiple instances of the service. To avoid providing a service in multiple modules or at multiple places in the app, we can use the `@SkipSelf()` decorator with the `@Optional()` decorator in the services' constructor to check whether the service has already been provided in the app.

See also

Hierarchical Dependency Injection in Angular (https://angular.io/guide/hierarchical-dependency-injection)

Creating a singleton service using forRoot()

In this recipe, you'll learn how to use `ModuleWithProviders` and the `forRoot()` statement to ensure your Angular service is being used as a singleton in the entire app. We'll start with an app that has multiple instances of `NotificationsService`, and we'll implement the necessary code to make sure we end up with a single instance of the app.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-singleton-service-forroot` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-singleton-service-forroot` to serve the project

This should open the app in a new browser tab and you should see the following:

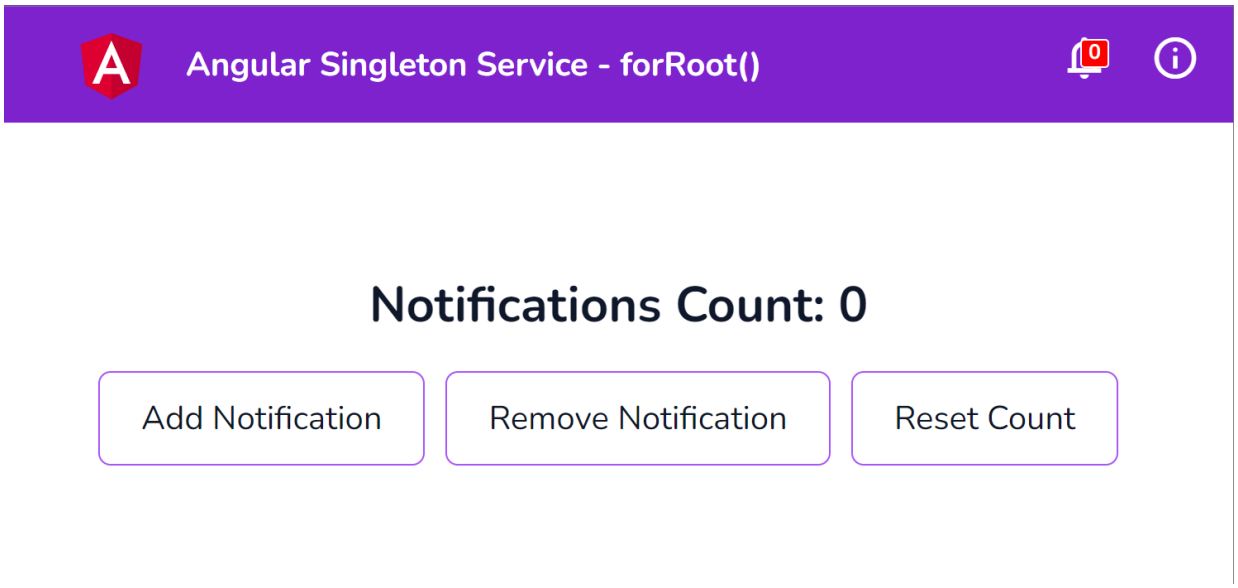


Figure 3.10 – The ng-singleton-service-forroot app running on http://localhost:4200

Now that we have the app running, in the next section, we can move on to the steps for the recipe.

How to do it

In order to make sure we only have a singleton service in the app with the `forRoot()` method, you need to understand how `ModuleWithProviders` and the `static forRoot()` method are created and implemented. Perform the following steps:

1. First, we'll make sure that the service has its own module. In many Angular applications, you'll probably see `CoreModule` where the services are provided (given we're not using the `providedIn: 'root'` syntax for some reason). To begin, we'll create a module, named `ServicesModule`, using the following command from the project root:

```
cd start && npx nx g m services --project ng-singleton-service-forroot
```

1. Let's create a static method `forRoot()` inside the `ServicesModule` class in the `services.module.ts` file. We'll name the method `forRoot` and return a `ModuleWithProviders` object that contains the `NotificationsService` provided in the `providers` array, as follows:

```
import { ModuleWithProviders, NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { NotificationsService } from './notifications.service';
@NgModule({
  declarations: [],
  imports: [CommonModule],
})
export class ServicesModule {
  static forRoot(): ModuleWithProviders<ServicesModule> {
    return {
      ngModule: ServicesModule,
      providers: [NotificationsService],
    };
  }
}
```

1. Now we'll remove the `NotificationsService` from the `app.module.ts` file's `imports` array and include `ServicesModule` in the `app.module.ts` file; in particular, we'll add in the `imports` array using the `forRoot()` method, as highlighted in the following code block.

2. This is because it injects `ServicesModule` with the providers in `AppModule`, for instance, with the `NotificationsService` being provided as follows:

```
...
import { NotificationsService } from './services/notifications.service'; // <-- Remove this
import { ServicesModule } from './services/services.module';
@NgModule({
  declarations: [...],
  imports: [
    ...,
    ServicesModule.forRoot()
  ],
  providers: [
    NotificationsService // <-- Remove this
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1. You'll notice that when adding/removing notifications, the count in the header still doesn't change. This is because we're still providing the `NotificationsService` in the `home.module.ts` file.
2. We'll remove the `NotificationsService` from the `providers` array in the `home.module.ts` file, as follows:

```
...
import { NotificationsService } from '../services/notifications.service'; //  <-- Remove this
import { ServicesModule } from '../services/services.module';
@NgModule({
  declarations: [HomeComponent, NotificationsManagerComponent],
  imports: [CommonModule, HomeRoutingModule, ServicesModule],
  providers: [
    NotificationsService // <-- Remove this
  ],
})
export class HomeModule {}
```

Great job. Now that we have finished the recipe, in the next section, let's discuss how it works.

How it works

`ModuleWithProviders` is a wrapper around `NgModule`, which is associated with the `providers` array that is used in `NgModule`. It allows you to declare `NgModule` with providers, so the module where it is being imported gets the providers as well. We created a `forRoot()` method in our `ServicesModule` class that returns `ModuleWithProviders` containing our provided `NotificationsService`. This allows us to provide `NotificationsService` only once in the entire app, which results in only one instance of the service in the app.

See also

The `ModuleWithProviders` Angular documentation (https://angular.io/api/core/ModuleWithProviders).The `ModuleWithProviders` migration documentation (https://angular.io/guide/migration-module-with-providers).

Providing alternate classes against same DI Token

In this recipe, you'll learn how to provide two different services to the app using `Aliased class providers`. This is extremely helpful in complex applications where you need to narrow down the implementation of the service/class for some components/modules. That is providing different classes against the same DI token to have a polymorphic behavior. Additionally, aliasing is used in component/service unit tests to mock the dependent service's actual implementation so that we don't rely on it.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-aliased-class-providers` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-aliased-class-providers` to serve the project

This should open the app in a new browser tab and you should see the app as shown in Figure 3.11

1. Click on the **Login as Admin** button. You should see something similar to the following screenshot:

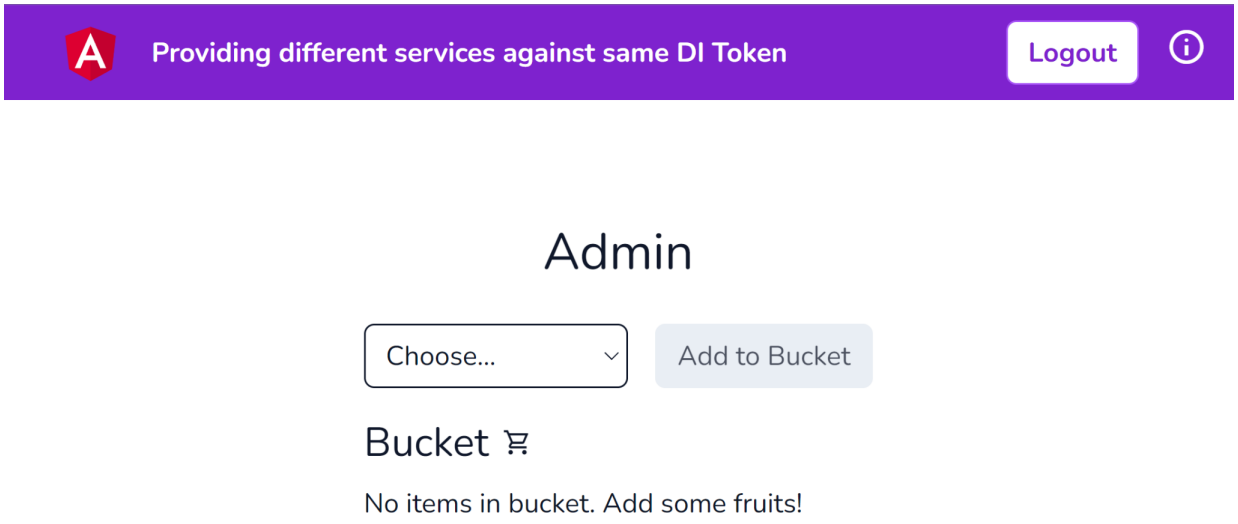


Figure 3.11 – The ng-aliased-class-providers app running on http://localhost:4200

Now that we have the app running, let's move to the next section to follow the steps for the recipe.

How to do it

We have a standalone component named `BucketComponent`, which is being used in both the admin and employee components. `BucketComponent` uses `BucketService` behind the scenes to add/remove items from and to a bucket. For the employee, we'll restrict the ability to remove an item by providing an `aliased` class provider and a different `EmployeeBucketService`. This is so that we can override the remove item functionality. Perform the following steps:

1. We'll start by creating `EmployeeBucketService` within the `employee` folder, as follows:

```
npx nx g service employee/employee-bucket --project ng-aliased-class-providers
```

1. Next, we'll extend `EmployeeBucketService` from `BucketService` so that we get all the goodness of `BucketService`. Let's modify the code as follows:

```
import { Injectable } from '@angular/core';
import { BucketService } from '../bucket/bucket.service';
```

```
...
export class EmployeeBucketService extends BucketService {
  constructor() {
    super();
  }
}
```

1. We will now override the `removeItem()` method to simply display a simple `alert()` mentioning that the employees can't remove items from the bucket. Your code should appear as follows:

```
...
export class EmployeeBucketService extends BucketService {
  constructor() {...}
  override removeItem() {
    alert('Employees can not delete items');
  }
}
```

1. As a final step, we need to provide the `aliased` class provider to the `employee.component.ts` file, as follows:

```
...
import { BucketService } from '../bucket/bucket.service';
import { EmployeeBucketService } from './employee-bucket.service';
@Component({
  ...
  providers: [{
    provide: BucketService,
    useClass: EmployeeBucketService,
  }],
})
export class EmployeeComponent {}
```

If you now log in as an employee in the app and try to remove an item, you'll see an alert pop up, which says `"Employees cannot delete items"`.

How it works

When we inject a service into a component, Angular tries to find that component inside the component/module we've provided the dependency in. And then by moving up the hierarchy of components and modules. Our `BucketService` is provided in `'root'` using the `providedIn: 'root'` syntax. Therefore, it resides at the top of the hierarchy. However, since, in this recipe, we use an `aliased` class provider in `EmployeeComponent` class against the DI token `BucketService`, when Angular searches for `BucketService` for the `EmployeeComponent`, it quickly finds the `EmployeeBucketService` against the token and stops there. I.e. it doesn't reach the 'root' to get the actual `BucketService`. And this is what we intended.

See also

Dependency Injection in Angular (https://angular.io/guide/dependency-injection)Hierarchical Injectors in Angular (https://angular.io/guide/hierarchical-dependency-injection)

Dynamic configurations using Value providers

In this recipe, you'll learn how to use value providers in Angular to provide constants and config values to your app. We'll start with the same example from the previous recipe, that is with the `EmployeeComponent` and `AdminComponent` using the `BucketComponent` to manage bucket of fruits. We will restrict the `EmployeeComponent` from deleting items from the bucket by using the configuration using value provider. As a result. the employees won't even see the `Delete` button.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-aliased-class-providers` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-aliased-class-providers` to serve the project

This should open the app in a new browser tab and you should see the app as shown in *Figure 3.11*

1. Click on the **Login as Admin** button. You should see something similar to the following screenshot:

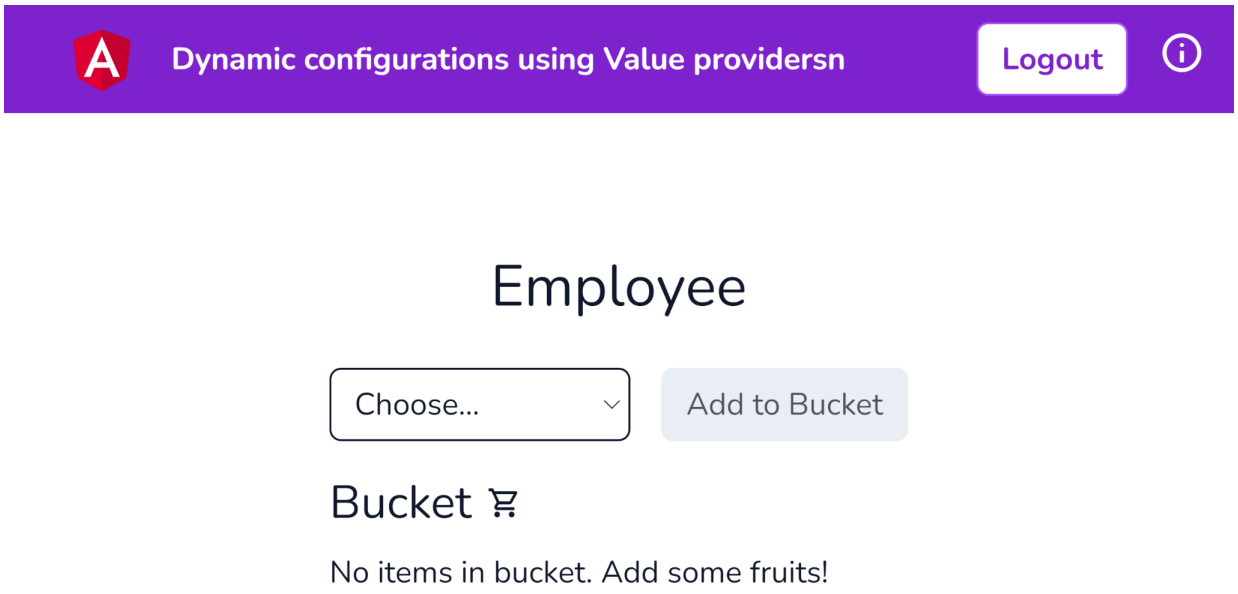


Figure 3.12 – The ng-value-providers app running on http://localhost:4200

Now that you see the app running, let's look at the next section to follow the recipe.

How to do it

1. First, we'll start by creating the value provider with `InjectionToken` within a new file, named `app-config.ts`, in the root of the project. The code should appear as follows:

```
import { InjectionToken } from '@angular/core';
export interface IAppConfig {
  canDeleteItems: boolean;
}
export const APP_CONFIG = new InjectionToken<IAppConfig>('APP_CONFIG');
export const AppConfig: IAppConfig = {
  canDeleteItems: true,
};
```

1. Before we can actually use this `AppConfig` constant in our `BucketComponent`, we need to register it to the `AppModule` so that when we inject this in the `BucketComponent`, the value of the provider is resolved.
2. Let's add the provider to the `app.module.ts` file, as follows:

```
...
import { AppConfig, APP_CONFIG } from './app-config';
```

```
@NgModule({
  declarations: [AppComponent],
  imports: [...],
  providers: [{
    provide: APP_CONFIG,
    useValue: AppConfig,
  }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

1. Now the app knows about the `AppConfig` constants. The next step is to use this constant in `BucketComponent`.
2. We'll use the `@Inject()` decorator to inject it inside the `BucketComponent` class, in the `bucket/bucket.component.ts` file, as follows:

```
import { Component, Inject, OnInit } from '@angular/core';
...
import { APP_CONFIG } from '../app-config';
...
export class BucketComponent implements OnInit {
  bucketService = inject(BucketService);
  appConfig = inject(APP_CONFIG);
  ...
}
```

Great! The constant has been injected. Now, if you refresh the app, you shouldn't get any errors. The next step is to use the `canDeleteItems` property from `config` in `BucketComponent` to show/hide the `delete` button.Now, we'll add an `*ngIf` directive in the `bucket/ bucket.component.html` file to only show the `delete` button if the value of `appConfig.canDeleteItems` is `true`. Update the element with the class `fruites__item__delete-icon` as follows:

```
...
<div *ngIf="appConfig.canDeleteItems" class="fruites__item__delete-icon" (click)="deleteFromBucke
  <div class="material-symbols-outlined">delete</div>
</div>
...
```

You can test whether everything works by setting the `AppConfig` constant's `canDeleteItems` property to `false`. Note that the `Delete` button is now hidden for both the admin and employee. Once tested, set the value of `canDeleteItems` back to `true` again.Now we have everything set up. Let's add a new constant so that we can hide the `delete` button for the employee only.

1. Let’s create an employee configuration object now. We’ll create an `employee.config.ts` file inside the `employee` folder. And we will add the following code to it:

```
import { IAppConfig } from '../app-config';
export const EmployeeConfig: IAppConfig = {
  canDeleteItems: false,
};
```

1. Now, we'll provide this `EmployeeConfig` constant to the `EmployeeComponent` for the same `APP_CONFIG` injection token. The code in the `employee.component.ts` file should look as follows:

```
...
import { APP_CONFIG } from '../app-config';
import { EmployeeConfig } from './employee.config';
@Component({
  ...
  providers: [{
    provide: APP_CONFIG,
    useValue: EmployeeConfig,
  }],
})
export class EmployeeComponent {}
```

And we're done! The recipe is now complete. You can see that the **Delete** button is visible to the admin but hidden for the employee. It's all thanks to the magic of value providers.

How it works

When we inject a token into a component, Angular tries to find the resolved value of the token in the injected place. And then by moving up the hierarchy of components and modules. We provided `EmployeeConfig` as `APP_CONFIG` in `EmployeeComponent`. When Angular tries to resolve its value for `BucketComponent`, it finds it early as `EmployeeConfig,` within the `EmployeeComponent` instead of the value provided in the `AppModule` as `AppConfig`. Therefore, Angular stops right there and doesn't reach `AppModule`. This is amazing as we can now have a global configuration and override the configuration within nested modules/components.

See also

- Dependency Injection in Angular (https://angular.io/guide/dependency-injection)
- Hierarchical Injectors in Angular (https://angular.io/guide/hierarchical-dependency-injection)

4 Understanding Angular Animations

Join our book community on Discord

https://packt.link/EarlyAccess

In this chapter, you'll learn about working with animations in Angular. You'll learn about multi-state animations, staggering animations, keyframe animations, how to implement animations for switching routes in your Angular apps and how to conditionally disable animations.The following are the recipes that we're going to cover in this chapter:

- Creating your first two-state Angular animation
- Working with multi-state animations
- Creating complex Angular animations using keyframes
- Animating lists in Angular using stagger animations
- Sequential vs Parallel animations in Angular
- Route animations in Angular
- Complex route animations in Angular using keyframes

Technical requirements

For the recipes in this chapter, make sure you have **Git** and **Node.js** installed on your machine. You also need to have the `@angular/cli` package installed, which you can install by using `npm install -g @angular/cli` from your terminal. The code for this chapter can be found at https://github.com/PacktPublishing/Angular-Cookbook/tree/master/chapter04.

Creating your first two-state Angular animation

In this recipe, you'll create a basic two-state Angular animation having a fading effect. We'll start with an Angular app with some UI already built into it. We'll then enable animations in the app using Angular Animations and will move towards creating our first animation.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-basic-animation` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-basic-animation` to serve the project

This should open the app in a new browser tab and you should see the following:

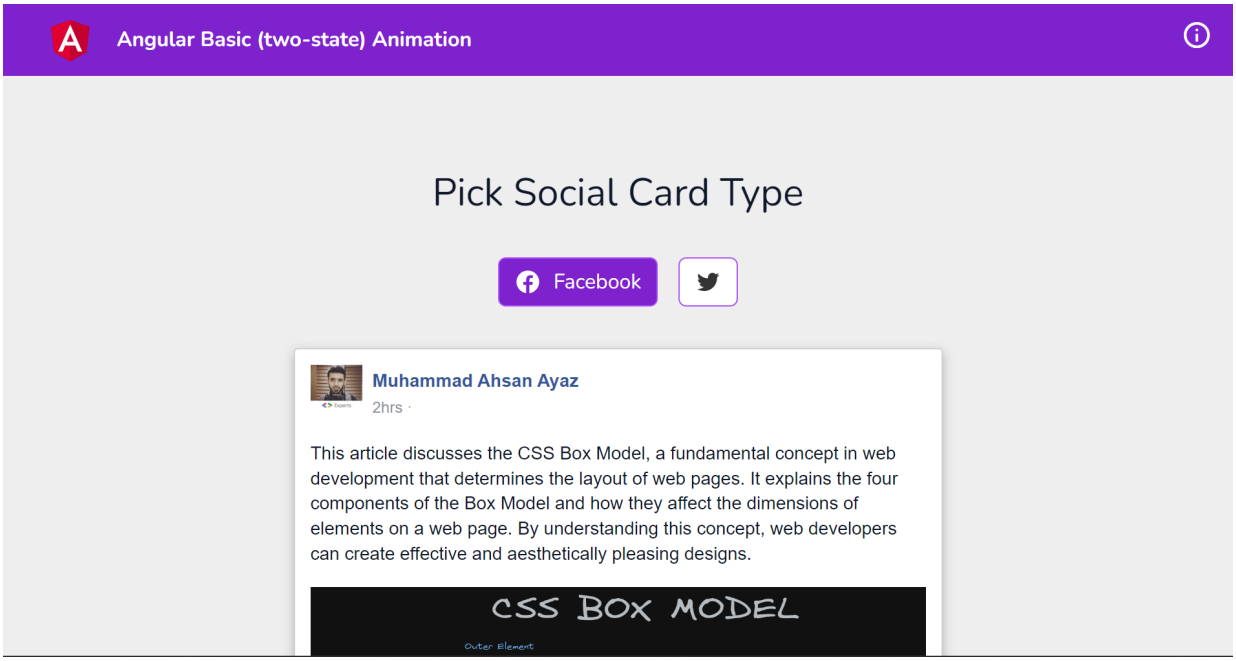


Figure 4.1 – ng-basic-animation app running on http://localhost:4200

Now that we have the app running, we will move on to the steps for the recipe.

How to do it...

We have an app that doesn't have Angular animations configured at all. We will create a fading effect for the cards using Angular animations. Let's continue with the steps as follows:

1. First, we'll import the `provideAnimations()` method from the `@angular/platform-browser/animations` package in our `app.component.ts` file so we can use animations in the app. We'll use it in the `providers` array as follows:

```
...
import { provideAnimations } from '@angular/platform-browser/animations';
import { AppComponent } from './app/app.component';
import { appRoutes } from './app/app.routes';
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation()),
    provideAnimations(),
  ],
}).catch((err) => console.error(err));
```

1. Now modify the `app.component.ts` file to add the animations as follows:

```
import { trigger, transition, style, animate } from '@angular/animations';
...
@Component({
  ...
  imports: [CommonModule, FbCardComponent, TwitterCardComponent],
  animations: [
    trigger(fadeInOut, [
      transition(':enter', [
        style({ opacity: 0 }),
        animate('300ms 100ms', style({ opacity: 1 })),
      ]),
      transition(':leave', [
        style({ opacity: 1 }),
```

```
        animate('100ms', style({ opacity: 0 })),
      ]),
    ]),
  ],
})
export class AppComponent {...}
```

1. Finally, add the `fadeInOut` animation on both the cards in the `app.component.html` file as follows:

```
  <!-- Toolbar -->
<div class="toolbar" role="banner">...</div>
<main class="content" role="main">
  <div class="type-picker mb-8">...</div>
  <ng-container [ngSwitch]="selectedCardType">
    <app-fb-card [@fadeInOut] *ngSwitchCase="'facebook'"></app-fb-card>
    <app-twitter-card [@fadeInOut] *ngSwitchCase="'twitter'"></app-twitter-card>
  </ng-container>
</main>        }
```

Great! You now have implemented a basic fade-in <=> fade-out animation for the cards. Simple, but pretty! See the next section to understand how the recipe works.

How it works...

Angular provides its own Animation API that allows you to animate any property that the CSS transitions work on. The benefit is that you can configure them dynamically based on the conditions required. We first used the `trigger()` method to register the animation named `fadeInOut`. Then we have the `:enter` and `:leave` transitions being registered with the `transition()` method. Finally, we define the styles and animations for those transitions using the `style()` and `animate()` methods. Note that we're using `'300ms 100ms..'` in the `:enter` transition. The `300ms` is the duration of the transition while `100ms` is the delay. We add this `delay` so we can wait for the `:leave` transition of the previously shown card to be finished before we can move to the `:enter` transition of the next card to be shown.

See also

- Animations in Angular (https://angular.io/guide/animations)
- *Angular Animations Explained with Examples* (https://www.freecodecamp.org/news/angular-animations-explained-with-examples/)

Working with multi-state animations

In this recipe, we'll work with Angular animations containing multiple states. This means that we'll work with more than two states for a particular item. We'll be using the same Facebook and Twitter cards example for this recipe as well. But we'll configure the state of the cards for their state before they appear on screen, when they're on screen, and when they're about to disappear from the screen again.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-multi-state-animations` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-multi-state-animations` to serve the project

This should open the app in a new browser tab and you should see the following:

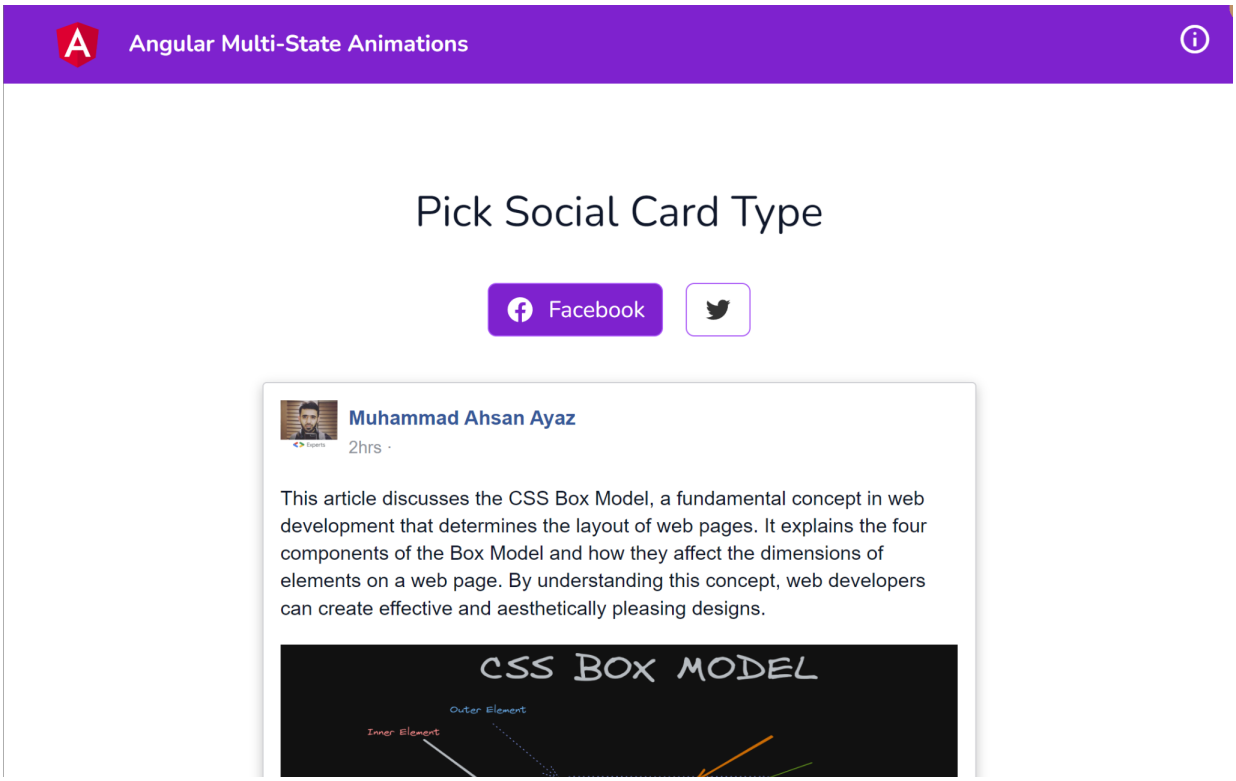


Figure 4.2 – ng-multi-state-animations app running on http://localhost:4200

Now that we have the app running locally, let's look at the steps of the recipe in the next section.

How to do it...

We already have a working app that has a single animation built for the reach of social cards. When you tap either the Facebook or Twitter button, you'll see the respective card appearing with a slide-in animation from left to right. To keep the recipe simple, we'll implement two more states and two animations for when the user moves the mouse cursor over the card and when the user moves away from the card. Let's add the relevant code in the following steps:

1. We'll start with adding two `@HostListener` instances to `FbCardComponent` in the `components/fb-card/fb-card.component.ts` file, one for the `mouseenter` event on the card and one for the `mouseleave` event. We'll name the states `hovered` and `active` respectively. The code should look as follows:

```
import { Component, HostListener } from '@angular/core';
...
@Component({...})
export class FbCardComponent {
  cardState = 'active';
  @HostListener('mouseenter')
  onMouseEnter() {
    this.cardState = 'hovered';
  }
  @HostListener('mouseleave')
  onMouseLeave() {
    this.cardState = 'active';
  }
}
```

1. Now, we'll do the same for `TwitterCardComponent` in the `twitter-card-component.ts` file. The code should look as follows:

```
import { Component, HostListener } from '@angular/core';
...
@Component({...})
export class TwitterCardComponent {
  cardState = 'active';
  @HostListener('mouseenter')
  onMouseEnter() {
    this.cardState = 'hovered';
  }
  @HostListener('mouseleave')
  onMouseLeave() {
    this.cardState = 'active';
  }
}
```

There should be no visual change so far since we're only updating the `cardState` variable to have the hover and active states. We haven't defined the transitions yet for the animation.

1. We'll now define our state for when the user's cursor enters the card, that is, the `mouseenter` event. The state is called `hovered` and should look as follows in the `animation.ts` file:

```
...
export const cardAnimation = trigger('cardAnimation', [
  state('active', style({
    color: 'rgb(51, 51, 51)',
    backgroundColor: 'white'
  })),
  state('hovered', style({
    transform: 'scale3d(1.05, 1.05, 1.05)',
    backgroundColor: '#333',
    color: 'white'
  })),
  transition('void => active', [...]),
])
```

If you refresh the app now, tap either the Facebook or Twitter button, and hover the mouse over the card, you'll see the card's UI changing. That's because we changed the state to `hovered`. However, there's no animation yet. Let's add one in the next step.

1. We'll add the `active => hovered` transition now in the `animations.ts` file so that we can smoothly navigate from `active` to the `hovered` state:

```
...
export const cardAnimation = trigger('cardAnimation', [
  state('active', style(...)),
  state('hovered', style(...)),
  transition('void => active', [...]),
  transition('active => hovered', [
    animate('0.3s 0s ease-out', style({
      transform: 'scale3d(1.05, 1.05, 1.05)',
      backgroundColor: '#333',
      color: 'white'
    }))
  ]),
])
```

1. You should now see the smooth transition on the `mouseenter` event if you refresh the app.

Finally, we'll add the final transition, `hovered => active`, so when the user leaves the card, we revert to the active state with a smooth animation. The code should look as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  state('active', style(...)),
```

```
  state('hovered', style(...)),
  transition('void => active', [...]),
  transition('active => hovered', [...]),
  transition('hovered => active', [
    animate('0.3s 0s ease-out', style({
      transform: 'scale3d(1, 1, 1)',
      color: 'rgb(51, 51, 51)',
      backgroundColor: 'white'
    }))
  ]),
])
```

Ta-da! You now know how to implement different states and different animations on a single element using `Angular Animations`.

How it works...

1. Angular uses triggers for understanding what state the animation is in. An example syntax looks as follows:

```
<div [@animationTriggerName]="expression">...</div>;
```

The `expression` can be a valid JavaScript expression, and it evaluates to the name of the state. In our case, we bind it to the `cardState` property, which either contains `'active'` or `'hovered'`. Therefore, we end up with three transitions for our cards:

- `void => active` (when the element is added to the DOM and is rendered)
- `active => hovered` (when the `mouseenter` event triggers on the card)
- `hovered => active` (when the `mouseleave` event triggers on the card)

See also

- *Triggering the animation* (https://angular.io/guide/animations#triggering-the-animation)
- *Reusable animations* (https://angular.io/guide/reusable-animations)

Creating complex Angular animations using keyframes

Since you already know about Angular animations from the previous recipes, you might be thinking, "Well, that's easy enough." Well, time to level up your animation skills in this recipe. You'll create a complex Angular animation using `keyframes` in this recipe to get started with writing some advanced animations.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-animations-keyframes` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-animations-keyframes` to serve the project

This should open the app in a new browser tab and you should see the following:

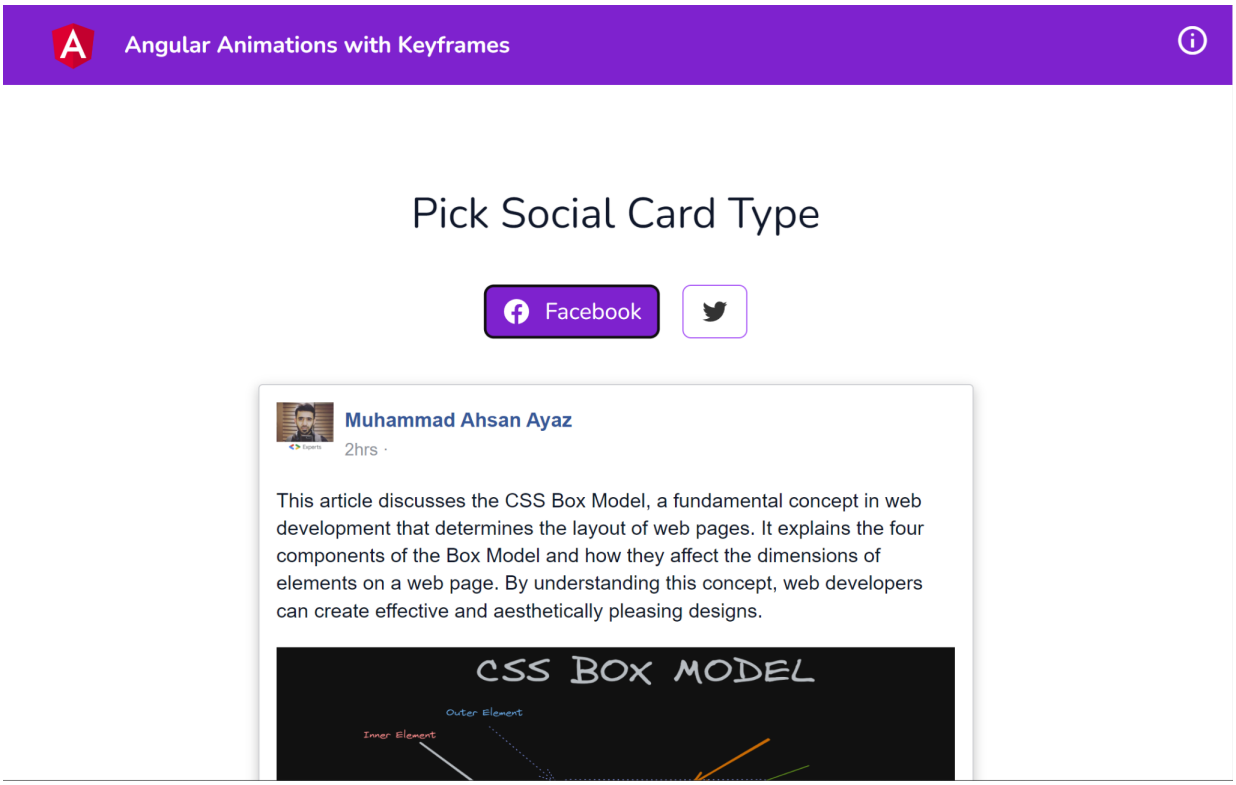


Figure 4.3 – ng-animations-keyframes app running on http://localhost:4200

Now that we have the app running locally, let's look at the steps of the recipe in the next section.

How to do it...

We have an app right now that has a single transition, that is, `void => active`, which triggers when the element enters the DOM. Right now, the animation is pretty simple. We'll use the `keyframes()` method to build a complex animation:

1. Let's begin with adding the `keyframes()` method from `@angular/animations` to the `animations.ts` file as follows:

```
import {
  …,
  keyframes
} from '@angular/animations';
...
```

1. Now, we'll convert the single style animation for the `void => transition` to use keyframes as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  ...,
  transition('void => active', [

      transform: 'translateX(-200px)',
      opacity: 0
    }),
    animate('0.2s ease', keyframes([
      style({
        transform: 'translateX(-200px)',
        offset: 0
```

```
      }),
      style({
        transform: 'translateX(0)',
        offset: 1
      })
    ]))
  ]),
])
```

If you refresh the app now and try it, you'll still see the same animation as before. But now we have it using `keyframes`.

1. Finally, let's start adding some complex animations. Let's start the animation with a scaled-down card by adding `scale3d` to the `transform` property of `style` at `offset: 0`. We'll also increase the animation time to `1.5s`:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('void => active', [
    animate('1.5s ease', keyframes([
      style({
        transform: 'translateX(-200px) scale3d(0.4, 0.4,         0.4)',
        offset: 0
      }),
      style({...})
    ]))
  ]),
])
```

You should now see that the card animation starts with a small card that slides from the left and moves toward the right, increasing in size.

1. Now we'll implement a *zig-zag-ish* animation for the appearance of the card instead of the slide-in animation. Let's add the following keyframe elements to the `keyframes` array to add a bumpy effect to our animation:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('void => *', [
    animate('1.5s 0s ease', keyframes([
      style({
        transform: 'translateX(-200px) scale3d(0.4, 0.4,         0.4)',
        offset: 0
      }),
      style({
        transform: 'translateX(0px) rotate(-90deg)         scale3d(0.5, 0.5, 0.5)',
        offset: 0.25
      }),
      style({
        transform: 'translateX(-200px) rotate(90deg)         translateY(0) scale3d(0.6, 0.6, 0.6)
        offset: 0.5
      }),
      style({
        transform: 'translateX(0)',
        offset: 1
      })
    ]))
  ]),
])
```

If you refresh the app and tap any of the buttons, you should see the card bumping to the right wall, and then to the left wall of the card, before returning to the normal state:

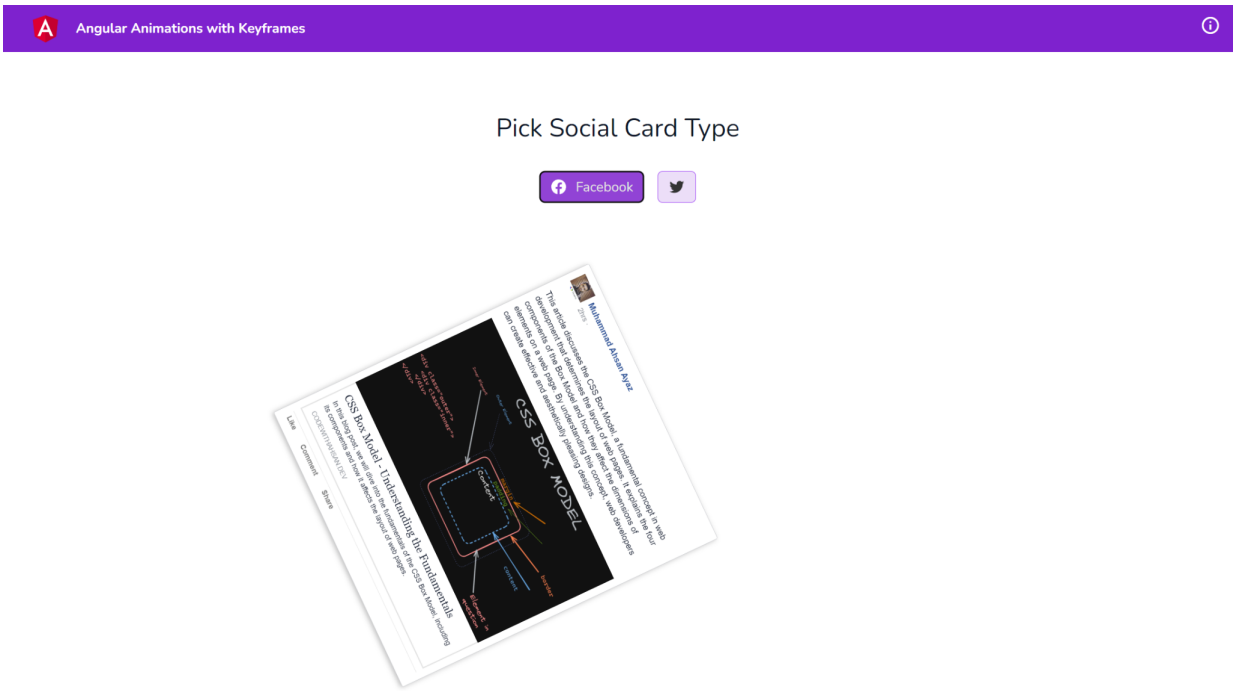


Figure 4.4 – Card bumping to right and then the left wall

1. As the last step, we'll spin the card clockwise before it returns to its original position. For that, we'll use `offset: 0.75`, using the `rotate` method with some additional angles. The code should look as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('void => *', [
    animate('1.5s 0s ease', keyframes([
      style({...}),
      style({...}),
      style({...}),
      style({
        transform: 'translateX(-100px) rotate(135deg)          translateY(0) scale3d(0.6, 0.6, 0.
        offset: 0.75
      }),
      style({...})
    ]))
  ]),
])
```

Awesome! You now know how to implement complex animations in Angular using the `keyframes` method Angular Animations. See in the next section how it works.

How it works...

For complex animations in Angular, the `keyframes` method is an amazing way of providing different timing offsets for the animation throughout its journey. We can define the offsets using the `styles` method, which takes `AnimationStyleMetadata` as a parameter. `AnimationStyleMetadata` also allows us to pass the `offset` property, which can have a value between `0` and `1`; reflecting the timing from `0%` to `100%` of the animation. Thus, we can define different styles for different offsets to create advanced animations.

See also

Animations in Angular (https://angular.io/guide/animations)*Angular Animations Explained with Examples* (https://www.freecodecamp.org/news/angular-animations-explained-with-examples/)

Animating lists in Angular using stagger animations

No matter what web application you build today, you are going to implement some sort of lists in them. And to make those lists even better, why not implement elegant animations for them? In this recipe, you'll learn how to animate lists in Angular using stagger animations.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-animating-lists` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-animating-lists` to serve the project

This should open the app in a new browser tab and you should see the following:

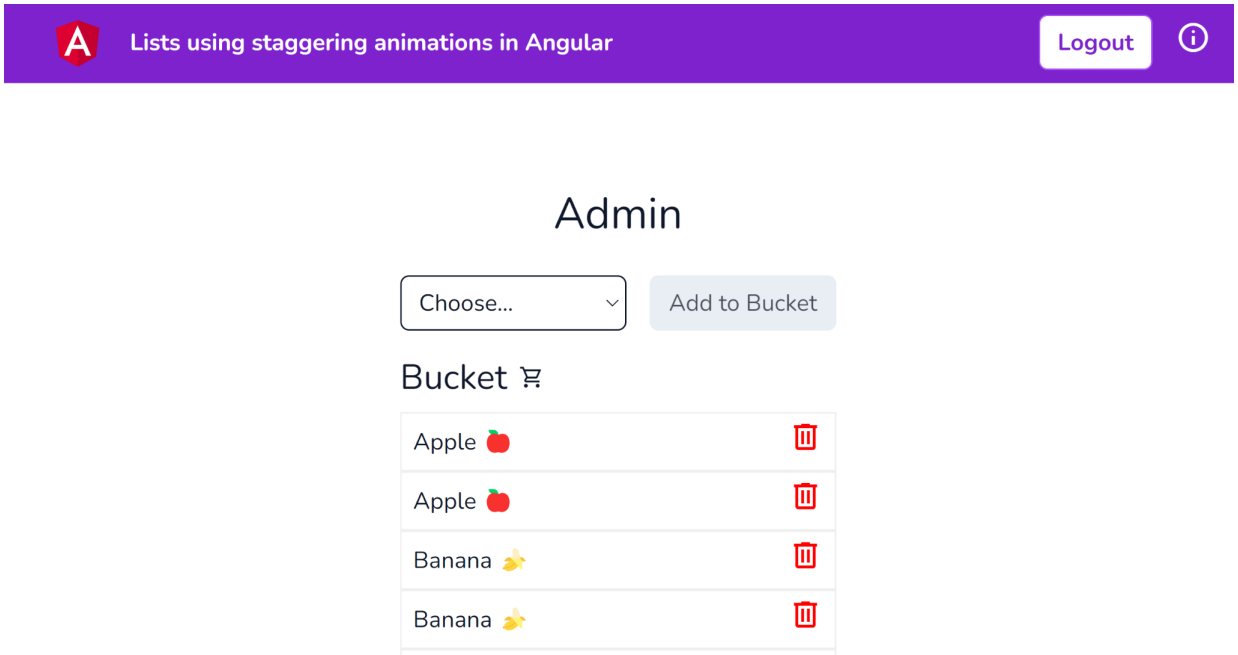


Figure 4.5 – ng-animating-lists app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an app right now that has a list of bucket items. We need to animate the list using staggering animations. We'll be doing this step by step. I'm excited – are you?Cool. We'll go through the following steps for the recipe:

1. First, let's provide Angular Animations using the `provideAnimations()` method from Angular in the `main.ts` file as follows:

```
import { bootstrapApplication } from '@angular/platform-browser';
...
```

```
import { provideAnimations } from '@angular/platform-browser/animations';
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation()),
    provideAnimations(),
  ],
}).catch((err) => console.error(err));
```

1. Now, create a file named `animations.ts` in the project's `app` folder and add the following code to register a basic list item animation named `listItemAnimation`:

```
import { trigger, style, animate, transition } from '@angular/animations';
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition(':enter', [
      style({ opacity: 0 }),
      animate('0.5s ease', style({ opacity: 1 })),
    ]),
    transition(':leave', [
      style({ opacity: 1 }),
      animate('0.5s ease', style({ opacity: 0 })),
    ]),
  ]),
};
```

1. Now, we'll add the animation to `BucketComponent` in the `app/bucket/bucket.component.ts` file as follows:

```
...
import { ANIMATIONS } from '../../../constants/animations';
@Component({
  ...
  animations: [ANIMATIONS.LIST_ITEM_ANIMATION]
})
```

Since we have the animation imported in the component, we can use it in the template now.

1. Let's add the animation to the html element having the class `fruits__item` in the `bucket.component.html` file as follows:

```
<div class="fruits__item" *ngFor="let item of bucket" @listItemAnimation>
  ...
</div>
```

If you now refresh the app and add an item to the bucket list, you should see it appear with a fade-in effect. And if you delete an item, you should see it disappear with the animation as well.

1. We'll modify `LIST_ITEM_ANIMATION` now to use the `stagger` method. This is because the staggering animation is applied on the list; and not the list items.. First, we need to import the `stagger` method from `@angular/animations`. Then we need create a wild card transition for the list as follows::

```
import {
  ...,
  stagger,
} from '@angular/animations';
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition('* <=> *', [
      ...
    ]),
  ]),
};
```

1. Now we'll add a `query` for what happens when a new item is added to the list. We'll use staggering animation here. The code should look as follows:

```
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition('* <=> *', [
      query(
        ':enter',
        [
          style({ opacity: 0 }),
          stagger(100, [
            animate('0.5s ease', style({ opacity: 1 }))
          ]),
        ],
        { optional: true }
      ),
    ]),
  ]),
};
```

1. Now we'll add the query for when an item leaves the list. The code should look as follows:

```
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition('* <=> *', [
      query(':enter', [...],
        { optional: true }
      ),
      query(
        ':leave',
        [
          style({ opacity: 1 }),
          animate('0.5s ease', style({ opacity: 0 }))
        ],
        { optional: true }
      ),
    ]),
  ]),
};
```

Now we can apply the animation to the list itself. Update the `bucket.component.html` as follows to put the animation on the `div` with class `fruits` instead:

```
<div class="buckets" *ngIf="$bucket | async as bucket">
  ...
  <div class="fruits" [@listItemAnimation]="bucket.length">
    ...
  </div>
</div>
...
```

Notice that we're binding the `[@listAnimation]` property to `bucket.length`. This will make sure that the animation triggers whenever the length of the bucket changes, that is, when an item is added or removed from the bucket.Awesome! You now know how to implement staggering animations for lists in Angular. See in the next section how it works.

How it works...

Stagger animations only work inside `query` methods and are applied on the list (containing the items) instead of the items themselves. In order to search or query the items, we first use the `query` method. Then we use the `stagger` method to define how many milliseconds of staggering we want before the animation starts for the next list item. We also use the `animation()` as well in the `stagger()` methods to define the animation for each element found in the query. Notice that we're using `{ optional: true }` for both the `:enter` query and the `:leave` query. This is because if the list binding changes (`bucket.length`), we don't get an error if no new element has entered the DOM or no element has left the DOM.

See also

Animations in Angular (https://angular.io/guide/animations)Angular animations stagger docs (https://angular.io/api/animations/stagger)

Sequential vs Parallel animations in Angular

In this recipe, you'll learn how to run angular animations in a sequence vs in parallel. This is handy for when we want to have one animation finished before we start the next one, or to run the animations simultaneously.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-seq-parallel-animations` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-seq-parallel-animations` to serve the project

This should open the app in a new browser tab and you should see the following:

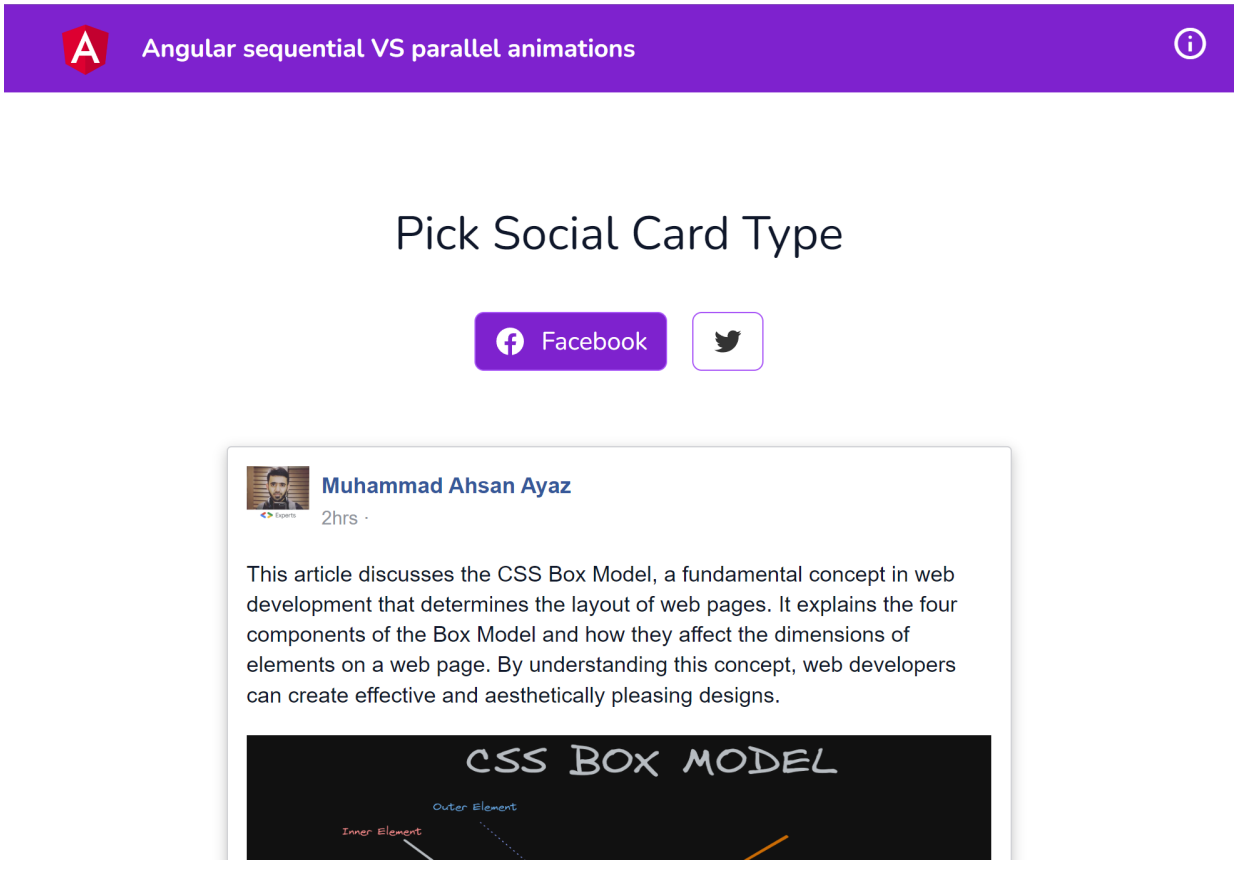


Figure 4.6 – ng-seq-parallel-animations app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an app that displays two cards we used in the previous recipes. To see how to use animation callbacks, we'll simply perform an action once the animation of a list item entering the DOM is finished and have one action for when a list item leaves the DOM. Let's get started:

1. First, let's provide Angular Animations using the `provideAnimations()` method from Angular in the `main.ts` file as follows:

```
import { bootstrapApplication } from '@angular/platform-browser';
...
import { provideAnimations } from '@angular/platform-browser/animations';
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation()),
    provideAnimations(),
  ],
}).catch((err) => console.error(err));
```

1. We will create a simple wrapper transition to handle the cards entering and leaving the dom. And afterwards we'll handle how they can be triggered together when the current card leaves the view. Create a new file named `animations.ts` in the `app` folder. And add the following code to it:

```
import { trigger, style, transition, animate, query, group, keyframes } from '@angular/animations
const duration = '1.5s';
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    // more code here later
  ]),
]);
```

1. Now let's add the query for when the card leaves the view. Inside the `transition` array, add the following `query` as follows:

```
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [
      style({ opacity: 0 }),
    ]),
    query( ':leave', [
        style({ transform: 'translateX(0)', opacity: 1 }),
        animate( `${duration} ease`, style({
            transform: 'translateX(100%)',
          })
        ),
        animate( `${duration} ease`, style({
            opacity: 0,
          })
        ),
      ],
      { optional: true }
    )
  ]),
]);
```

1. We will import the animation in the `app.component.ts` file and will add to the animations array as follows:

```
...
import { cardAnimation } from './animation';
...
@Component({
  ...
  animations: [cardAnimation],
  ...
})
export class AppComponent {...}
```

1. Now we'll update the `app.component.html` to use the animation on the element having the `card-container` class. Update the file as follows:

```
...
<main>
  …
  <div class="card-container relative h-[600px] w-full overflow-hidden py-4" [@cardAnimation]="s
    ...
  </div>
</main>
```

You should be able to see the animation now by clicking the Facebook and twitter buttons. However, it doesn't look pretty.

1. Let's add another query for when the next card enters the view. We'll first make sure that the card is `invisible` when starting to enter the DOM. Update the `animations.ts` file as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [
      style({ opacity: 0 }),
    ]),
    query( ':leave', [...]),
]);
```

1. Now add the second `query` for the card to be entered in the screen. We'll make sure it slides in from the left and becomes visible slowly. Update the `animations.ts` file as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [...]),
    query(':leave', [...]),
    query( ':enter', [
        style({
          transform: 'translateX(-100%)',
          opacity: 0,
        }),
        animate( `${duration} ease`, style({
            transform: 'translateX(0)',
            opacity: 1,
          })
        ),
      ],
      { optional: true }
    ),
  ]),
]);
```

You'll notice that the animations are working now. Although, they are pretty slow. That's because they are all running in a sequence.

1. We can wrap the 2nd and third query in a group() method to run them in parallel. Update the code in the `animations.ts` as follows:

```
import { ..., group } from '@angular/animations';
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [...]),
    group([
      query( ':leave', [...], { optional: true }),
      query(':enter', [...], { optional: true }),
    ]),
  ]),
]);
```

And boom! You can now see that the animations are running in parallel and are not waiting for the `:leave` transition to be finished before the `:enter` transition is executed.

How it works...

Animations in Angular run in sequence by default. If a transition has more than one step, i.e. `style()` and `animate()` usages, the animations will run in sequence. The `group()` method makes it possible for us to run animations in parallel. For this recipe, we wanted both `:enter` and `:leave` transitions to run at the same time and so we grouped them to run in parallel in the end.

See also

Sequential vs Parallel animations in Angular (https://angular.io/guide/complex-animation-sequences#sequential-vs-parallel-animation)Angular Animations `sequence` method (https://angular.io/api/animations/sequence)

Route animations in Angular

In this recipe, you'll learn how to implement route animations in Angular. You'll learn how to configure route animations by passing the transition state name to the route as a data property. You'll also learn how to use the `RouterOutlet` API to get the transition name and apply it to the animation to be executed. We'll implement some 3D transitions so it is going to be fun!

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-route-animations` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-route-animations` to serve the project

This should open the app in a new browser tab and you should see the following:

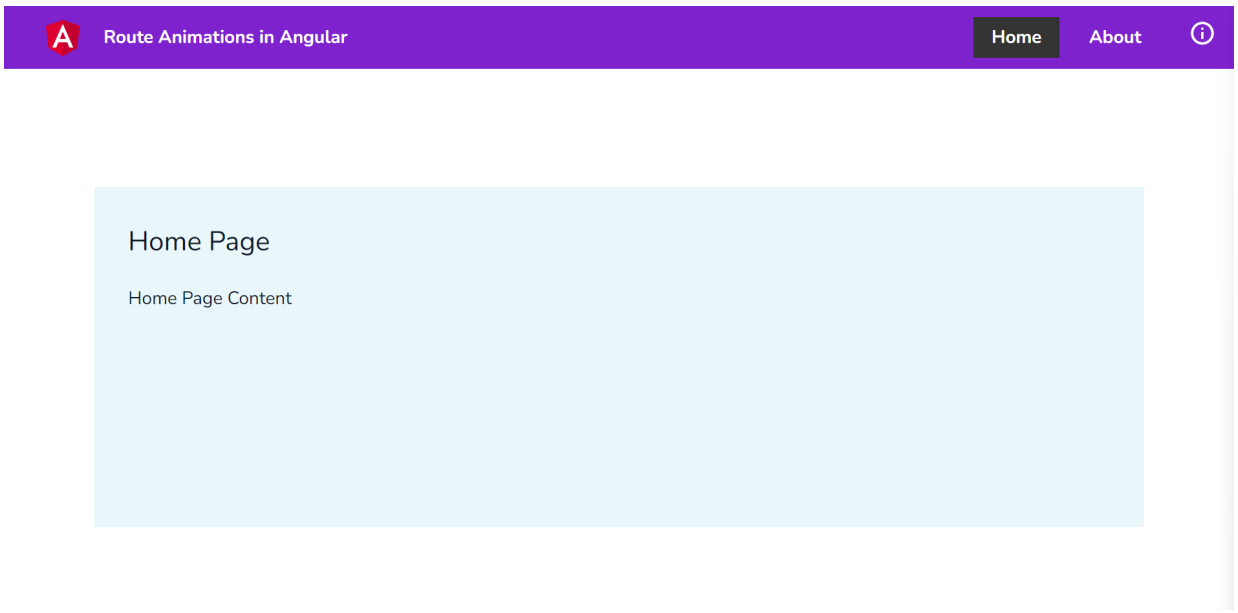


Figure 4.8 – ng-route-animations app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have a really simple app with two lazy-loaded routes at the moment. The routes are for the `Home` and the `About` pages, and we'll now start configuring the animations for the app:

1. First, let's provide Angular Animations using the `provideAnimations()` method from Angular in the `main.ts` file as follows:

```
import { bootstrapApplication } from '@angular/platform-browser';
...
import { provideAnimations } from '@angular/platform-browser/animations';
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation()),
    provideAnimations(),
  ],
}).catch((err) => console.error(err));
```

1. We'll now create a new file named `animations.ts` inside the app folder. Let's put the following code in the `animations.ts` file to register a basic trigger:

```
import {trigger,  transition } from '@angular/animations';
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    // states and transitions to be added here
  ])
]);
```

1. We'll now register some queries and basic states for the animations. Let's add the following items in the `transition()` method's array as follows:

```
import { trigger, style, transition, query
} from '@angular/animations';
const optional = { optional: true };
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative', perspective: '1000px' }),
    query(
      ':enter, :leave',
      [style({ position: 'absolute', width: '100%' })],
      optional
    ),
  ]),
]);
```

1. Alright! We have the `routeAnimation` trigger registered now for transition from every route to every other route. Now, let's provide those transition states in the routes.

We can provide the states for the transitions using a unique identifier for each route. There are many ways to do it, but the easiest way is to provide it using the `data` attribute in the route configuration as follows in `app.routes.ts` file:

```
export const appRoutes: Route[] = [
  ...,
  {
    path: 'home',
    data: { transitionState: 'HomePage' },
    loadComponent: () =>
      import('./home/home.component').then((m) => m.HomeComponent),
  },
  {
    path: 'about',
    data: { transitionState: 'AboutPage' },
    loadComponent: () =>
```

```
      import('./about/about.component').then((m) => m.AboutComponent),
  },
];
```

1. Now, we need to provide this `transitionState` property from the current route to the route animation trigger in the `app.component.html` file. For this, create a `@ViewChild` instance in the `app.component.ts` file. This View Child is going to target the `<router-outlet>` element in the `app.component.html` template. This is so we can get the current route's `data` and the `transitionState` value provided. The code in the `app.component.ts` file should look as follows:

```
import { CommonModule } from '@angular/common';
import { Component, ViewChild } from '@angular/core';
import { RouterModule, RouterOutlet } from '@angular/router';
export class AppComponent {
  @ViewChild(RouterOutlet) routerOutlet!: RouterOutlet;
}
```

1. We'll also import `ROUTE_ANIMATION` from the `animations.ts` file into `app.component.ts` as follows:

```
...
import { ROUTE_ANIMATION } from './animations';
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.scss"],
  animations: [
    ROUTE_ANIMATION
  ]
})
```

1. We'll now create a function named `getRouteAnimationTransition()`, which will get the current route's data and the `transitionState` value and return it back. This function will later be used in `app.component.html`. Modify your code in `app.component.ts` as follows:

```
...
@Component({
...
})
export class AppComponent {
  @ViewChild(RouterOutlet) routerOutlet!: RouterOutlet;
  getRouteAnimationState() {
    return (
      this.routerOutlet &&
      this.routerOutlet.activatedRouteData &&
      this.routerOutlet.activatedRouteData['transitionState']
    );
  }
}
```

1. Finally, let's use the `getRouteAnimationState()` method with the `@routeAnimation` trigger in `app.component.html` so we can see the animation in play:

```
...
<div class="content" role="main">
  <div class="router-container"   [@routeAnimation]="getRouteAnimationState()">
    <router-outlet></router-outlet>
  </div>
</div>
```

1. Now that we have everything set up, let's finalize the animations. We'll add a query for when a route leaves the view. Update the `animations.ts` file as follows:

```
import { trigger, style, transition, query, animate, keyframes } from '@angular/animations';
...
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative', perspective: '1000px' }),
```

```
    query( ':enter, :leave', [...] ,optional),
    query( ':leave', [
        animate( '1s ease-in', keyframes([
            style({ opacity: 1, offset: 0,
              transform: 'rotateY(0) translateX(0) translateZ(0)',
            }),
            style({ offset: 0.25, transform: 'rotateY(45deg) translateX(25%) translateZ(100px) t
            }),
            style({ offset: 0.5, transform: 'rotateY(90deg) translateX(75%) translateZ(400px) tra
            }),
            style({ offset: 0.75, transform: 'rotateY(135deg) translateX(75%) translateZ(800px) t
            }),
            style({ opacity: 0, offset: 1, transform: 'rotateY(180deg) translateX(0) translateZ(1
            }),
          ])
        ),
      ],
      optional
    ),
  ]),
]);
```

If you navigate between routes, you'll notice the leaving route going out with animation behind the entering route. Let's add the animation for the entering route as well.

1. We'll add the animation for the route entering the view. Update the `animations.ts` as follows:

```
...
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative', perspective: '1000px' }),
    query(':enter, :leave', ...),
    query(':leave', ...),
    query( ':enter', [
        animate( '1s ease-out', keyframes([
            style({ opacity: 0, offset: 0, transform: 'rotateY(180deg) translateX(25%) translateZ
            }),
            style({ offset: 0.25, transform:  'rotateY(225deg) translateX(-25%) translateZ(1200px
            }),
            style({ offset: 0.5, transform: 'rotateY(270deg) translateX(-50%) translateZ(400px)',
            }),
            style({ offset: 0.75, transform: 'rotateY(315deg) translateX(-50%) translateZ(25px)',
            }),
            style({ opacity: 1, offset: 1, transform: 'rotateY(360deg) translateX(0) translateZ((
            }),
          ])
        ),
      ],
      optional
    ),
  ]),
]);
```

If you look at the animation route while navigating, you'll notice that the entering route appears immediately, then we see the leaving route's animation, and after that we see the entering route's animation. Let's group the entering and leaving animations together to run them in parallel.

1. Update the `animations.ts` as follows to run the entering and leaving routes' animations in parallel:

```
import {..., group } from '@angular/animations';
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative', perspective: '1000px' }),
    query( ':enter, :leave', ...),
    group([
      query( ':leave', [...], optional ),
      query( ':enter', [...], optional),
    ]),
```

```
  ]),
]);
```

Voila! Refresh the app and see the magic in place. You should now see the 3D animation for both the entering and leaving routes as you navigate from the `Home` page to the `About` page and vice versa. The sky's the limit when it comes to what you can do with keyframes and animations in Angular.

How it works...

1. In the `animations.ts` file, we first define our animation trigger named `routeAnimation`. Then we make sure that by default, the HTML element to which the trigger is assigned has `position: 'relative'` set as a style:

```
transition('* <=> *', [
    style({
      position: 'relative'
    }),
    ...
])
```

1. Then we apply the styled `position: 'absolute'` to the children, as mentioned, using `:enter` and `:leave` as follows:

```
query(':enter, :leave', [
  style({
    position: 'absolute',
    width: '100%'
  })
], {optional: true}),
```

1. This makes sure that these elements, that is, the routes to be loaded, have the `position: 'absolute'` style and a full width using `width: '100%'` so they can appear on top of each other. You can always fiddle around by commenting either of the styles to see what happens (at your own risk, though!).

Then we defined our route transitions as a combination of two animations, the first for `query :leave` and the second for `query :enter`. For the route leaving the view, we set the opacity to `0` via animation, and for the route entering the view, we set the opacity to `1` via animation as well. Note that animations via Angular animations run in sequence:

```
query(':leave', [
  ...
], {optional: true}),
query(':enter', [
  ...
], {optional: true}),
```

You'll notice that in our code we're using `keyframes` for animations. For the leaving route, it starts with `opacity 1`, and without any transformations initially. And then it ends with `opacity 0` but with the transform set to `'rotateY(180deg) translateX(0) translateZ(1200px) translateY(25%)'`. And it is the opposite for the entering route.Finally, we use the `group()` method to wrap both the leaving and entering animations together so they can run in parallel instead of in sequence. This makes it so that the entering route comes in as the leaving route goes away.

See also

Animations in Angular (https://angular.io/guide/animations)Angular route transition animations (https://angular.io/guide/route-animations)

How it works...

Since we wanted to implement a 3D animation in this recipe, we first made sure that the animation host element had a value for the `perspective` style, so we can see all the magic in 3D. Then we defined our animations using the `keyframes` method with an animation state for each offset so we could set different angles and rotations at those states, just so it all looks cool. One important thing that we did was group our `:enter` and `:leave` queries using the `group` method, where we defined the animations. This made sure that we had the route entering and leaving the view simultaneously.

See also

Animations in Angular (https://angular.io/guide/animations)Angular route transition animations (https://angular.io/guide/route-animations)Fireship.io's tutorial on Angular route animations (https://fireship.io/lessons/angular-router-animations/)Angular complex animation sequences (https://angular.io/guide/complex-animation-sequences)

Disabling Angular animations conditionally

In this recipe, you'll learn how to disable animations in Angular conditionally. This is useful for a variety of cases including disabling animations on a particular device for example. Protip: Use `ngx-device-detector` to identify if your Angular app is running on a mobile or tablet etc. I built it :) Shameless plug aside, in this recipe we'll disable the animations for employees of the app considering we're rolling out animations only for admins at the moment.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-disable-animations` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-disable-animations` to serve the project

This should open the app in a new browser tab. Log in as an admin, add a few bucket items, and you should see the following:

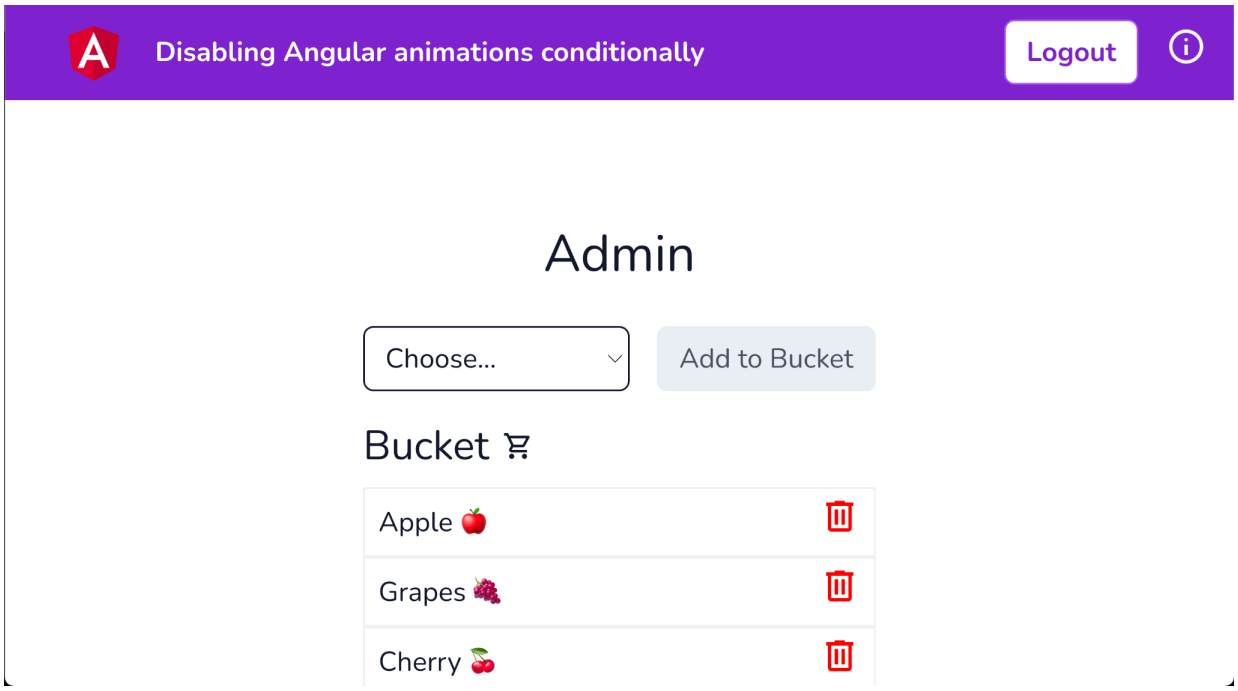


Figure 4.1 – ng-disable-animations app running on http://localhost:4200

Now that we have the app running, we will move on to the steps for the recipe.

How to do it...

We have an app that already has some Angular animations configured. You will notice that both the admin and the employee page has the animations enabled. We will disable the animations for the employee page using a config. Let's continue with the steps as follows:

1. First, we'll add a new property named `disableAnimations` to our `IAppConfig` interface and the `APP_CONFIG` variable in `the app-config.ts` file as follows:

```
import { InjectionToken } from '@angular/core';
export interface IAppConfig {
  canDeleteItems: boolean;
  disableAnimations: boolean;
}
export const APP_CONFIG = new InjectionToken<IAppConfig>('APP_CONFIG');
export const AppConfig: IAppConfig = {
  canDeleteItems: true,
  disableAnimations: false,
};
```

1. Typescript will start throwing errors as we need to add the same `disableAnimations` property in the `employee.config.ts` file as well.

Update the `employee.config.ts` file as follows:

```
import { IAppConfig } from '../app-config';
export const EmployeeConfig: IAppConfig = {
  canDeleteItems: false,
  disableAnimations: true
};
```

1. Finally, add a `HostBinding` in the `bucket.component.ts` to disable the animations in the bucket component based on the config. Update the `bucket.component.ts` file as follows:

```
import { CommonModule } from '@angular/common';
import { Component, HostBinding, inject, OnInit } from '@angular/core';
...
@Component({...})
export class BucketComponent implements OnInit {
  ...
  fruits: string[] = Object.values(Fruit);
  @HostBinding('@.disabled')
  public animationsDisabled = this.appConfig.disableAnimations;
  ngOnInit(): void {...}
        ...
}
```

Great! If you now refresh the app and see the Admin page, you'll see the animations working. And if you go to the Employee page. You'll see that the animations are disabled there. Magic! See the next section to understand how the recipe works.

How it works...

Angular provides a way to disable the animations using the `[@.disabled]` binding. You can put it anywhere in the template with an expression that evaluates a truthy value. And in that case all the **child animations** that are applied in its nested HTML tree will be disabled. We had an application level config that is overridden in the employee component via the `EmployeeConfig` object. So first we create a `disableAnimations` property in the `IAppConfig` interface. This interface is used both by the `AppConfig` variable in the `app-config.ts` file and by the `EmployeeConfig` variable in the `employee.config.ts` file. As you can see we set the value of `disabledAnimations` to `false` for `AppConfig` and `true` for `EmployeeConfig`. And then we use the `@HostBinding()` decorator in the `BucketComponent` class by assigning its value to the provided config's `disabledAnimations` property. Since the `AdminComponent` class gets the `AppConfig` and the `EmployeeComponent` gets the `EmployeeConfig`, the animations are enabled and disabled respectively for these components.

See also

- Animations in Angular (https://angular.io/guide/animations)
- *Angular Animations Explained with Examples* (https://www.freecodecamp.org/news/angular-animations-explained-with-examples/)

5 Angular and RxJS – Awesomeness Combined

Angular and RxJS create a killer combination of awesomeness. By combining these, you can handle your data reactively, work with streams, and do really complex stuff in your Angular apps. That's exactly what you're going to learn in this chapter.Here are the recipes we're going to cover in this chapter:

- Sequential and Parallel HTTP calls in Angular with RxJS
- Listening to multiple observable streams
- Unsubscribing streams to avoid memory leaks
- Using Angular's `async` pipe to unsubscribe streams automagically
- Using the `map` operator to transform data
- Using the `switchMap` and `debounceTime` operators with autocompletes for better performance
- RxJS custom operator
- Retry failed HTTP calls with RxJS

Technical requirements

For the recipes in this chapter, make sure you have **Git** and **Node.js** installed on your machine. You also need to have the `@angular/cli` package installed, which you can do with `npm install -g @angular/cli` from your Terminal. The code for this chapter can be found at the following link: https://github.com/PacktPublishing/Angular-Cookbook/tree/master/chapter05.

Sequential and Parallel HTTP calls in Angular with RxJS

In this recipe, you'll learn how to use different RxJS operators to make sequential and parallel http calls in Angular apps. We'll work with the famous Star Wars API (Swapi) to get some data to display it on the UI

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-seq-parallel-http` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-seq-parallel-http` to serve the project

This should open the app in a new browser tab and you should see the following:

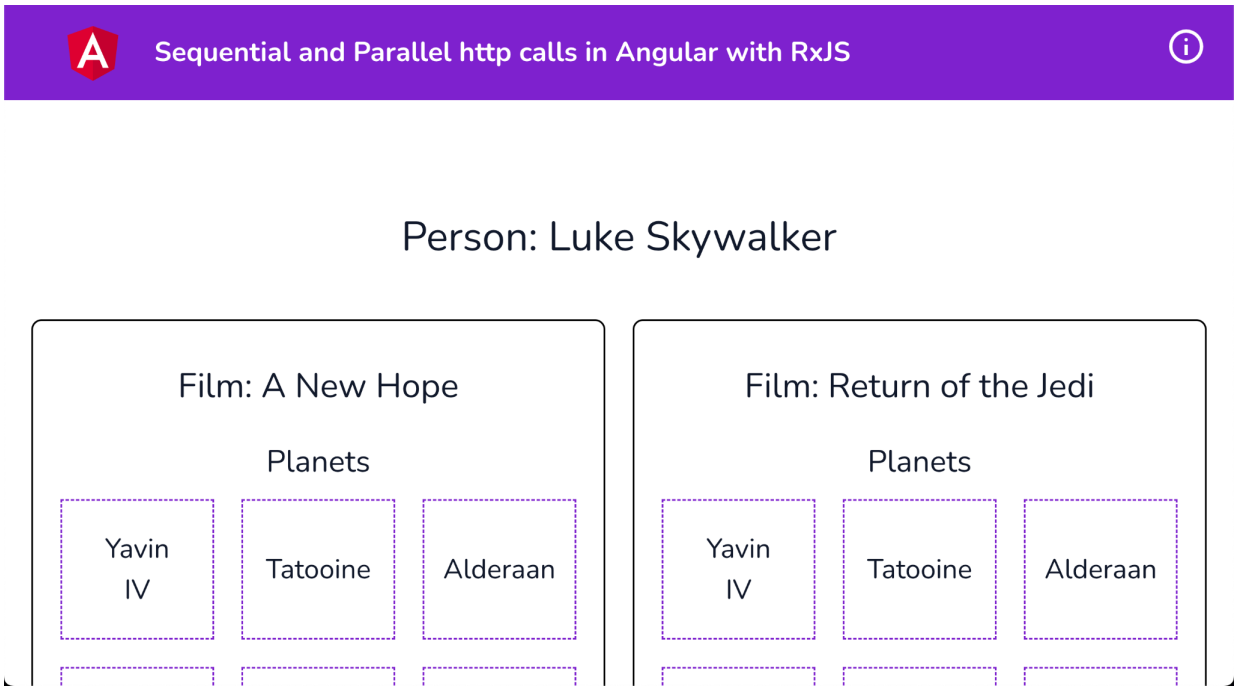


Figure 5.1 – The rx-seq-parallel-http app running on http://localhost:4200

Now that we have the app running, we will move on to the steps for the recipe.

How to do it...

We have an Angular app that uses the Star Wars API (Swapi) to fetch a person from Star Wars, and the films the person has been in. All of it happens using a lot of HTTP calls and our code is totally garbage so far. This is because we show the loader but we hide it before we've retrieved all the data. Also, if you keep refreshing the page, you'll see that the sequence of the films can change each time. Hence, we see the UI jumping a lot. We will improve our code using RxJS. Let's begin.

1. First, we'll avoid using the `setTimeout()` method to rely on the person data to be available in `1500ms`. We'll rather move this inside the subscribe block. Update the `fetchData()` method inside the `app.component.ts` as follows:

```
fetchData() {
    this.loadingData = true;
    this.swapi.fetchPerson('1').subscribe((person) => {
      this.person = person;
      this.person.filmObjects = [];
      this.loadingData = false; <-- remove this
      this.person.films.forEach((filmUrl) => {
        this.swapi.fetchFilm(filmUrl).subscribe((film) => {
          this.person.filmObjects.push(film);
          this.loadingData = false;
        });
      });
    });
  }
```

1. Now, instead of using the API calls of films inside the subscribe block of the person's API call, we're going to use the `mergeMap()` operator from RxJS to chain these calls. Update the `fetchData()` method as follows now:

```
...
import { mergeMap, of } from 'rxjs';
```

```
...
  fetchData() {
    this.loadingData = true;
    this.swapi
      .fetchPerson('1')
      .pipe(
        mergeMap((person) => {
          this.person = person;
          this.person.filmObjects = [];
          return of(this.person.films);
        })
      )
      .subscribe((films) => {
        films.forEach((filmUrl) => {
          this.swapi.fetchFilm(filmUrl).subscribe((film) => {
            this.person.filmObjects.push(film);
            this.loadingData = false;
          });
        });
      });
  }
```

Notice that the UI becomes *a bit better*. The loader now hides as soon as one of the films is retrieved from the server. However, we want to hide the loader when all of them have been retrieved. Also the sequence of the films is still unpredictable.

1. Now, we'll use the `forkJoin()` to make the API calls for the films in parallel and to wait for the combined response. We're doing it instead of the `of()` operator which just passes the film urls from the `mergeMap()`. Update the `fetchData()` method as follows and update the imports at the top:

```
import { forkJoin, mergeMap, of (<-- remove of) } from 'rxjs';
...
  fetchData() {
    this.loadingData = true;
    this.swapi
      .fetchPerson('1')
      .pipe(
        mergeMap((person) => {
          this.person = person;
          this.person.filmObjects = [];
          return forkJoin([
            ...this.person.films.map((filmUrl) =>
              this.swapi.fetchFilm(filmUrl)
            ),
          ]);
        })
      )
      .subscribe((filmObjects) => {
        this.person.filmObjects = filmObjects;
        this.loadingData = false;
      });
  }
```

Woohoo! Now if you refresh the app, you'll notice two things. First, the loader always stops only when all the data has been fetched. And second, the sequence of the films is always the same (and correct). Now that you've finished the recipe let's move on to the next section to understand how it all works.

How it works...

The `mergeMap()` operator allows us to chain observables by returning an `Observable` from its callback. You can think of it like how we chain `Promise.then()`, but this is for observables. We first removed the `setTimeout()` (which is mostly stupid to put into the code for these cases) and moved the logic of fetching films inside the subscribe block of fetching the person. We also used the `of` operator to return the `person.films` array from the `mergeMap()` method's callback. The `mergeMap()` method is used to chain observables together and in our context, it can be chained to wait for one HTTP call to finish so we can execute the other ones. In step 3, we intend to execute multiple HTTP calls together in parallel for all

the films for the person. We do this using the `forkJoin()` operator which takes an array of observables. In this case, those observables are HTTP calls for each film. `forkJoin()` also makes it possible to wait for all the parallel calls to finish and then trigger the subscribe block's callback. One more thing `forkJoin()` does is it provides us the responses in the form of an array having the responses in the same sequences as the observables. This makes the responses predictable and we always show the same data on the UI.

See also

Catch the Dot Game—RxJS documentation (https://www.learnrxjs.io/learn-rxjs/recipes/catch-the-dot-game)RxJS `mergeMap` operator documentation (https://www.learnrxjs.io/learn-rxjs/operators/transformation/mergemap)RxJS `merge` operator documentation (https://www.learnrxjs.io/learn-rxjs/operators/combination/forkjoin)

Listening to multiple observable streams

In this recipe, we'll work with `combineLatest()` operator to listen to multiple observable streams at once. Using this operator results in having an array as an output, combining all the streams. This approach is appropriate for when you want the latest output from all the streams, combined in a single subscribe.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-multiple-streams` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-multiple-streams` to serve the project

This should open the app in a new browser tab and you should see the following:

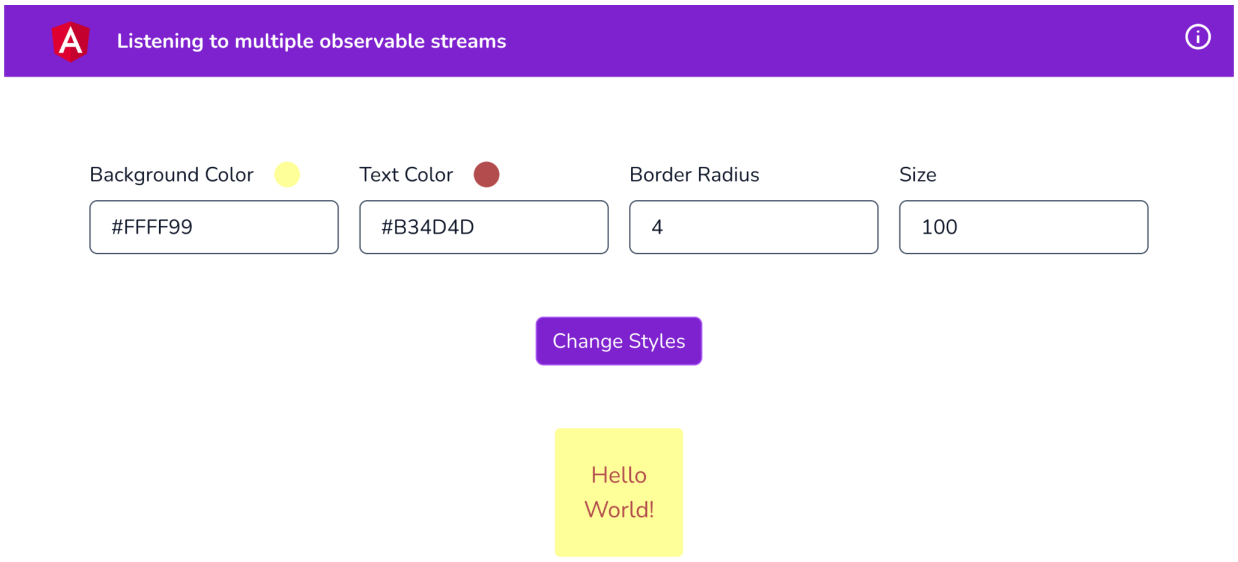


Figure 5.2 – The rx-multiple-streams app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

For this recipe, we have an app that displays a box. The box has a size (width and height), a border radius, a background color, and a color for its text. It also has four inputs using the **Reactive Forms** to modify all the mentioned factors. Right now, we have to apply the changes manually with the click of a button. What if we could subscribe to the changes to the inputs and update the box right away? That's what we're going to do here.

1. We'll begin by creating a method named `listenToInputChanges()`. We'll create an array of controls we want to work with. Update the code of `home.component.ts` as follows:

```
...
export class HomeComponent implements OnInit {
  ...
  ngOnInit() {
    this.applyChanges();
  }
  listenToInputChanges() {
    const controls: AbstractControl[] = [
      this.boxForm.controls.size,
      this.boxForm.controls.borderRadius,
      this.boxForm.controls.textColor,
      this.boxForm.controls.backgroundColor,
    ];
  }
  ...
}
```

1. Now we'll loop over the controls to give them the initial value so when the observable stream is subscribed, they have a value to work with. Update the `listenToInputChanges()` method further as follows:

```
import { startWith } from 'rxjs';
...
export class HomeComponent implements OnInit {
...
listenToInputChanges() {
    const controls: AbstractControl[] = [...];
    controls.map((control) =>
      control.valueChanges.pipe(startWith(control.value))
    );
  }
}
```

1. Now we'll replace the `boxStyles` property with an `Observable` named `boxStyles$`. Then we'll wrap the `valueChanges` streams inside a `combineLatest()` operator to join them. Finally, we'll assign the result of this joined stream to the `boxStyles$` observable. Update the `home.component.ts` file as follows:

```
...
import { Observable, combineLatest, startWith} from 'rxjs';
...
export class HomeComponent implements OnInit, OnDestroy {
  ...
  boxStyles$!: Observable<BoxStyles>;
        ...
  listenToInputChanges() {
    this.boxStyles$ = combineLatest(
      controls.map((control) =>
        control.valueChanges.pipe(startWith(control.value))
      );
    );
  }
  ...
}
```

1. Now we will use a `map()` operator with `.pipe()` on the combined stream to map it to the `BoxStyle` type values. Update the `listenToInputChanges()` method in the `home/home.component.ts` file, as follows:

```
import { combineLatest, map, Observable, startWith } from 'rxjs';
export class HomeComponent implements OnInit {
  listenToInputChanges() {
    const controls: AbstractControl[] = [...];
    this.boxStyles$ = combineLatest(...).pipe(
      map(([size, borderRadius, textColor, backgroundColor]) => {
        return { width: `${size}px`, height: `${size}px`,
          backgroundColor: backgroundColor,
          color: textColor,
          borderRadius: `${borderRadius}px`,
        };
      })
    );
  }
}
```

1. We need to remove the `setBoxStyles()` and `applyChanges()` methods and the usages of the `applyChanges()` method from the `home.component.ts` file. Update the file, as follows:

```
export class HomeComponent implements OnInit {
  ...
  ngOnInit() {
    this.listenToInputChanges(); ← Add this call
    ...
    this.applyChanges(); ← Remove this call
  }
  ...
  setBoxStyles(...) {...}  ← Remove this method
  applyChanges() {...} ← Remove this method
  ...
}
```

1. We also need to remove the usage of the `applyChanges()` method from the template as well. Remove the `(ngSubmit)` handler from the `<form>` element in the `home.component.html` file so that it looks like this:

```
<div class="home" [formGroup]="boxForm" (ngSubmit)="applyChanges()" ← Remove this>
  ...
</div>
```

1. We also need to get rid of the `submit-btn-container` element from the `home.component.html` template as we don't need it anymore. Delete the following chunk from the file:

```
<div class="row submit-btn-container" ← Remove this element>
  <button class="btn btn-primary" type="submit"   (click)="applyChanges()">Change Styles</button>
</div>
```

1. Now that we can work with the `boxStyles$` Observable. Let's use it in the template, i.e. the `home.component.html` file instead of the `boxStyles` property:

```
  ...
  <div class="row" *ngIf="boxStyles$ | async as boxStyles">
    <div class="box" [ngStyle]="boxStyles">
      <div class="box__text">
        Hello World!
      </div>
    </div>
  </div>
  ...
```

And voilà! If you refresh the app, you should be able to see the box appearing with the default styles. And if you change any of the options, you'll see the changes reflecting as well. Congratulations on finishing

the recipe. You're now the master of handling multiple streams using the `combineLatest()` operator. See the next section to understand how it works.

How it works...

The beauty of **Reactive Forms** is that they provide much more flexibility than the regular `ngModel` binding or even template-driven forms. And for each form control, we can subscribe to its `valueChanges` observable, which receives a new value whenever the input is changed. So, instead of relying on the `Submit` button's click, we subscribed directly to the `valueChanges` property of each `form control`. In a regular scenario, that would result in four different streams for four inputs, which means we would have four subscriptions that we need to take care of and make sure we unsubscribe them. This is where the `combineLatest` operator comes into play. We used the `combineLatest` operator to combine those four streams into one, which means we needed to unsubscribe only one stream on component destruction. But hey! Remember that we don't need to do this if we use the `async` pipe? That's exactly what we did. We removed the subscription from the `home.component.ts` file and used the `.pipe()` method with the `.map()` operator. The `.map()` operator transformed the data to our needs, and then returned the transformed data to be set to the `boxStyles$` observable. Finally, we used the `async` pipe in our template to subscribe to the `boxStyles$` observable and assigned its value as the `[ngStyle]` to our box element. Since the `valueChanges` is a `Subject` instead of a `ReplaySubject`, we also piped a `startWith()` with the valueChanges to provide an initial value. If we don't use `startWith()`, the box won't show unless all the inputs have a value changed manually at least once. Try it!

See also

`combineLatest` operator documentation (https://www.learnrxjs.io/learn-rxjs/operators/combination/combinelatest)Visual representation of the `combineLatest` operator (https://rxjs-dev.firebaseapp.com/api/index/function/combineLatest)

Unsubscribing streams to avoid memory leaks

Streams are fun to work with and they're awesome And you'll know much more about RxJS and streams when you've finished this chapter. One reality is facing unseen problems that occur when streams are used without caution. One of the biggest mistakes to do with streams is to not unsubscribe them when we no longer need them, and in this recipe, you'll learn how to unsubscribe streams to avoid memory leaks in your Angular apps.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-unsubscribing-streams` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-unsubscribing-streams` to serve the project

This should open the app in a new browser tab and you should see the following:

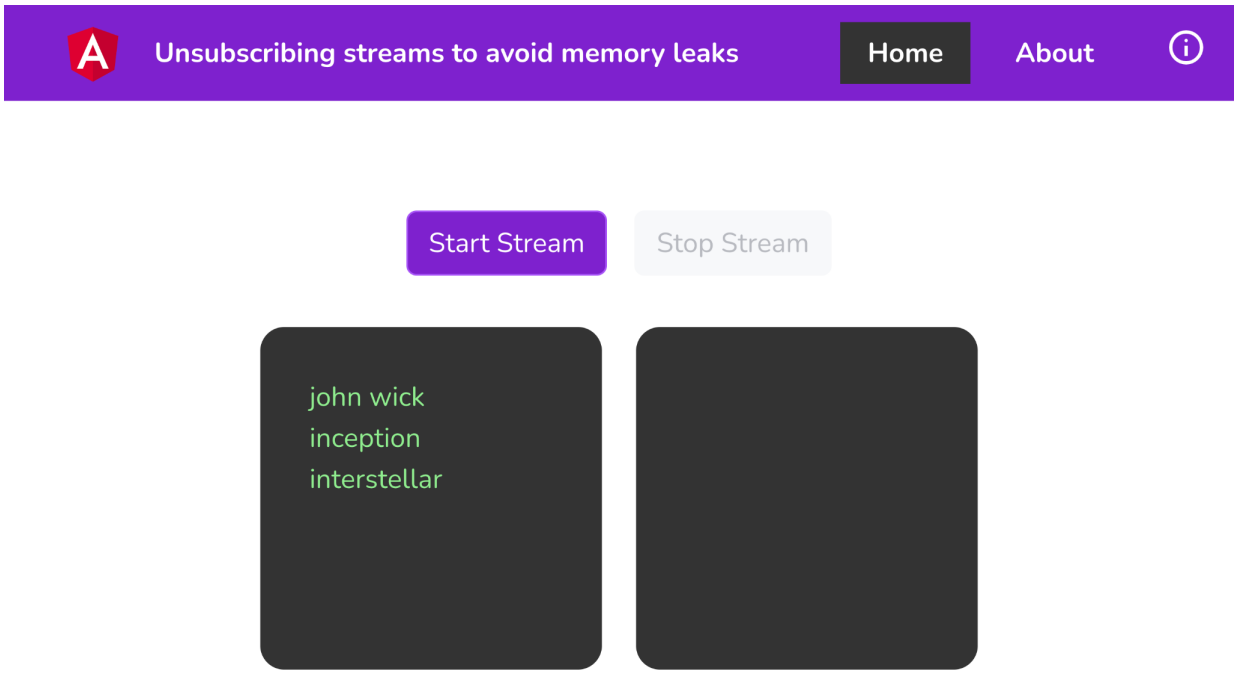


Figure 5.3 – The rxjs-unsubscribing-streams app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We currently have an app with two routes—that is, `Home` and `About`. This is to show you that unhandled subscriptions can cause memory leaks in an app. The default route is `Home`, and in the `HomeComponent` class, we handle a single stream that outputs data using the `interval` method.

1. Tap the **Start Stream** button, and you should see the stream emitting values.
2. Then, navigate to the **About** page by tapping the **About** button from the header (top right), and then come back to the **Home** page.

Do you see anything weird? No? Everything looks fine, right? Well, not exactly.

1. To see whether we have an unhandled subscription, let's put a `console.log` inside the `startStream` method in the `home.component.ts` file—specifically, inside the `.subscribe` method's block, as follows:

```
...
export class HomeComponent implements OnInit {
  ...
  startStream() {
    const streamSource = interval(1500);
    this.subscription = streamSource.subscribe((input) => {
      this.outputStreamData.push(input);
      console.log({ input });
    });
  }
  stopStream() {...}
}
```

1. If you now perform the same steps as mentioned in *Step 1*, you'll see the following output on the console:

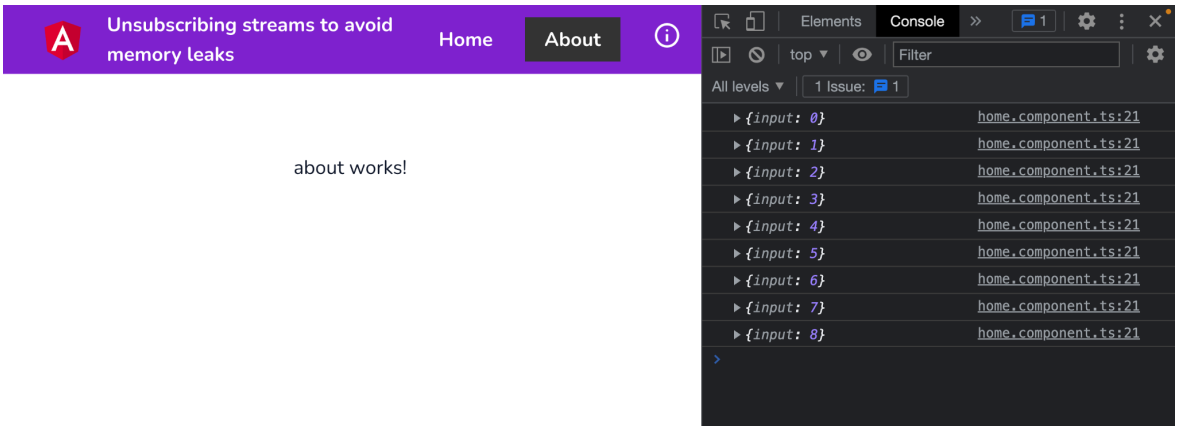


Figure 5.9 – interval emitting values on about page

2. Want to have some more fun? Try performing *Step 1* a couple of times without refreshing the page even once. What you'll see will be `CHAOS` !

So, to solve the issue, we'll use the simplest approach—that is, unsubscribing the stream when the user navigates away from the route. Let's implement the `ngOnDestroy` lifecycle method for that, as follows:

```
import { Component, OnDestroy } from '@angular/core';
...
@Component({...})
export class HomeComponent implements OnDestroy {
  ...
  startStream() {...}
  ngOnDestroy() {
    this.stopStream();
  }
  stopStream() {...}
}
```

Great! If you follow the instructions from *Step 1* again, you'll see that there's no further log on the console once you navigate away from the **Home** page, and our app doesn't have an unhandled stream causing memory leaks now. Read the next section to understand how it works.

How it works...

When we create an Observable/stream and we subscribe to it, RxJS automagically adds our provided `.subscribe` method block as a handler to the `Observable` . So, whenever there's a value emitted from the `Observable` , our method is supposed to be called. The fun part is that Angular doesn't automatically destroy that subscription/handler when the component unmounts or when you have navigated away from the route. That's because the core of Observables is `RxJS` , not Angular, and therefore it isn't Angular's responsibility to handle it.Angular provides certain lifecycle methods, and we used the `OnDestroy` (`ngOnDestroy`) method. So we used `ngOnDestroy()` to call the `stopStream()` method so that the subscription is destroyed as soon as the user navigates away from the page. This becomes possible because when we navigate away from a route, Angular destroys that route and thus we can execute our `stopStream()` method.

There's more...

1. In a complex Angular app, there will be cases where you'd have more than one subscription in a component, and when the component is destroyed, you'd want to clean all those subscriptions at once. Similarly, you might want to unsubscribe based on certain events/conditions rather than the `OnDestroy` lifecycle. Here is an example, where you have multiple subscriptions in hand and you want to clean up all of them together when the component destroys:

```
startStream() {
    const streamSource = interval(1500);
    const secondStreamSource = interval(3000);
    const fastestStreamSource = interval(500);
    streamSource.subscribe((input) => {
      this.outputStreamData.push(input);
      console.log('first stream output', input);
    });
    secondStreamSource.subscribe(input => {
      this.outputStreamData.push(input);
      console.log('second stream output', input)
    });
    fastestStreamSource.subscribe(input => {
      this.outputStreamData.push(input);
      console.log('fastest stream output', input)
    });
  }
  stopStream() {
    // remove code from here
  }
```

1. Notice that we're not saving the `Subscription` from `streamSource` to `this.subscription` anymore, and we have also removed the code from the `stopStream` method. The reason for this is because we don't have individual properties/variables for each Subscription. Instead, we'll have a single variable to work with. Let's look at the following recipe steps to get things rolling.
2. First, we'll create a property in the `HomeComponent` class named `isStreamActive`:

```
import { Component, OnDestroy } from '@angular/core';
    ...
export class HomeComponent implements OnDestroy {
  isStreamActive!: boolean;
  ...
}
```

1. Now, we'll import the `takeWhile` operator from `rxjs/operators`, as follows:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
...
import { interval, Subscription, takeWhile } from 'rxjs';
```

1. We'll now use the `takeWhile` operator with each of our streams to make them work only when the `isStreamActive` property is set to `true`. Since `takeWhile` takes a `predicate` method, it should look like this:

```
startStream() {
    ...
    streamSource
      .pipe(takeWhile(() => !!this.isStreamActive))
      .subscribe(input => {...});
    secondStreamSource
      .pipe(takeWhile(() => !!this.isStreamActive))
      .subscribe(input => {...});
    fastestStreamSource
      .pipe(takeWhile(() => !!this.isStreamActive))
      .subscribe(input => {...});
  }
```

If you press the **Start Stream** button right now on the `Home` page, you still won't see any output or logs because the `isStreamActive` property is still `undefined`.

1. To make the streams work, we'll set the `isStreamActive` property to `true` in the `startStream` method. The code should look like this:

```
  ngOnDestroy() {
    this.stopStream();
  }
  startStream() {
```

```
  isStreamActive = true;
  const streamSource = interval(1500);
  const secondStreamSource = interval(3000);
  const fastestStreamSource = interval(500);
  ...
}
```

After this step, if you now try to start the stream and navigate away from the page, you'll still see the same issue with the streams—that is, they've not been unsubscribed.

1. To unsubscribe all streams at once, we'll set the value of `isComponentAlive` to `false` in the `stopStream` method, as follows:

```
stopStream() {
  this.isStreamActive = false;
}
```

Finally, update the template to handle which button is disabled based on the `isStreamActive` property instead of `subscription`. Update the `home.component.html` file as follows:

```
<div class="home">
    <div class="buttons-container">
      <button [disabled]="!!isStreamActive" class="btn btn-primary" (click)="startStream()">Sta
      <button [disabled]="!isStreamActive" class="btn btn-dark" (click)="stopStream()">Stop Str
    </div>
    ...
</div>
```

And boom! Now, if you navigate away from the route while the streams are emitting values, the streams will stop immediately as soon as you navigate away from the `Home` route. Voilà!

See also

Read about RxJS Subscription (https://www.learnrxjs.io/learn-rxjs/concepts/rxjs-primer#subscription) `takeWhile` docs (https://www.learnrxjs.io/learn-rxjs/operators/filtering/takewhile)

Using Angular's async pipe to unsubscribe streams automagically

As you learned in the previous recipe, it is crucial to unsubscribe the streams you subscribe to. What if we had an even simpler way to unsubscribe them when the component gets destroyed—that is, letting Angular take care of it somehow? In this recipe, you'll learn how to use Angular's `async` pipe with an `Observable` to directly bind the data in the stream to the Angular template instead of having to subscribe in the `*.component.ts` file.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/ng-async-pipe` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-async-pipe` to serve the project

This should open the app in a new browser tab and you should see the following:

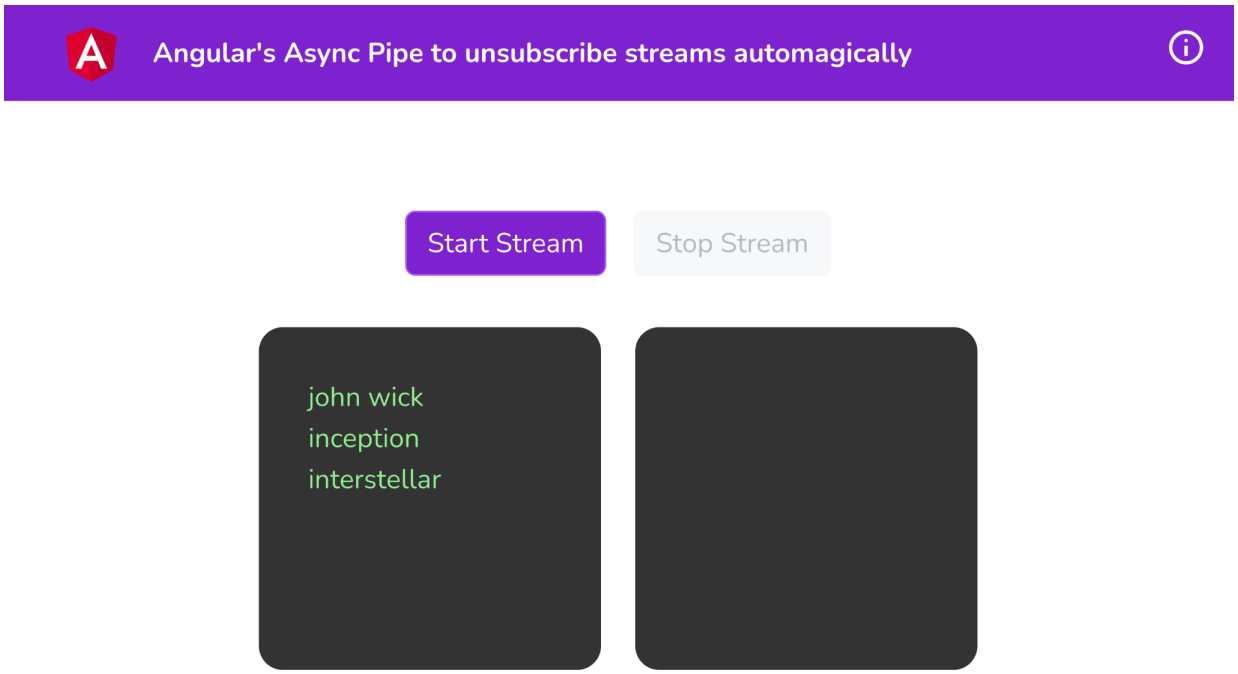


Figure 5.4 – The ng-async-pipe app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

The app we have right now has three streams/observables observing values at different intervals. We're relying on the `isStreamActive` property to keep the subscription alive or make it stop when the property is set to `false`. We'll remove the usage of `takeWhile` and somehow make everything work similarly to what we have right now.

1. First, add an `Observable` type property in the `HomeComponent` class named `streamOutput$`. Update the code in the `home.component.ts` file as follows:

```
...
import { interval, Observable, takeWhile } from 'rxjs';
...
export class HomeComponent implements OnDestroy {
  ...
  isStateActive!: boolean;
  streamsOutput$!: Observable<number>;
  constructor() { }
  ...
}
```

1. We'll now combine all the streams to give out a single output—that is, the `outputStreamData` array. We'll remove all the existing `.pipe` and `.subscribe` methods from the `startStream()` method, so the code should now look like this:

```
...
import { interval, merge, scan, Observable, takeWhile } from 'rxjs';
...
export class HomeComponent implements OnInit, OnDestroy {
  ...
  startStream() {
    ...
```

```
    const fastestStreamSource = interval(500);
    this.streamsOutput$ = merge(
      streamSource,
      secondStreamSource,
      fastestStreamSource
    ).pipe(
      scan((acc, next) => {
        return [...acc, next];
      }, [] as number[])
    );
  }
  ...
}
```

1. Since we want to stop the stream on the **Stop Stream** button click, we'll use the `takeWhile` operator in th stream to work with the stream to only emit values when we click the **Start Stream** button, and to stop when we hit the **Stop Stream** button. Update the `startStream()` function in `home.component.ts` as folows:

```
startStream() {
    ...
    this.streamsOutput$ = merge(...).pipe(
      takeWhile(() => !!this.isStreamActive),
      scan((acc, next) => {
        return [...acc, next];
      }, [] as number[])
    )
  }
```

Remove the `ngOnDestroy` method since our stream is going to unsubscribe automatically when we leave the component. Also remove the `implements OnDestroy` statement and remove the `OnDestroy` import.

1. Finally, modify the template in `home.component.html` to use the `streamOutput$` Observable with the `async` pipe to loop over the output array:

```
<div class="output-stream">
  <div class="input-stream__item" *ngFor="let item       of streamsOutput$ | async">
    {{item}}
  </div>
</div>
```

1. To verify that the subscription REALLY gets destroyed on component destruction, let's put a `console.log` in the `startStream` method inside the `map` operator, as follows:

```
import { ..., takeWhile, tap } from 'rxjs';
startStream() {
    ...
    this.streamsOutput$ = merge(...).pipe(
      takeWhile(...),
      scan(...),
      tap((output) => console.log('output', output))
    )
  }
```

Hurray! With this change, you can try refreshing the app, navigate away from the `Home` route, and you'll see that the console logs stop as soon as you navigate away from the Home page. Also, you can start and stop the stream to see the output on console too. Do you feel great about what we just got by removing all that extra code? I certainly do. Well, see in the next section how it all works.

How it works...

Angular's `async` pipe automatically destroys/unsubscribes the subscription as soon as the component destroys. This gives us a great opportunity to use it where possible. In the recipe, we basically combined all the streams using the `merge` operator. The fun part was that for the `streamsOutput$` property, we wanted an Observable of the output array on which we could loop over. However, merging the stream

only combines them and emits the latest value emitted by any of the streams. So, we added a `.pipe()` method with the `.scan()` operator to take the latest output out of the combined stream and to add it to an array of all the previous emitted outputs. This sort of works like the `reduce` method of a javascript array. Fun fact—streams don't emit any value unless they're subscribed to. "*But Ahsan, we didn't subscribe to the stream, we just merged and mapped the data. Where's the subscription?*" Glad you asked. Angular's `async` pipe subscribes to the stream itself, which triggers our `console.log` as well that we added in *Step 6* using the `tap` function.

IMPORTANT NOTE

The `async` pipe has a limitation, which is that you cannot stop the subscription until the component is destroyed. In such cases, you'd want to go for in-component subscriptions using something such as the `takeWhile` / `takeUntil` operator or doing a regular `.unsubscribe` method yourself when the component is destroyed.

See also

Angular `async` pipe documentation (https://angular.io/api/common/AsyncPipe)

Using the map operator to transform data

When making the API/HTTP calls in a web application, it is often the case that the server doesn't return the data in a form that is easier to directly render to the UI. We often need some sort of transformation of the data received from the server to map it to something our UI can work with. In this recipe, you're going to learn how to use the map operator to transform responses from an HTTP call.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-map-operator` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-map-operator` to serve the project

This should open the app in a new browser tab and you should see the following:

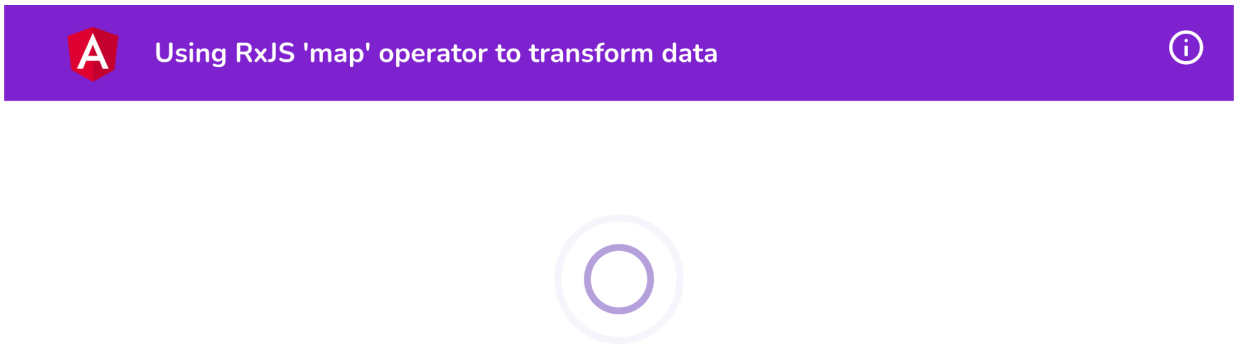


Figure 5.5 – The rx-map-operator app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

Our template (`app.component.html`)for the app is set up already. And so is our `app.component.ts` file and the data structure `appData` that we require..

1. We'll start by creating a method in the `swapi.service.ts` file to fetch the data. We want only one function to be able to bring the data from different API calls, combine it and to return it. Update the file as follows:

```
...
import { delay, forkJoin, Observable } from 'rxjs';
import { IFilm, IPerson } from './interfaces';
...
export class SwapiService {
  ...
  fetchData(personId: string): Observable<{person: IPerson}> {
  }
  fetchPerson(id: string) {...}
  fetchPersonFilms(films: string[]) {...}
}
```

You're gonna see that typescript is absolutely mad at us. Don't worry about it. We'll make it happy in time.

1. Let's add the following code to first get the person, and then get the films of that person in the `fetchData` function.:

```
...
export class SwapiService {
  ...
  fetchData(personId: string): Observable<{person: IPerson}> {
    let personInfo: IPerson;
    return this.fetchPerson(personId)
      .pipe(
        mergeMap((person) => {
          personInfo = person;
    return this.fetchPersonFilms(person.films);
        })
      )
  }
  ...
}
```

Now we can decide what to do when we receive the films back.

1. We'll map over the response from the films HTTP calls and add that to the `personInfo` object. Update the `swapi.service.ts` file as follows:

```
...
import { delay, forkJoin, map, mergeMap, Observable } from 'rxjs';
...
export class SwapiService {
 ...
  fetchData(personId: string): Observable<{ person: IPerson }> {
    let personInfo: IPerson;
    return this.fetchPerson(personId).pipe(
      mergeMap((person) => {
        personInfo = person;
        return this.fetchPersonFilms(person.films);
      }),
      map((films: IFilm[]) => {
        personInfo.filmObjects = films;
        return {
          person: personInfo,
```

```
        };
      })
    );
  }
  ...
}
```

1. Finally, let's use the `fetchData` method from the `SwapiService` inside our `app.component.ts` file. Update the `fetchData()` method in the file as follows and make sure to remove unused dependencies from the file:

```
...
export class AppComponent implements OnInit {
  ...
  fetchData() {
    this.loadingData = true;
    this.swapi.fetchData('1').subscribe((response) => {
      this.appData = response;
      this.loadingData = false;
    });
  }
}
```

And yes! If you now refresh the app, you'll notice that the data is being displayed in the view:

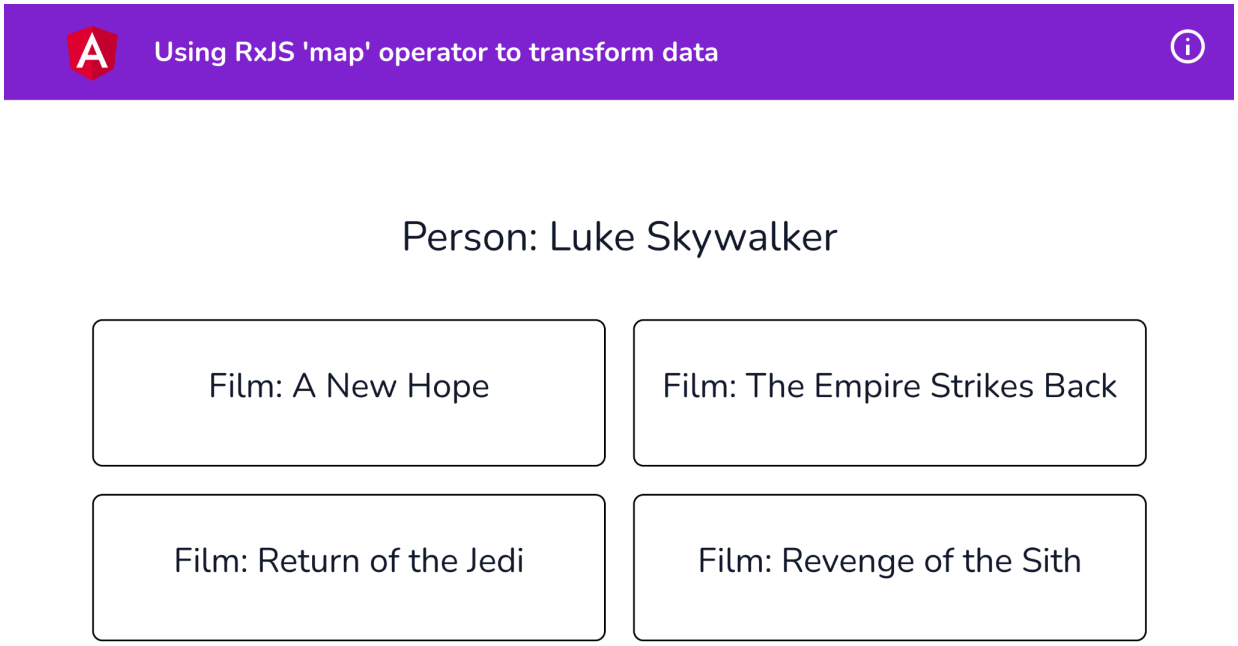


Figure 5.6 – UI showing the received data from swapi

Now that you've finished the recipe, see the next section on how this works.

How it works...

The `map` operator is one of the most used RxJS operators of all time. Especially in Angular when we make the HTTP calls. In this recipe, our target was to do as less work as possible in the `app.component.ts` file. That's because as one of the community adopted best-practices, the component should request data from the service and the service should give it in a way that it can be bound to the UI variables as is. Therefore we created this `fetchData()` method in the `SwapiService` class to use the

`fetchPerson` and `fetchPersonFilms` methods to first make the HTTP calls, and then we used the `map()` operator to transform the data in exactly the data structure our component/UI is expecting.

See also

`map` documentation (https://www.learnrxjs.io/learn-rxjs/operators/transformation/map)

Using the switchMap and debounceTime operators with autocompletes for better performance

For a lot of apps, we have features such as searching content as the user types. This is a really good **user experience** (**UX**) as the user doesn't have to press a button to do a search. However, if we send a call to the server on every key press, that's going to result in a lot of HTTP calls being sent, and we can't know which HTTP call will complete first; thus, we can't be sure if we will have the correct data shown on the view or not. In this recipe, you'll learn to use the `switchMap` operator to cancel out the last subscription and create a new one instead. This would result in canceling previous calls and keeping only one call—the last one. And we'll use the `debounceTime` operator to wait for the input to be idle before it even tries to make one call.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-switchmap-operator` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-switchmap-operator` to serve the project

This should open the app in a new browser tab and you should see the following:

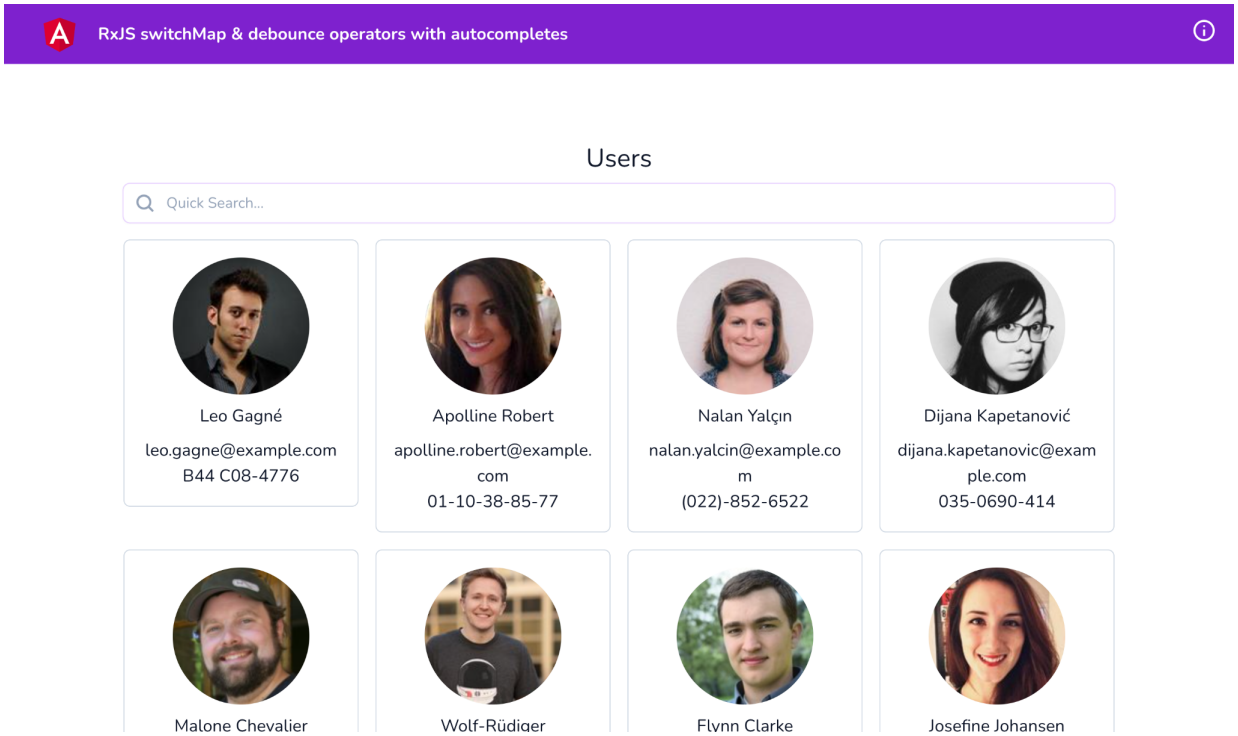


Figure 5.7 – The rx-switchmap-operator app running on http://localhost:4200

Now that we have the app running locally, open Chrome DevTools and go to the **Network** tab. Type `'wolf'` in the search input, and you'll see four calls being sent to the API server, as follows:

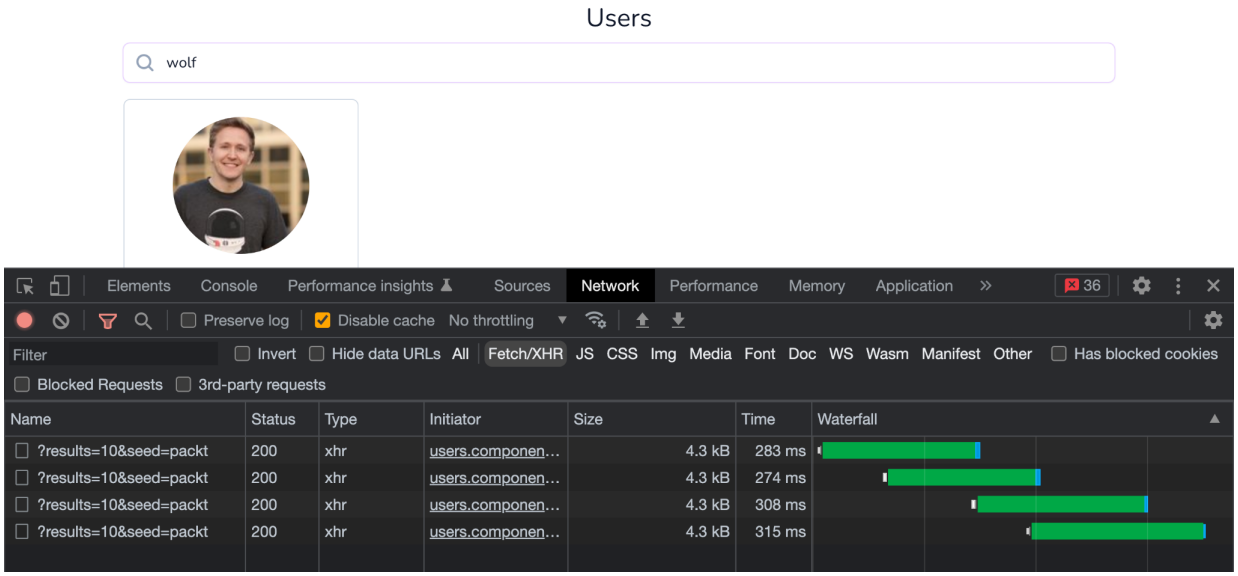


Figure 5.8 – A separate call sent for each input change

How to do it...

You can start typing into the search box on the home page to see the filtered users, and if you see the **Network** tab, you'll notice that whenever the input changes, we send a new HTTP call. Let's avoid sending a call on each keypress by using the `switchMap` operator.

1. First, import the `switchMap` operator from `rxjs/operators` in the `home/home.component.ts` file, as follows:

```
...
import { mergeMap, startWith, takeWhile, switchMap } from 'rxjs/operators';
```

1. We will now modify our subscription to the `username` form control— specifically, the `valueChanges` Observable to use the `switchMap` operator for the `this.userService.searchUsers(query)` method call. This returns an `Observable` containing the result of the HTTP call. The code should look like this:

```
ngOnInit() {
    ...
    this.searchForm.controls['username'].valueChanges
      .pipe(
        startWith(''),
        takeWhile(() => !!this.componentAlive),
        switchMap((query) => this.userService.searchUsers(query))
      )
      .subscribe((users) => {...});
  }
```

If you refresh the app now, open Chrome DevTools, and check the network type while typing `'wolf'`, you'll see that all the previous calls are canceled and we only have the latest HTTP call succeeding:

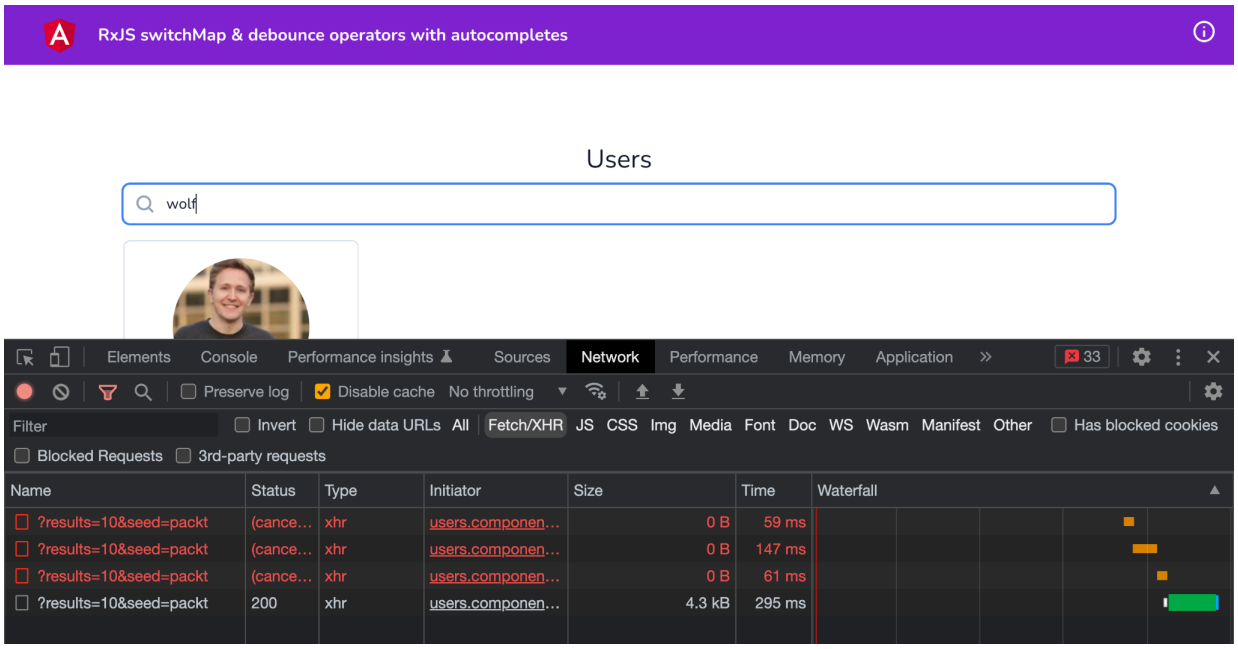


Figure 5.9 – switchMap canceling prior HTTP calls

Well, this looks better, but the `backend/api` endpoint still receives those calls.

1. We're going to use the `debounceTime` operator now to wait for the search input to be idle before starting to execute a call. Update the `users.component.ts` file as follows:

```
...
import { debounceTime, ...} from 'rxjs/operators';
...
export class UsersComponent implements OnInit {
  ...
  ngOnInit() {
    ...
    this.searchForm.controls['username'].valueChanges
      .pipe(
        startWith(''),
        debounceTime(500),
        takeWhile(() => !!this.componentAlive),
        switchMap((query) => this.userService.searchUsers(query))
      )
      .subscribe((users) => {...});
  }
}
```

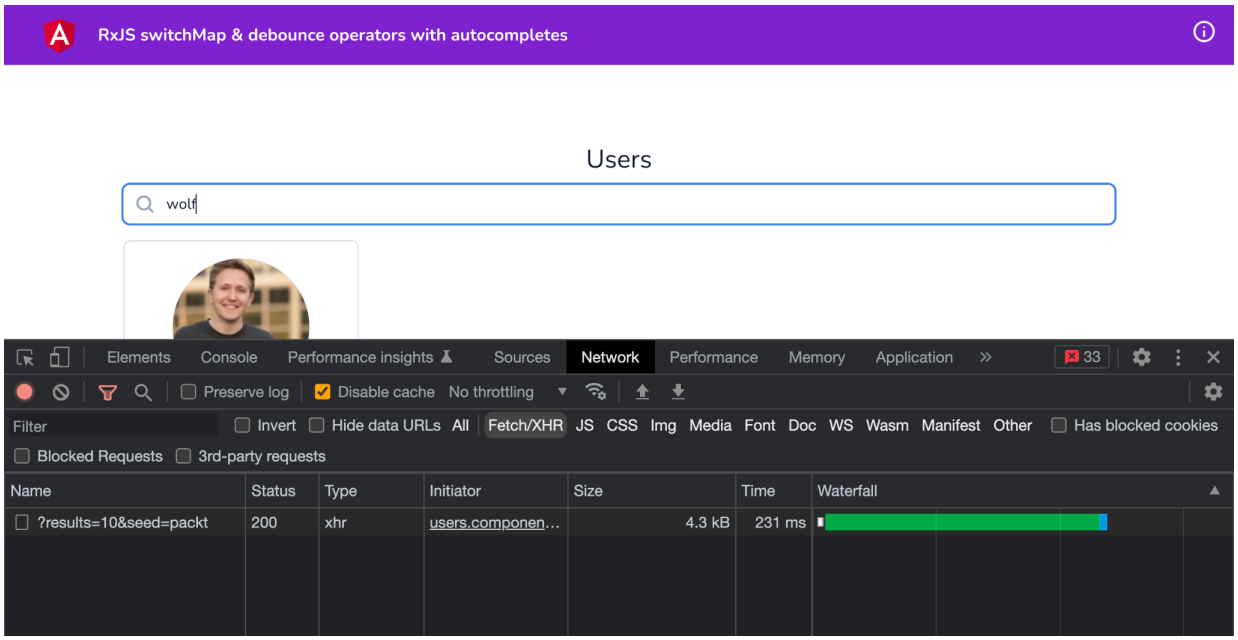


Figure 5.10 – debounceTime waiting for the input to be idle

Woot! We now have only one call that'll succeed, process the data, and end up in the view. See the next section on how it works.

How it works...

The `switchMap` operator cancels the previous (inner) subscription and subscribes to a new Observable instead. That's why it cancels all the HTTP calls sent before in our example and just subscribes to the last one. However, the call still reaches the API endpoint. If this was our server, we may still receive the API calls so we use the `debounceTime` on the form control to wait for the input to be idle before we even send our first call.

See also

`switchMap` operator documentation (https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap) `debounceTime` operator documentation (https://www.learnrxjs.io/learn-rxjs/operators/filtering/debouncetime)

RxJS custom operator

By following the other recipes in this chapter, I have to ask if you've become a fan of RxJS yet. Have you? Well, I am. And in this recipe, you're going to level up your RxJS game. You're going to create your own custom RxJS operator that just taps into any observable stream and logs the values on console. We'll call it the `logWithLabel` operator.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-custom-operator` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-custom-operator` to serve the project

This should open the app in a new browser tab. If you click the `Start Stream` button while you have the Devtools open, you should see the following:

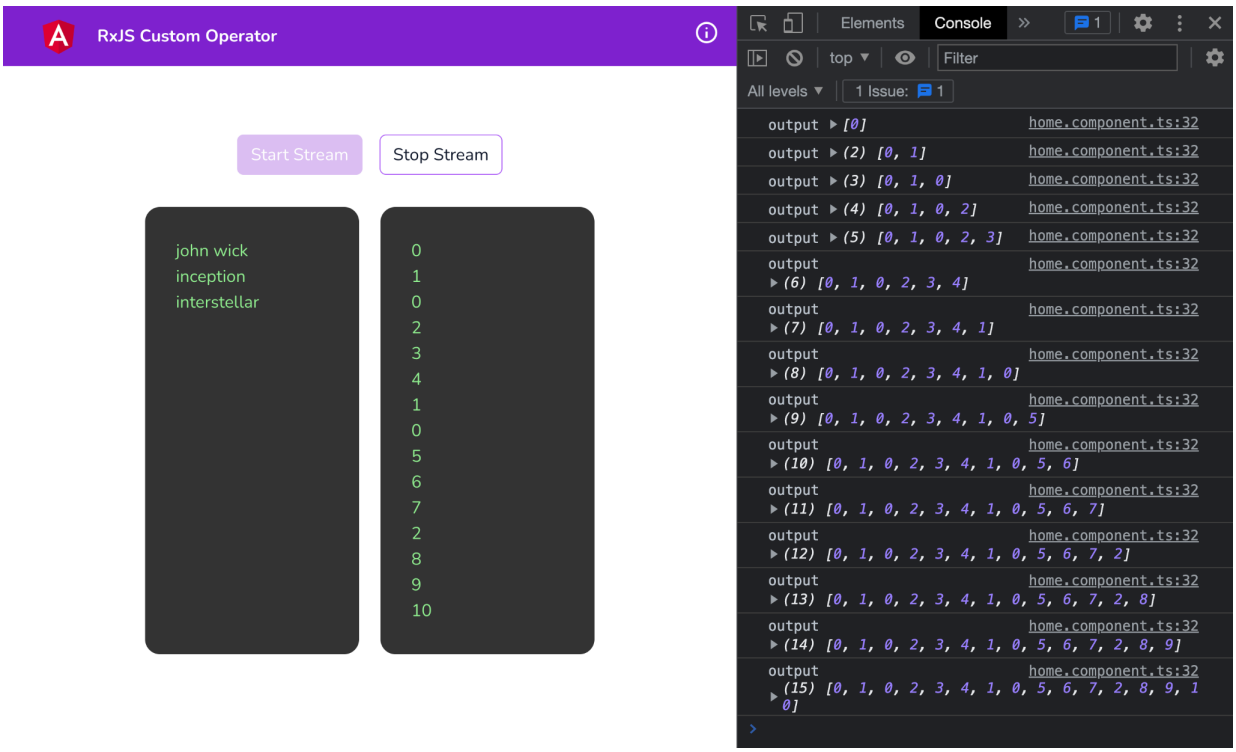


Figure 5.11 – The rx-custom-operator app running on http://localhost.4200

Let's jump into the recipe steps in the next section.

How to do it...

We're going to create a custom RxJS operator named `logWithLabel` which will log the values of the observable stream on the console with a label.

1. Create a new file inside the `app` folder and name it `log-with-label.ts`. Then add the following code inside the file:

```
import { Observable } from 'rxjs/internal/Observable';
import { tap } from 'rxjs/operators';
const logWithLabel = <T>(
  label: string
): ((source$: Observable<T>) => Observable<T>) => {
  return (source$) => source$.pipe(tap((value) => console.log(label, value)));
};
export default logWithLabel;
```

1. Now we can import the `logWithLabel` operator from the `log-with-label.ts` file inside the the `home/home.component.ts` file, as follows:

```
...
import logWithLabel from '../log-with-label';
@Component({...})
export class HomeComponent {
  ...
  startStream() {
    ...
    this.streamsOutput$ = merge(...).pipe(
```

```
      takeWhile(...),
      scan(...),
      logWithLabel('stream-output')
    );
  }
  ...
}
```

1. And that's it! If you refresh the app and click the **Start Stream** button, you can see the output using the `logWithLabel` operator as follows:

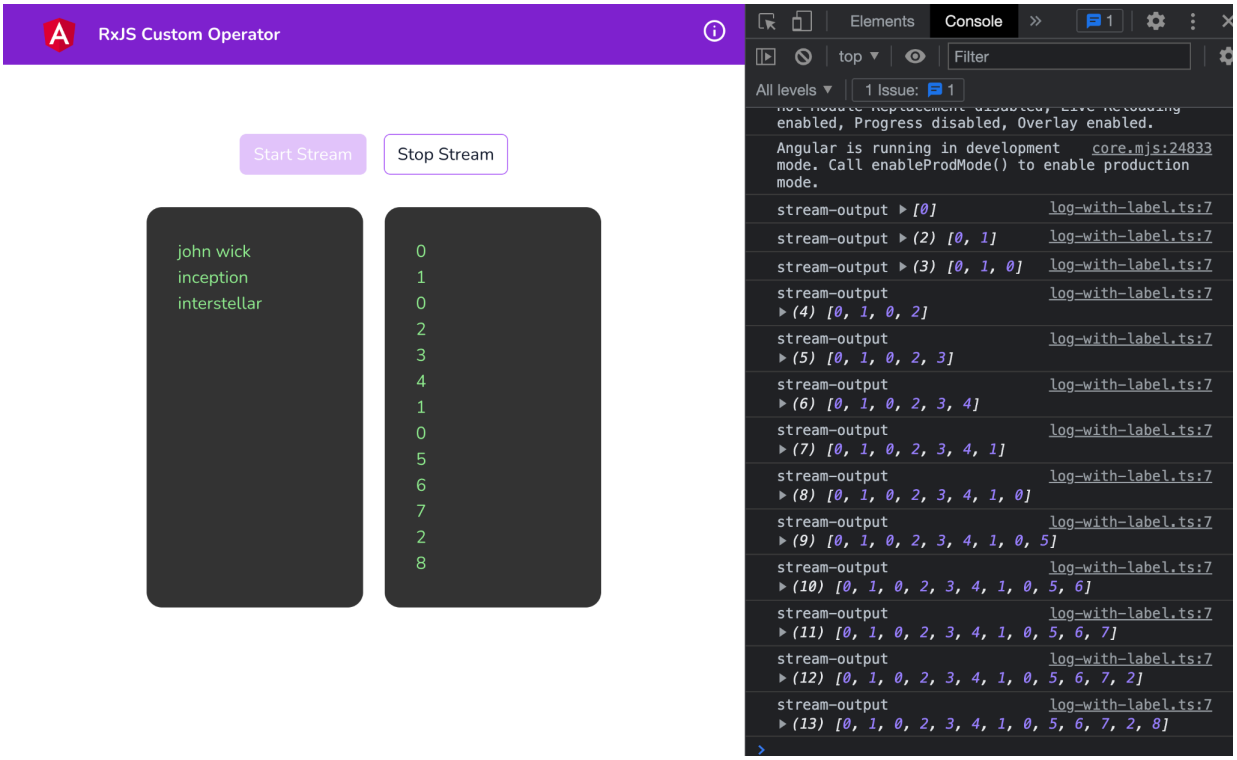


Figure 5.12 – logs using logWithLabel custom rxjs operator

See the next section to understand how it all works.

How it works...

A custom RxJS operator is essentially a function that should take an observable source stream and return back `something`. That `something` is usually an observable. In this recipe, we wanted to tap into the stream to log something on the console every time the stream emits a value. We also wanted to have a custom label for the logs for this particular stream. This is why we ended up creating the custom operator as a `factory` function that can take the `label` as input. I.e. when we call the function `logWithLabel` (let's call it `function A`) it returns a function (let's call `function B`) from within. The returned function `(B)` is what RxJS calls with the observable stream when we use the `logWithLabel` method inside the `.pipe()` method of the stream. Inside the `function B` we use the `tap` operator from RxJS to intercept the source observable and to log the values on the console using the `label` provided.

See also

`tap` operator documentation (https://rxjs.dev/api/operators/tap) `Observable` documentation (https://rxjs.dev/guide/observable)

Retry failed HTTP calls with RxJS

In this recipe, you're going to learn how to retry HTTP calls smartly with RxJS operators. We're going to use a technique called `Exponential Backoff` technique. Which means that we retry the http calls but with each next call having a delay more than the previous time for the attempt. And we stop after a number of maximum tries. Sounds exciting? Let's get into it

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-retry-http-calls` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve rx-retry-http-calls with-server` to serve the project with the backend server

This should open the app in a new browser tab and you should see the following:

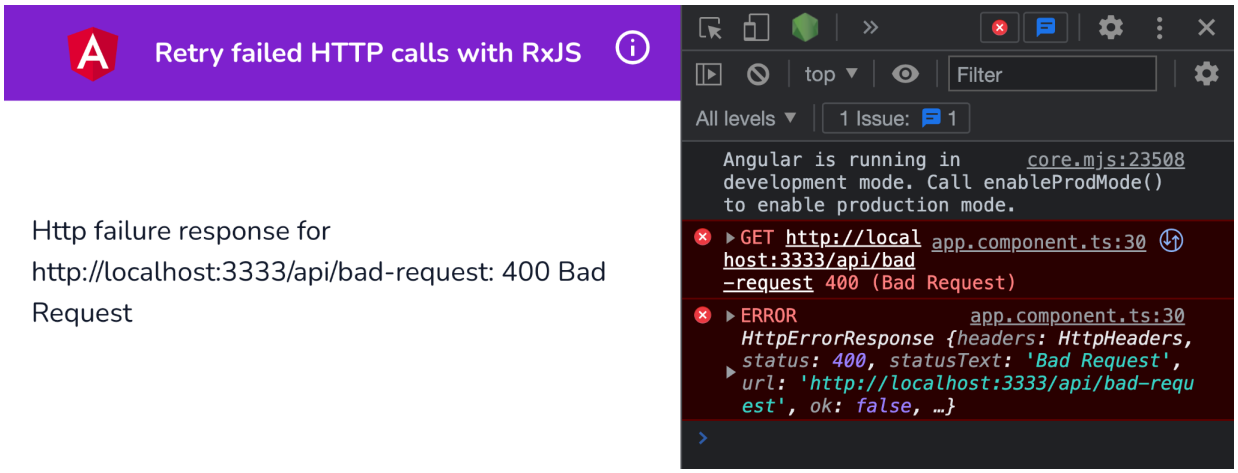


Figure 5.13 – The rx-retry-http-calls running on http://localhost.4200

Let's jump into the recipe steps in the next section.

How to do it...

We're going to create a custom RxJS operator named `backoff` which will retry the HTTP calls for us with the **exponential backoff** strategy.

1. Create a new file inside the `app` folder and name it `retry-backoff.ts`. Then add the following code inside the file:

```
import { of, pipe, throwError } from 'rxjs';
import { retry } from 'rxjs/operators';
export function retryBackoff(maxTries: number, delay: number) {
  return pipe(
    retry({
      delay: (error, retryCount) => {
        return retryCount > maxTries ? throwError(() => error) : of(retryCount);
      },
    })
  );
}
```

1. Now let's use this operator in the `app.component.ts` to retry the http calls. Update the file as follows:

```
...
import { retryBackoff } from './retry-backoff';
...
export class AppComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.isMakingHttpCall = true;
    this.http
      .get('http://localhost:3333/api/bad-request')
      .pipe(
        retryBackoff(3, 300),
        catchError(...)
      )
      .subscribe(...);
  }
}
```

Http failure response for http://localhost:3333/api/bad-request: 400 Bad Request

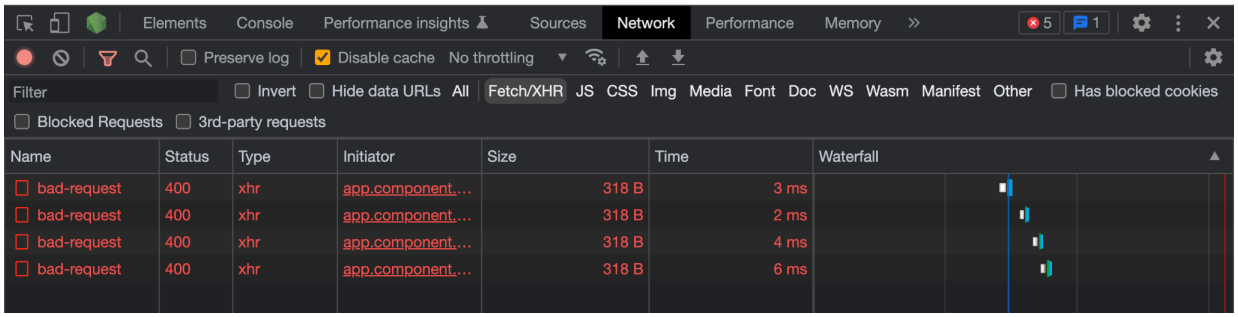


Figure 5.14 – Retrying HTTP calls multiple times instantly

You'll notice that now we retry the http calls. But all of the retries are done instantly (notice the waterfall column). We don't want that. We want every next try to be done with an increasing delay.

1. Update the `retry-backoff.ts` file to add a delay using the timer() operator and some calculations as follows:

```
import { of, pipe, throwError, timer } from 'rxjs';
import { map, mergeMap, retry } from 'rxjs/operators';
export function retryBackoff(maxTries: number, delay: number) {
  return pipe(
    retry({
      delay: (error, retryCount) => {
        return (
          retryCount > maxTries ? throwError(() => error) : of(retryCount)
        ).pipe(
          map((count) => count * count),
          mergeMap((countSq) => timer(countSq * delay))
        );
      },
```

```
    })
  );
}
```

1. And that's it! If you refresh the app and notice, you'll see that every next retry of the HTTP call has an increased delay compared to the previous one. Notice how far off the last http call is in the waterfall column (it is on the right edge of the figure).

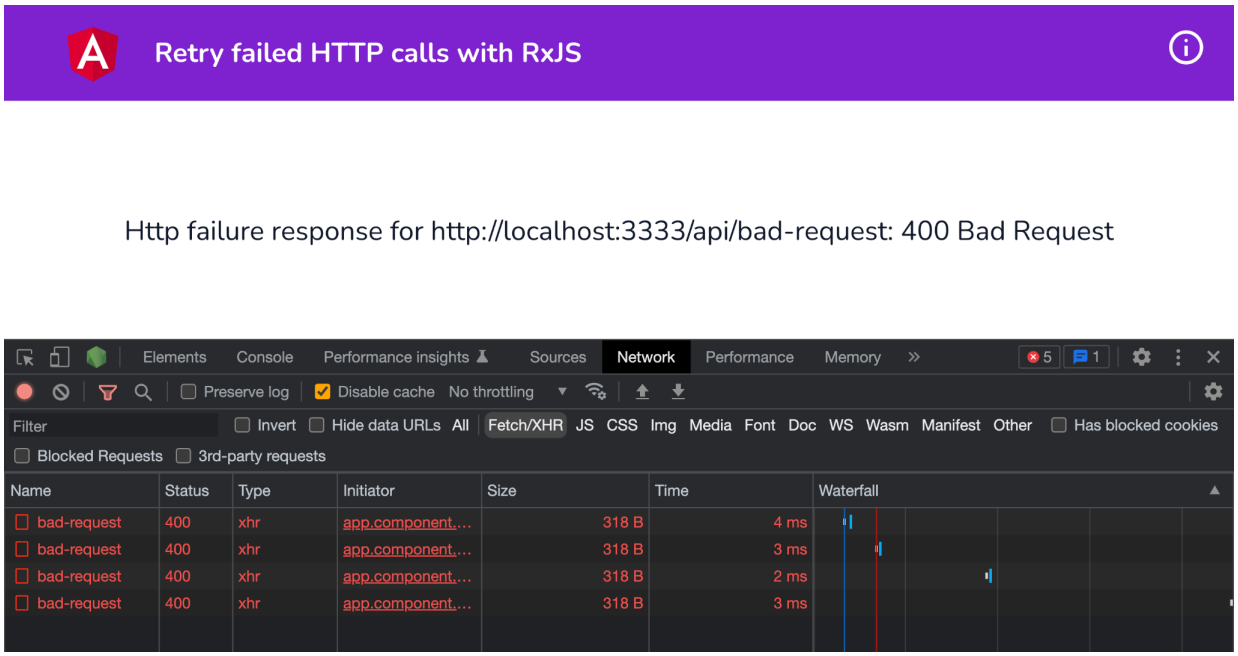


Figure 5.15 – Retrying HTTP calls with exponential backoff

See the next section to understand how it all works.

How it works...

The `retry()` operator has two overloads (at the time of writing this book). One of them takes the `number` and rxjs will just have the observable retried the number of times equal to that `number`. The other overload is that it takes a configuration object. In the configuration object we're using the `delay` method to handle our logic. The `delay` method receives the `error` and the `retryCount` from RxJS which we use to throw the error if we have already done our maximum tries, or to pass forward the `retryCount`. And we get the maximum tries from our `retryBackoff` method's arguments. Finally, we use the map and merge map work with the delay. Using `map()` operator, we take a square of the `retryCount` variable's value. And then in the `mergeMap()` operator, we multiple the square value with the delay provided to the `retryBackoff` method. As a result, every next request takes the delay equal to `(retryCount * retryCount) * delay`. Notice that we're using the `timer` method to have RxJS waiting before it can retry the http call again.

See also

- **RxJS custom operators** (https://indepth.dev/posts/1421/rxjs-custom-operators)
- **Exponential backoff** documentation (https://angular.io/guide/practical-observable-usage#exponential-backoff)

6 Reactive State Management with NgRx

Join our book community on Discord

https://packt.link/EarlyAccess

Angular and Reactive programming are best buddies, and handling an app's state reactively is one of the best things you can do with your app. NgRx is a framework that provides a set of libraries as reactive extensions for Angular. In this chapter, you'll learn how to use the NgRx ecosystem to manage your app's state reactively, and you'll also learn a couple of cool things the NgRx ecosystem will help you with.Here are the recipes we're going to cover in this chapter:

- Creating your first NgRx store with actions and reducer
- Using NgRx Store Devtools to debug the state changes
- Using NgRx selectors to select and render state in components
- Using NgRx effects to fetch data from API calls
- Using NgRx Component Store to manage state for a component

Technical requirements

For the recipes in this chapter, make sure you have **Git** and **Node.js** installed on your machine. You also need to have the `@angular/cli` package installed, which you can do with `npm install -g @angular/cli` from your terminal. The code for this chapter can be found at https://github.com/PacktPublishing/Angular-Cookbook/tree/master/chapter06.

Creating your first NgRx store with actions and reducer

In this recipe, you'll work your way through understanding NgRx's basics by setting up your first NgRx store. You'll also create some actions along with a reducer, and to see the changes in the reducer, we'll be putting in appropriate console logs.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-actions-reducer` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ngrx-actions-reducer` to serve the project

This should open the app in a new browser tab. If you add a couple of items, you and you should see the following:

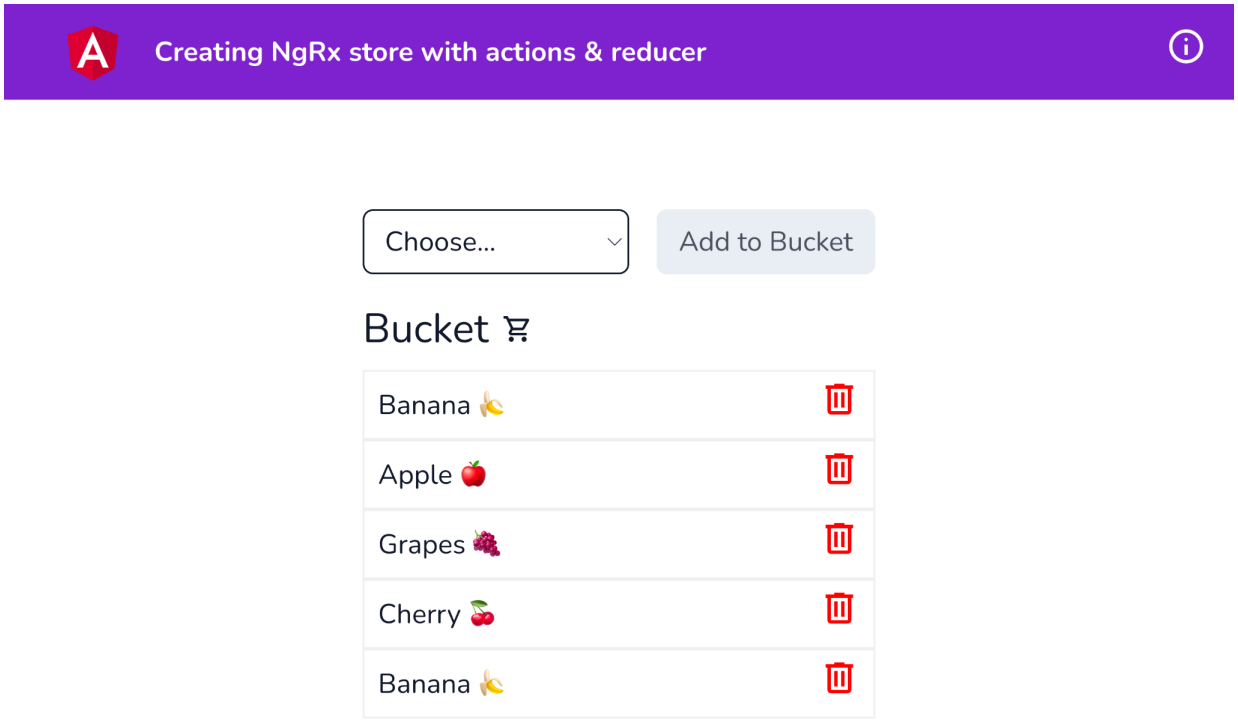


Figure 6.1 – ngrx-actions-reducers app running on http://localhost:4200

Now that we have the app running, we'll move on to the steps for the recipe.

How to do it...

1. We have an Angular application having a single page containing a bucket. You can add fruits to your buckets and can remove items from the bucket. We already have the `@ngrx/store package` installed in the workspace so you don't have to install it. However when you work on a standalone app (or on some of your own projects), you would start by adding NgRx and running the following command:

```
npm install @ngrx/store
```

1. Update the `main.ts` file to provide the NgRx store as follows:

```
...
import { provideStore } from '@ngrx/store';
...
bootstrapApplication(AppComponent, {
  providers: [
    ...
    provideAnimations(),
    provideStore({}),
  ],
}).catch((err) => console.error(err));
```

Notice that we've passed an empty object `{}` to the `provideStore()` method; we'll change that going forward.

1. Now, we'll create some actions. Create a folder named `store` inside the `app` folder. Then, create a file named `bucket.actions.ts` inside the `store` folder, and finally, add the following code to the newly created file:

```
import { createActionGroup, props } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';
export const BucketActions = createActionGroup({
  source: 'Bucket',
  events: {
    'Add Fruit': props<{ fruit: IFruit }>(),
    'Remove Fruit': props<{ fruitId: number }>(),
  },
});
```

Since we have the actions in place now, we have to create a reducer.

1. Create a new file inside the `store` folder, name it `bucket.reducer.ts`, and add the following code to it to define the necessary imports and the initial state:

```
import { createReducer, on } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';
import { BucketActions } from './bucket.actions';
export const initialState: ReadonlyArray<IFruit> = [];
```

1. Now well define the reducer. Add the following code to the `bucket.reducer.ts` file:

```
...
export const bucketReducer = createReducer(
  initialState,
  on(BucketActions.addFruit, (_state, { fruit }) => {
    console.log({ fruit });
    return [fruit, ..._state];
  }),
  on(BucketActions.removeFruit, (_state, { fruitId }) => {
    console.log({ fruitId });
    return _state.filter((fr) => fr.id === fruitId);
  })
);
```

1. We'll now update the `main.ts` file to use the reducer we just created. Update the file as follows:

```
...
import { bucketReducer } from './app/store/bucket.reducer';
bootstrapApplication(AppComponent, {
  providers: [
    ...,
    provideStore({
      bucket: bucketReducer,
    }),
  ],
}).catch((err) => console.error(err));
```

1. Now we will use the actions we created to see the console logs from the reducer functions. Update the `bucket.component.ts` file to first use the NgRx store as follows:

```
...
import { StoreModule, Store } from '@ngrx/store';
@Component({
  ...,
  imports: [CommonModule, FormsModule, StoreModule],
})
export class BucketComponent implements OnInit {
  ...
  fruits: string[] = Object.values(Fruit);
  store = inject(Store);
  ...
}
```

1. Now we can dispatch actions to the store for when we're adding and removing fruits to the bucket. We'll also modify the code a bit to avoid repetition. Update the `bucket.component.ts` file

as follows:

```
...
import { BucketActions } from '../store/bucket.actions';
...
export class BucketComponent implements OnInit {
  ...
        addSelectedFruitToBucket() {
    const newFruit: IFruit = {
      id: Date.now(),
      name: this.selectedFruit,
    };
    this.store.dispatch(
      BucketActions.addFruit({
        fruit: newFruit,
      })
    );
    this.bucketService.addItem(newFruit);
  }
  deleteFromBucket(fruit: IFruit) {
    this.store.dispatch(
      BucketActions.removeFruit({
        fruitId: fruit.id,
      })
    );
    this.bucketService.removeItem(fruit);
  }
}
```

Now if you add or remove items from the bucket, you're going to see the logs on the console as follows which means our actions and reducer are working.

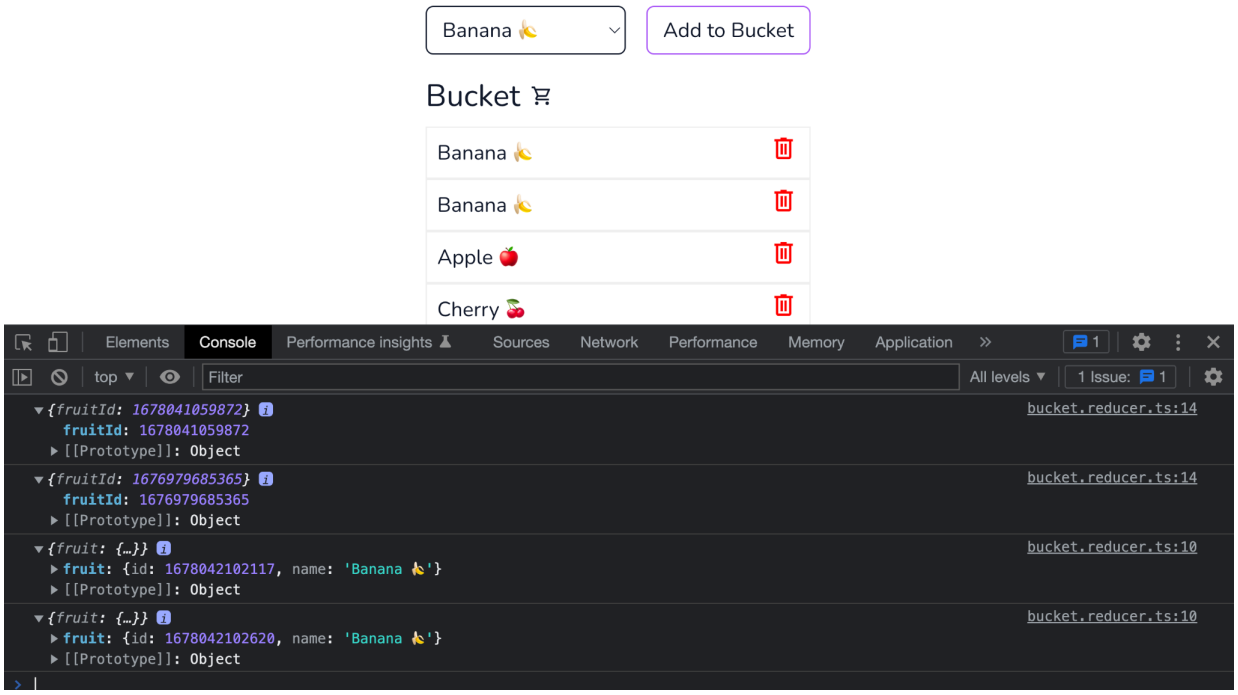


Figure 6.2 – Logs showing the actions for adding and removing items from a bucket

And that covers it all for this recipe! You now know how to integrate an NgRx store into an Angular app and how to create NgRx actions and dispatch them. You also know how to create a reducer,

define its state, and listen to the actions to act on the ones dispatched.

See also

- NgRx store walkthrough tutorial (https://ngrx.io/guide/store/walkthrough)
- NgRx reducers documentation (https://ngrx.io/guide/store/reducers)
- NgRx actions documentation (https://ngrx.io/guide/store/actions)

Using NgRx Store Devtools to debug the state changes

In this recipe, you'll learn how to use `@ngrx/store-devtools` to debug your app's state, the actions dispatch, and the difference in the state when the actions dispatch. We'll be using an existing app we're familiar with to learn about the process.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-devtools` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ngrx-devtools` to serve the project

This should open the app in a new browser tab. If you add a couple of items, you should see the following:

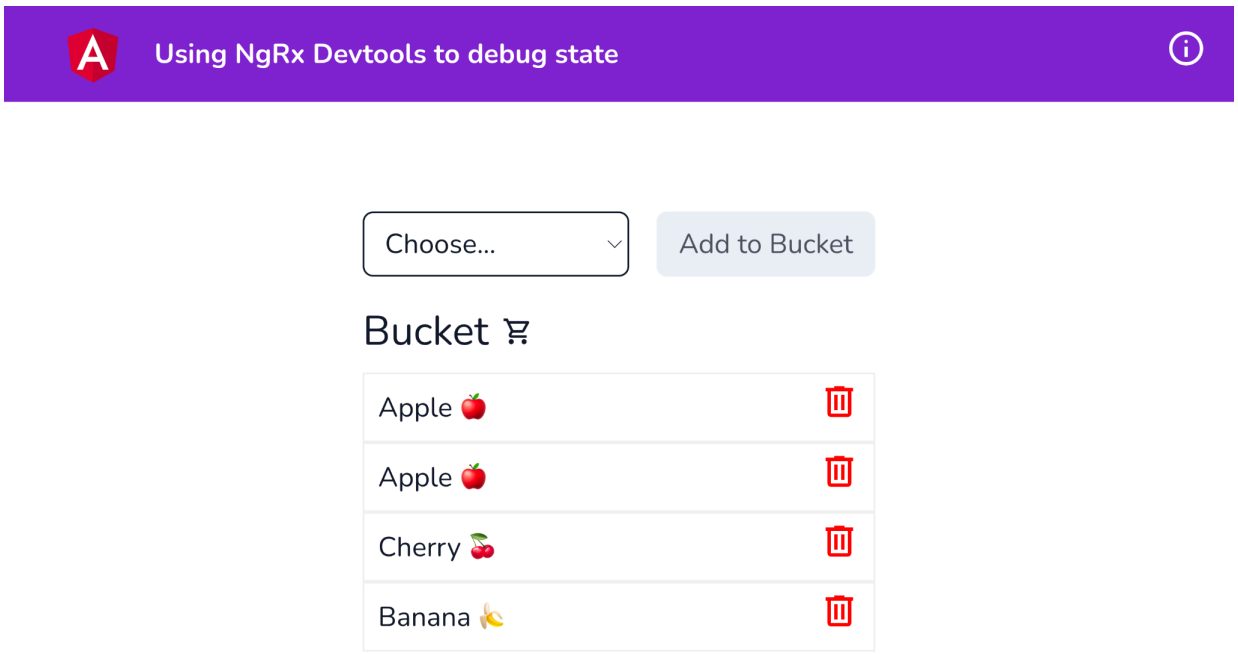


Figure 6.3 – Using ngrx-devtools app running on http://localhost:4200

Now that we have the app set up, let's see the steps of the recipe in the next section.

How to do it...

1. We have an Angular app that already has the `@ngrx/store` package integrated. We also have a reducer set up and some actions in place that are logged on the console as soon as you add or remove an item. We already have the `@ngrx/store-devtools` package installed in the workspace so you don't have to install it. However when you work on a standalone app (or on some of your your own projects), you would start by adding NgRx store devtools and running the following command:

```
npm install @ngrx/store-devtools
```

1. First, update your `app.module.ts` file to include a `StoreDevtoolsModule.instrument` entry, as follows:

```
...
import { provideStoreDevtools } from '@ngrx/store-devtools';
import { bucketReducer } from './app/store/bucket.reducer';
bootstrapApplication(AppComponent, {
  providers: [
    ...,
    provideStore({
      bucket: bucketReducer,
    }),
    provideStoreDevtools({
      maxAge: 50,
    }),
  ],
}).catch((err) => console.error(err));
```

And now, download the Redux DevTools extension from https://github.com/zalmoxisus/redux-devtools-extension/ for your particular browser and install it. I'll be consistently using the Chrome browser in this book.Open Chrome DevTools. There should be a new tab named `Redux`. Tap it and refresh the page. You'll see something like this:

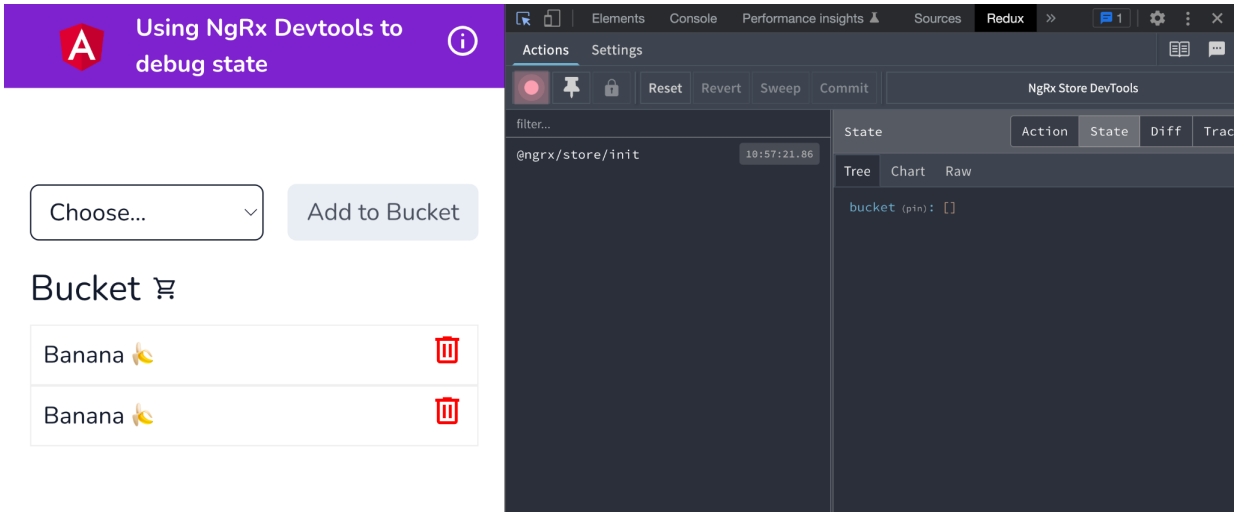


Figure 6.4 – Redux DevTools showing the initial Redux action dispatched

Remove all the fruits, add two cherries 🍒🍒 and a banana 🍌 to the bucket, and then remove the banana 🍌 from the bucket. You should see all the relevant actions being dispatched along with the state as follows:

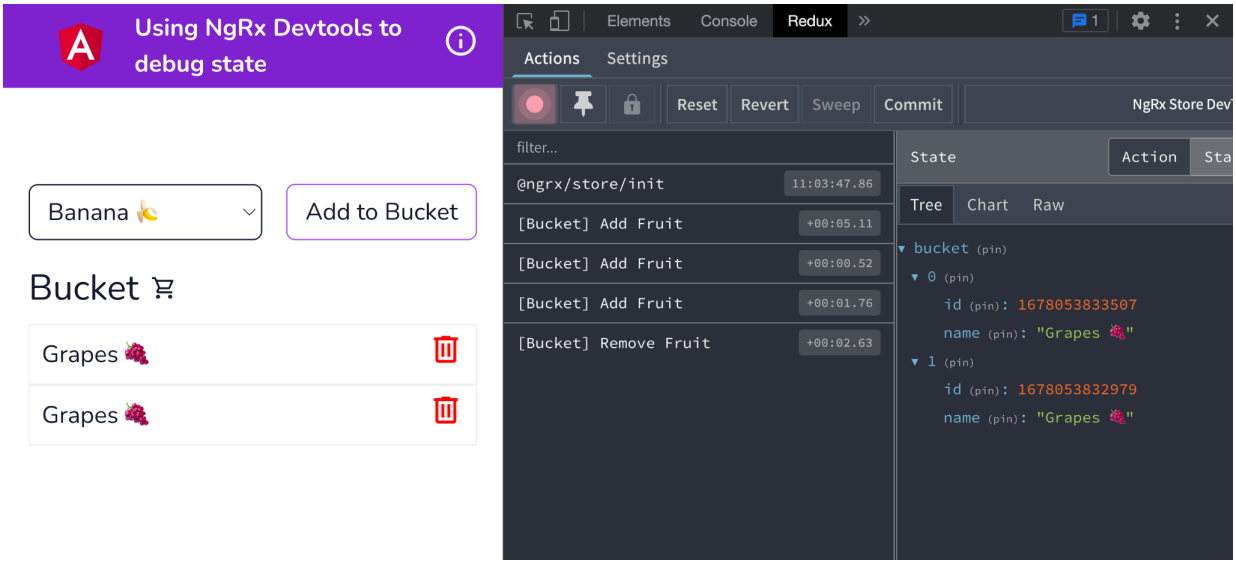


Figure 6.5 – Redux DevTools showing bucket actions and state

Great! You've just learned how to use the Redux DevTools extension to see your NgRx state and the actions being dispatched.

How it works...

It is important to understand that NgRx is a combination of Angular and Redux (Angular + RxJS). By using the Store Devtools package and the Redux DevTools extension, we're able to debug the app really easily, which helps us find potential bugs, predict state changes, and be more transparent about what's happening behind the scenes in the `@ngrx/store` package.

There's more...

1. You can also see the difference that an action caused within an app's state. I.e. for when we add an item to the bucket and we remove an item from the bucket See *Figure 6.6* and *Figure 6.7* for each cases respectively:

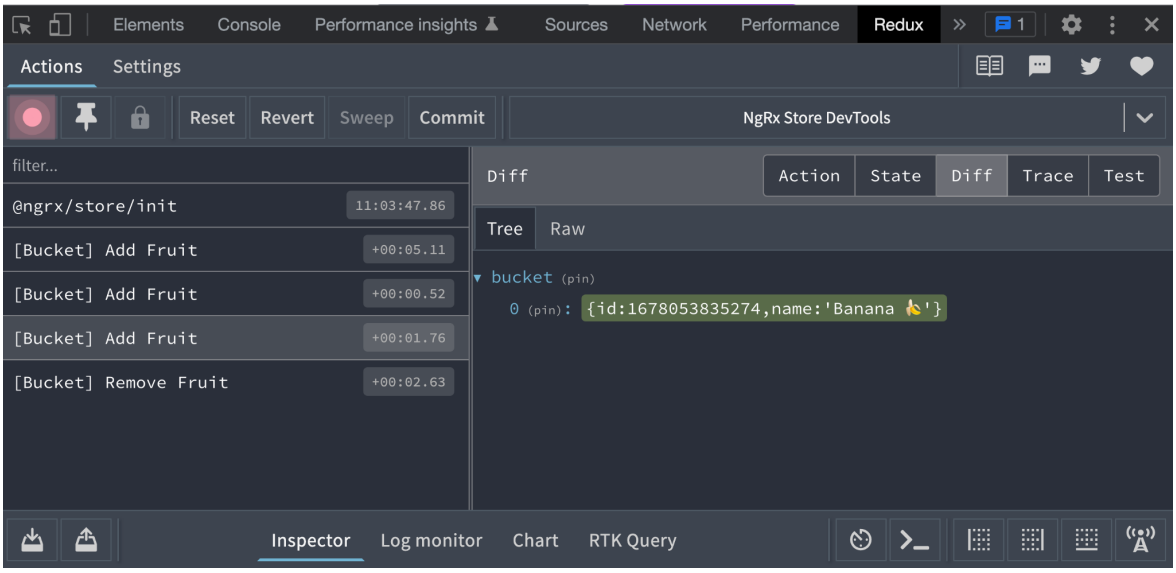


Figure 6.6 – `Add Fruit` action in Redux Devtools

2. Notice the green background around the bucket item in *Figure 6.6*. This represents an addition to the state. You can see the `ReemoveFruit` action in the following image:

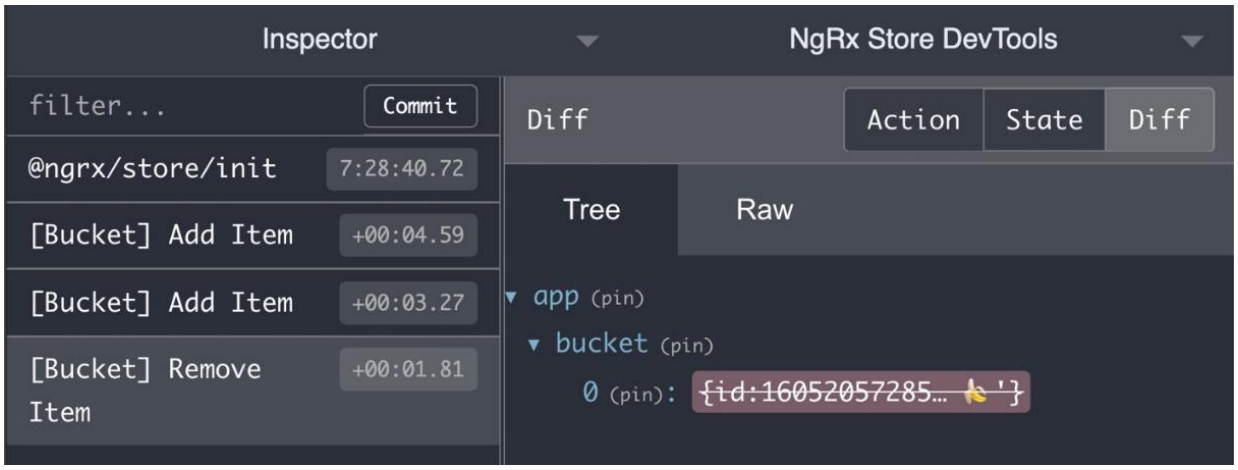


Figure 6.7 – Remove Fruit action in Redux Devtools

Now notice the red background and a strikethrough around the in the `Diff`. This represents removal from the state.

See also

NgRx Store Devtools documentation (https://ngrx.io/guide/store-devtools)

Using NgRx selectors to select and render state in components

In the previous recipes we created some actions, a reducer and integrated devtools to observe the state changes. However, our bucket application still renders the data using some variables in the `BucketService`. In this recipe, we're going all in with NgRx. We're going to render the bucket items from the state as we already are saving them in the NgRx store.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-selectors` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ngrx-selectors` to serve the project

This should open the app in a new browser tab and you should see the following:

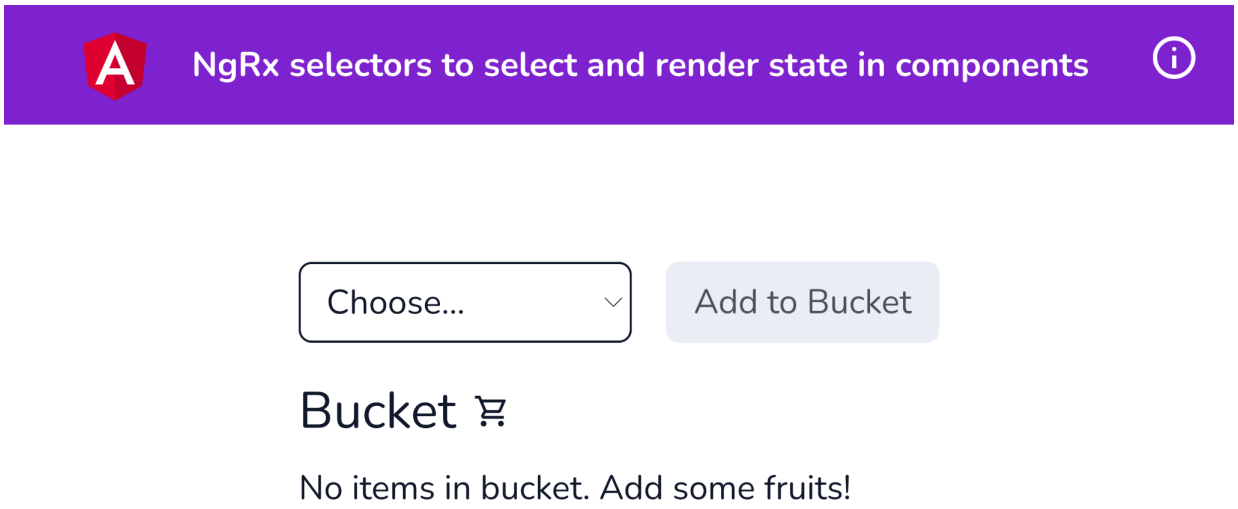


Figure 6.16 – Using ngrx-selectors app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

All we have to do in this recipe is to work with NgRx selectors. The store, actions and the reducer have already been set up. Easy peasy. Let's get started!We'll first show the users on the `Home` page and, in order to do that, we have to create our first NgRx selecto:

1. Create a new file inside the `store` folder. Name it `bucket.selectors.ts` and add the following code to it:

```
import { createFeatureSelector } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';
export const selectBucket =
 createFeatureSelector<ReadonlyArray<IFruit>>('bucket');
```

Now that we have the selector in place, let's use it in the `BucketComponent` class.

1. Modify the `bucket.component.ts` file as follows to reassign the `$bucket` observable:

```
...
import { selectBucket } from '../store/bucket.selectors';
      ...
```

```
export class BucketComponent implements OnInit {
  ...
  store = inject(Store);
  $bucket: Observable<IFruit[]> = this.store.select(selectBucket);

  ngOnInit(): void {
    this.bucketService.loadItems();
  }
  ...
}
```

Refresh app and try adding and removing items from the bucket. You can see that we still get the bucket items rendered. This time, it is from the NgRx store.

1. Now we can remove extra code that was managing the state from the `BucketService`. Update the `BucketComponent` class in the `bucket.component.ts` file as follows:

```
//  remove the `implements onInit` below
export class BucketComponent implements OnInit {
  bucketService = inject(BucketService); ← remove
  selectedFruit: Fruit = '' as Fruit;
  ...
  ngOnInit(): void { ← remove this method
    this.bucketService.loadItems();
  }
  addSelectedFruitToBucket() {
    const newFruit: IFruit = {...};
    this.store.dispatch(...);
    this.bucketService.addItem(newFruit); ← remove
  }
  deleteFromBucket(fruit: IFruit) {
    this.store.dispatch(...);
    this.bucketService.removeItem(fruit); ← remove
  }
}
```

Once you've removed the code, make sure to remove unused dependencies (imports) from the file as well. Try the app and you'll see it still works. Yayy!

1. Now we can update the `bucket.service.ts` file to not keep the bucket items in the `BehaviorSubject` we're manging there. Update the file as follows:

```
...
import { Injectable } from '@angular/core';
import { IFruit } from '../interfaces/fruit.interface';
import { IBucketService } from '../interfaces/bucket-service';
@Injectable({
  providedIn: 'root',
})
export class BucketService implements IBucketService {
  storeKey = 'bucket_ngrx-selectors';
  loadItems() {
    return JSON.parse(window.localStorage.getItem(this.storeKey) || '[]');
  }
  saveItems(items: IFruit[]) {
    window.localStorage.setItem(this.storeKey, JSON.stringify(items));
  }
}
```

1. You will see that TypeScript is angry. That's because the `BucketService` doesn't implement the `IBucketService` interface anymore. Update the `bucket-service.ts` file to update the interface as follows:

```
import { IFruit } from './fruit.interface';
export interface IBucketService {
```

```
  loadItems(): void;
  saveItems(fruit: IFruit[]): void;
}
```

Notice that we've removed the `addItem` and `removeItem` functions and added the `saveItem` method.And that's it! You've finished the recipe. You will notice that as soon as we refresh the app, we lose the bucket items. But fear not, we'll bring it back in the next recipe.

How it works...

1. In this recipe, we already had a reducer and an effect that fetches the third-party API data as users. We started by creating a selector for the users for the home screen. That was easy—we just needed to create a simple selector. Note that the reducer's state is in the following form:

```
app: {
  users: []
}
```

That's why we first used `createFeatureSelector` to fetch the `app` state, and then we used `createSelector` to get the `users` state.

1. The hard part was getting the current users and similar users. For that, we created selectors that could take the `uuid` as input. Then, we listened to the `paramMap` in the `UserDetailComponent` class for the `uuid`, and as soon as it changed, we fetched it. We then used it with the selectors by passing the `uuid` into them so that the selectors could filter the current user and similar users.

Finally, we had the issue that if someone lands directly on the `User Detail` page with the `uuid`, they won't see anything because we wouldn't have fetched the users. This is due to the fact that we only fetch the users on the home page, so anyone landing directly on a user's detail page wouldn't cause the effect to be triggered. That's why we created a method named `getUsersIfNecessary` so that it can check the state and fetch the users if they're not already fetched.

See also

NgRx selectors documentation (https://ngrx.io/guide/store/selectors)

Using NgRx effects to fetch data from API calls

In this recipe, you'll learn how to use NgRx effects using the `@ngrx/effects` package. We have an app that already has @ngrx/store and @ngrx/store-devtools installed in the app. And we are able to add and remove items from the bucket. However, in this recipe we'll use a server to receive, store, add and remove items from a bucket. I.e. the data will live in the NgRx store as well as on the backend.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-effects` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ngrx-effects with-server` to serve the project with the backend app.

This should open the app in a new browser tab. If you add a couple of items, you should see the following:

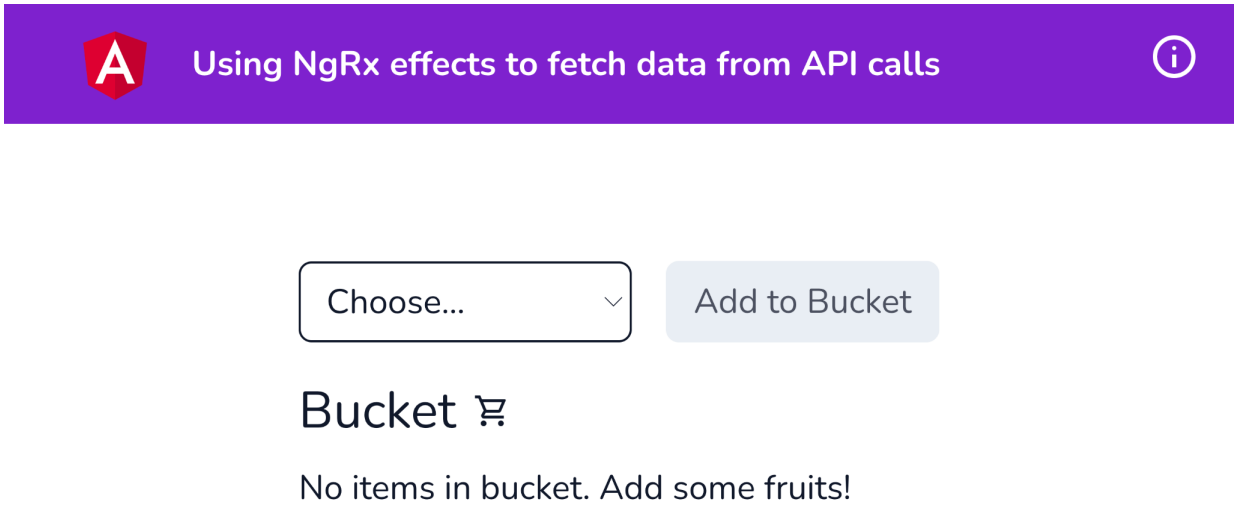


Figure 6.8 – ngrx-effects app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

1. We have an Angular application having a single page containing a bucket. You can add fruits to your buckets and can remove items from the bucket. We already have the `@ngrx/store`, `@ngrx/store-devtools` and `@ngrx/` effects packages installed in the workspace so you don't have to install them. However when you work on a standalone app (or on some of your own projects), you could add NgRx effects by running the following command:

```
npm install --save @ngrx/effects
```

1. We'll update the `bucket.actions.ts` file to add some actions including the API call. Update the file as follows:

```
import { createActionGroup, props, emptyProps } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';
export const BucketActions = createActionGroup({
  source: 'Bucket',
  events: {
    'Get Bucket': emptyProps(),
    'Get Bucket Success': props<{ bucket: IFruit[] }>(),
    'Get Bucket Failure': props<{ error: string }>(),
    'Add Fruit': props<{ fruit: IFruit }>(),
    'Add Fruit Success': props<{ fruit: IFruit }>(),
    'Add Fruit Failure': props<{ error: string }>(),
    'Remove Fruit': props<{ fruitId: number }>(),
    'Remove Fruit Success': props<{ fruitId: number }>(),
    'Remove Fruit Failure': props<{ error: string }>(),
  },
});
```

1. Create a file in the `store` folder named `bucket.effects.ts` and add the following code to it:

```
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect } from '@ngrx/effects';
import { of } from 'rxjs';
import { catchError, exhaustMap, map } from 'rxjs/operators';
```

```
import { BucketService } from '../bucket/bucket.service';
import { BucketActions } from './bucket.actions';
@Injectable()
export class BucketEffects {
  getBucket$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BucketActions.getBucket),
      exhaustMap(() =>
        this.bucketService.getBucket().pipe(
          map(({ bucket }) => BucketActions.getBucketSuccess({ bucket })),
          catchError((error) => of(BucketActions.getBucketFailure({ error })))
        )
      )
    )
  );
  constructor(
    private actions$: Actions,
    private bucketService: BucketService
  ) {}
}
```

1. Now we can provide the effects with the `BucketEffects` class in `main.ts` file as follows:

```
...
import { provideEffects } from '@ngrx/effects';
...
import { BucketEffects } from './app/store/bucket.effects';
...
bootstrapApplication(AppComponent, {
  providers: [
    ...,
    provideHttpClient(),
    provideEffects([BucketEffects]),
  ],
}).catch((err) => console.error(err));
```

1. We'll now update the `bucket.reducer.ts` file to add or remove item from the state on the success events of the HTTP calls. Also to set the bucket items retrieved from the server on app start. Update the file as follows:

```
...
export const initialState: ReadonlyArray<IFruit> = [];
export const bucketReducer = createReducer(
  initialState,
  on(BucketActions.getBucketSuccess, (_state, { bucket }) => {
    return bucket;
  }),
  on(BucketActions.addFruitSuccess, (_state, { fruit }) => {
    console.log({ fruit });
    return [fruit, ..._state];
  }),
  on(BucketActions.removeFruitSuccess, (_state, { fruitId }) => {
    console.log({ fruitId });
    return _state.filter((fr) => fr.id !== fruitId);
  })
);
```

1. We'll now dispatch the `getBucket()` action from the `bucket.componen.ts` when the component is mounted. Update the `bucket.component.ts` as follows:

```
import { CommonModule } from '@angular/common';
import { Component, inject, OnInit } from '@angular/core';
...
@Component({...})
export class BucketComponent implements OnInit {
  ...
  ngOnInit() {
```

```
    this.store.dispatch(BucketActions.getBucket());
  }
  ...
}
```

Now that we have linked everything, refresh the app and you'll see the bucket items from server in the app. And you can see the actions in the Redux Devtools as follows:

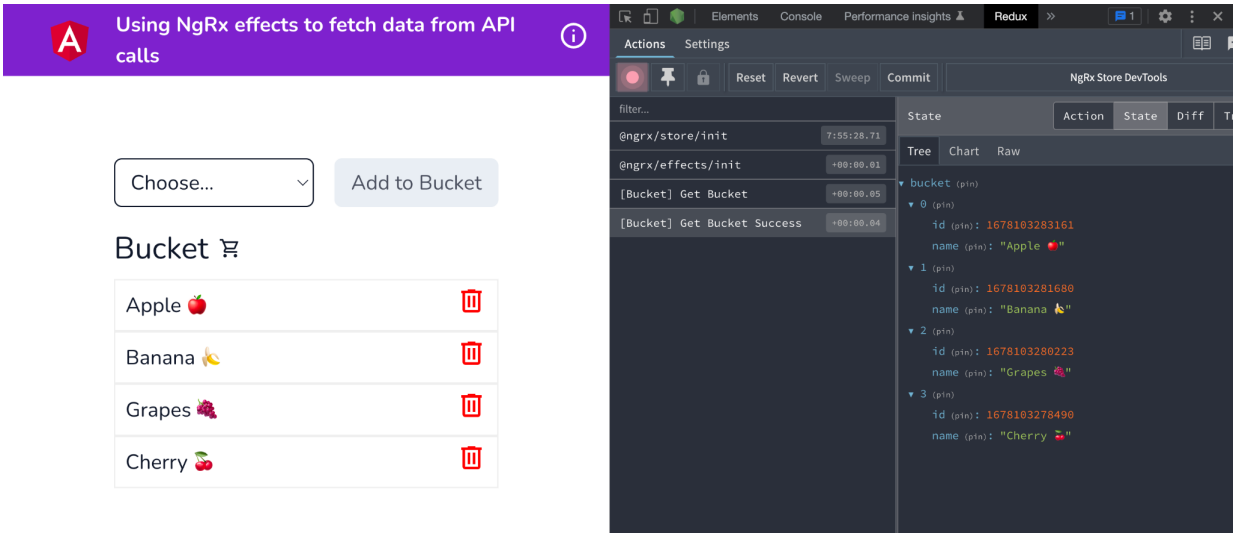


Figure 6.9 – Getting the bucket from server using NgRx Effects

1. You should also be able to see the network call (see Figure 6.10) that occurred due to the `[Bucket] Get Bucket` action in Figure 6.9.

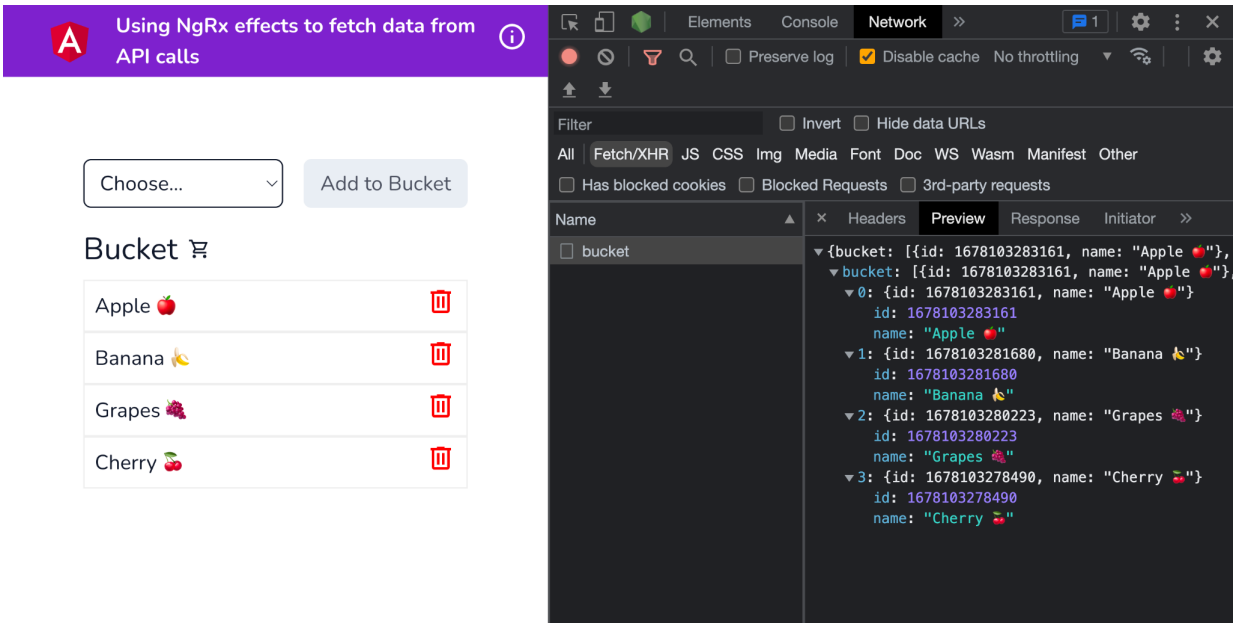


Figure 6.10 – @ngrx/effects initiating network calls

If you try to add or remove items, you'll see that they don't work. That's because we've changed our reducer to act when there's a `addFruitSuccess` or `removeFruitSuccess` even in step 4.

1. Now we will add the effect for `addFruit` in the `bucket.effects.ts`. Update the file as follows:

```
...
export class BucketEffects {
  ...
  addItem$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BucketActions.addFruit),
      exhaustMap((action) =>
        this.bucketService.addItem(action.fruit).pipe(
          map(({ fruit }) => BucketActions.addFruitSuccess({ fruit })),
          catchError((error) => of(BucketActions.addFruitFailure({ error })))
        )
      )
    )
  );
  ...
}
```

1. Let's add the effect for `removeFruit` as well in the `bucket.effects.ts`. Update the file as follows:

```
...
export class BucketEffects {
  ...
  removeItem$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BucketActions.removeFruit),
      exhaustMap((action) =>
        this.bucketService.removeItem(action.fruitId).pipe(
          map(() =>
            BucketActions.removeFruitSuccess({ fruitId: action.fruitId })
          ),
          catchError((error) => of(BucketActions.removeFruitFailure({ error })))
        )
      )
    )
  );
  ...
}
```

If you refresh the app now, add an apple, and remove it, you should see the following state in the Redux Devtools:

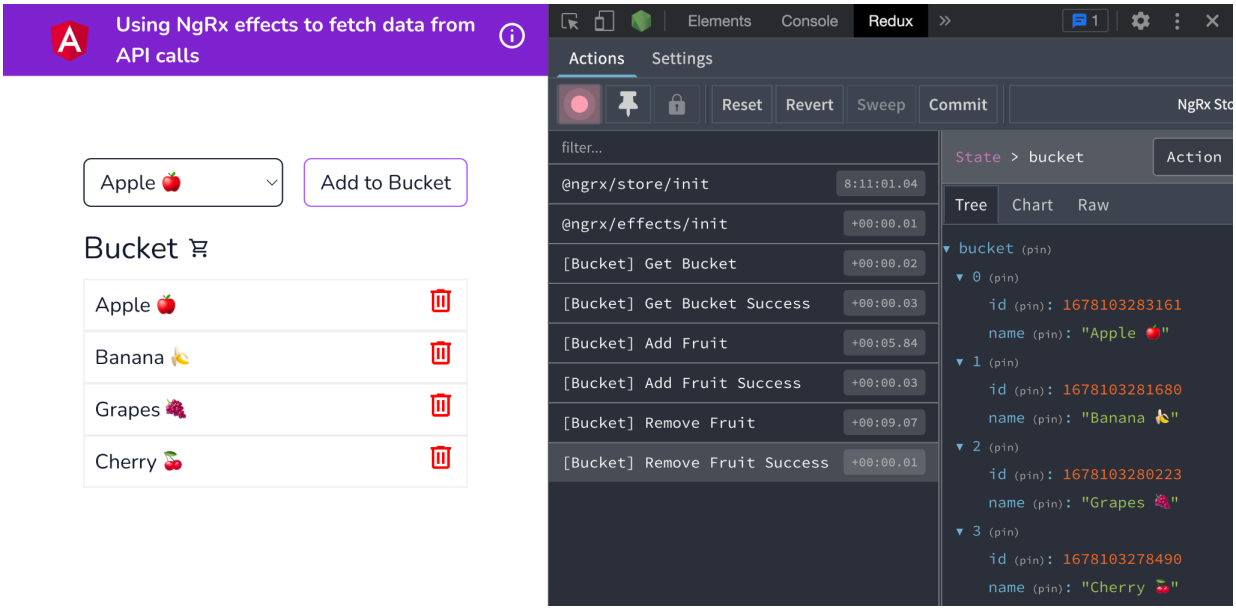


Figure 6.11 – NgRx effects getting bucket, adding and removing items

Notice in *Figure 6.11* that the state shows 4 items as shown in the UI. Also notice the sequence of the actions dispatched. Every API/HTTP action results in a following Success action in this case. Great! You now know how to use NgRx effects in your Angular apps. See the next section to understand how NgRx effects work.

How it works...

In order for the NgRx effects to work, we needed to install the `@ngrx/effects` package, create an effect, and the register it in an array of effects (root effects) in the `main.ts` file. An NgRx effect is essentially something that listens to an action, does something (we call this something a side-effect) and then returns an action as a result. When an action is dispatched to the store from any component or even from another effect, the registered effect triggers, does the job you want it to do, and is supposed to dispatch another action in return. For HTTP calls, we usually have three actions—that is, the main action, and the following success and failure actions. Ideally, on the success action (and perhaps on the failure action too), you would want to update some of your state variables. In step 4, inside the `BucketEffects` class, you'll notice that we inject the `Action` service from NgRx and the `BucketService`. Then we use the `createEffect()` to create the effect. The effect itself listens to the `Action` stream for a particular event using the `ofType()` operator. We then use the `exhaustMap()` operator to perform the HTTP call which results an observable. Finally we use the `map()` operator to return the success action and the `catchError()` operator to return the failure action.

See also

NgRx effects documentation (https://ngrx.io/guide/effects)

Using NgRx Component Store to manage state for a component

In this recipe, you'll learn how to use the NgRx Component Store and how to use it instead of the push-based Subject/BehaviorSubject pattern with services for maintaining a component's state. We'll also see how this can facilitate cross-component communication using the Component

Store.Remember that `@ngrx/component-store` is a stand-alone library and doesn't correlate with `Redux` or `@ngrx/store`, and so on.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-component-store` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ngrx-component-store` to serve the project with the backend app.

This should open the app in a new browser tab. If you add a couple of items, you should see the following:

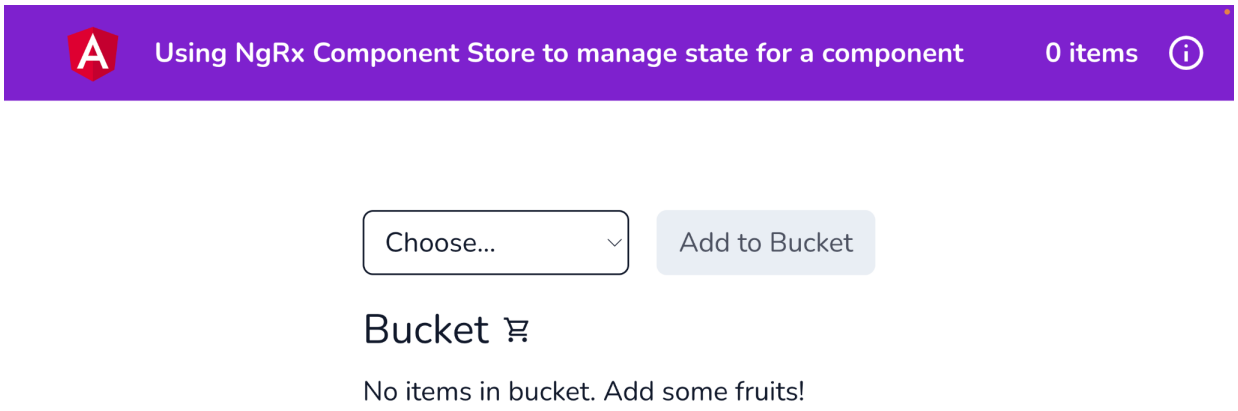


Figure 6.12 – ngrx-component-store app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

1. We have our favorite bucket app that we've used in a lot of recipes so far. The state of the buckets right now is stored in the component itself. This limits the state to be just in the component and we can't use the bucket's length in the top header. One way could be use a `BehaviorSubject` for this. But we'll reather use the NgRx Component Store. We already have the `@ngrx/component-store` package installed in the `monorepo`. But if you're working with a new project, you would install the package as follows:

```
npm install @ngrx/component-store
```

1. We're going to build our component store in our `BucketService`. Let's make it compatible with a `ComponentStore`. In order to do that, we'll create an interface for the bucket state, extend the `BucketService` from `ComponentStore`, and initialize the service by calling the `super` method. Update the `bucket.service.ts` file, as follows:

```
...
import { ComponentStore } from '@ngrx/component-store';
import { IFruit } from '../interfaces/fruit.interface';
export interface BucketState {
```

```
  bucket: IFruit[];
}
...
export class BucketService extends ComponentStore<BucketState> {
  storeKey = 'bucket_ngrx-component-store';
  constructor() {
    super({ bucket: [] });
  }
  loadItems() {...}
  saveItems(items: IFruit[]) {...}
}
```

1. None of this will make sense until we actually show the data from the `ComponentStore`. Let's work on that now.

Let's create a `bucket$` observable in the `BucketService` as follows:

```
...
import { Observable } from 'rxjs/internal/Observable';
import { IFruit } from '../interfaces/fruit.interface';
...
export class BucketService extends ComponentStore<BucketState> {
  storeKey = 'bucket_ngrx-component-store';
  readonly bucket$: Observable<IFruit[]> = this.select((state) => state.bucket);
  ...
}
```

1. First, let's make sure that we can initialize the component store using the values from the `localStorage`. We'll update the `constructor` method in the `bucket.service.ts` file as follows to use the `loadItems` method:

```
  constructor() {
    super({ bucket: [] });
    this.setState({
      bucket: this.loadItems(),
    });
  }
```

1. Now we'll add the methods for adding and removing the fruit items to and from the bucket.

```
export class BucketService extends ComponentStore<BucketState> {
  ...
  readonly addItem = this.updater((state: BucketState, fruit: IFruit) => {
    const bucketUpdated = [fruit, ...state.bucket];
    this.saveItems(bucketUpdated);
    return {
      bucket: bucketUpdated,
    };
  });
  readonly removeItem = this.updater((state: BucketState, fruitId: number) => {
    const bucketUpdated = state.bucket.filter((fr) => fr.id !== fruitId);
    this.saveItems(bucketUpdated);
    return {
      bucket: bucketUpdated,
    };
  });
  ...
}
```

1. Now that we have the Component Store set up, let's use it in the `bucket.component.ts` file as follows:

```
...
import { BucketService } from './bucket.service';
@Component({...})
export class BucketComponent {
```

```
  ...
  bucket: IFruit[] = []; ← remove
  store = inject(BucketService); ← add
  bucket$ = this.store.bucket$; ← add
  addSelectedFruitToBucket() {
    const newFruit: IFruit = {
      id: Date.now(),
      name: this.selectedFruit,
    };
    this.store.addItem(newFruit);
  }
  deleteFromBucket(fruit: IFruit) {
    this.store.removeItem(fruit.id);
  }
}
```

1. We can now use the `bucket$` obeservable in the template. Update the `bucket.component.html` as follows:

```
<div class="fruits" *ngIf="bucket$ | async as bucket" [@listItemAnimation]="bucket.length">
  <ng-container *ngIf="bucket.length > 0; else bucketEmptyMessage">
  ...
  </ng-container>
  <ng-template #bucketEmptyMessage>...</ng-template>
</div>
```

With this change, you should see the app working. If you now try to add or remove items, you should be able to see them being reflected in the app. And this is all being done using Component Store now.

1. The next thing we need to do is to use the bucket's length in the application's header. Let's create a new state selector in the `bucket.service.ts` file as follows:

```
...
export class BucketService extends ComponentStore<BucketState> {
  storeKey = 'bucket_ngrx-component-store';
  readonly bucket$: Observable<IFruit[]> = this.select((state) => state.bucket);
  readonly bucketLength$: Observable<number> = this.select(
    (state) => state.bucket.length
  );
  ...
}
```

1. And we can now use the component store in the app's header. First, let's import the store in the `app.component.ts` as follows:

```
...
import { BucketService } from './bucket/bucket.service';
...
export class AppComponent {
  store = inject(BucketService);
  bucketLength$ = this.store.bucketLength$;
}
```

1. We can now use the `bucketLength$` observable in the `app.component.html` as follows:

```
<span class="mr-4">{{bucketLength$ | async}} items</span>
```

And viola! If you add and remove items from the bucket, you should be able to see the bucket's length in the header as well.

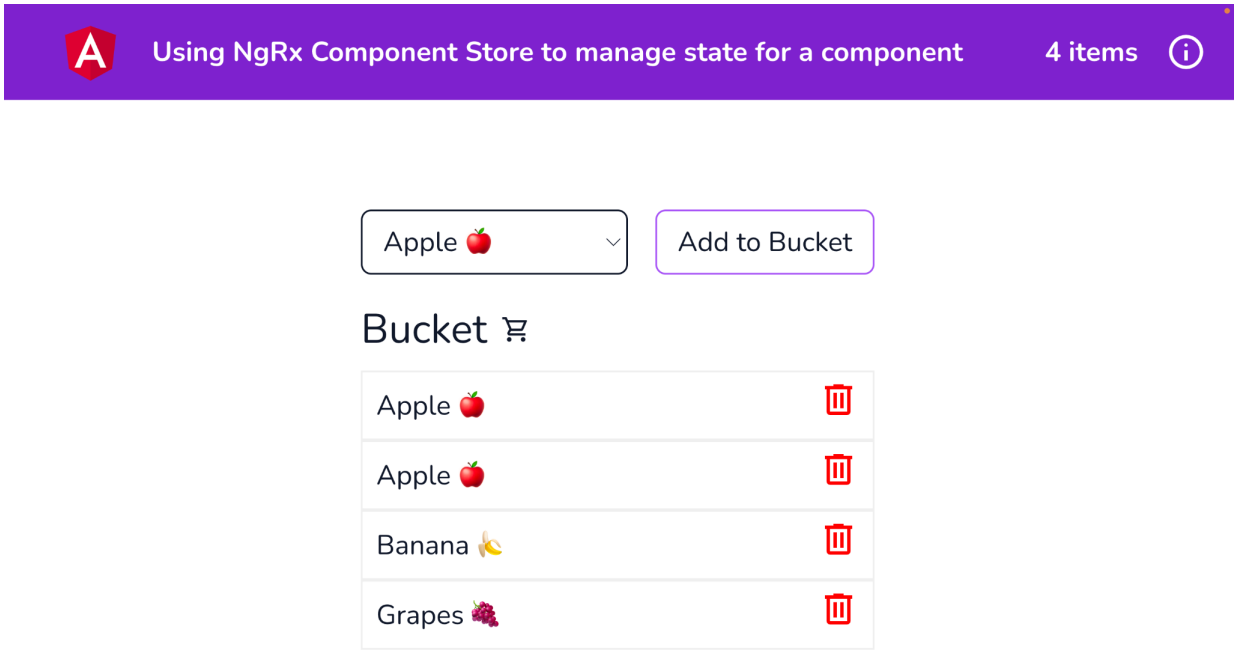


Figure 6.13 – Bucket length in header via Component Store

Congratulations! You finished the recipe. See the next section to understand how it works.

How it works...

As mentioned earlier, `@ngrx/component-store` is a standalone package that can easily be installed in your Angular apps without having to use `@ngrx/store`, `@ngrx/effects`, and so on. It is supposed to replace the usage of `BehaviorSubject` in Angular services, and that's what we did in this recipe. We covered how to initialize a `ComponentStore` and how to set the initial state using the `setState` method when we already had the values without accessing the state, and we learned how to create `updater` methods that can be used to update the state, as they can access the state and allow us to even pass arguments for our own use cases.

See also

`@ngrx/component-store` documentation (https://ngrx.io/guide/component-store)Effects in `@ngrx/component-store` documentation (https://ngrx.io/guide/component-store/effect)

Using @ngrx/router-store to work with route changes reactively

NgRx is awesome because it allows you to have your data stored in a centralized place. However, listening to route changes is still something that is out of the NgRx scope for what we've covered so far. We did rely on the `ActivatedRoute` service to watch for route changes, and when we want to test such components, the `ActivatedRoute` service becomes a dependency. In this recipe, you'll install the `@ngrx/router-store` package and will learn how to listen to the route changes using some actions built into the package.

Getting ready

The project that we are going to work with resides in `chapter06/start_here/ngrx-router-store`, inside the cloned repositor:

1. Open the project in VS Code.
2. Open the terminal and run `npm install` to install the dependencies of the project.
3. Once done, run `ng serve -o`.

This should open the app in a new browser tab, and you should see something like this:

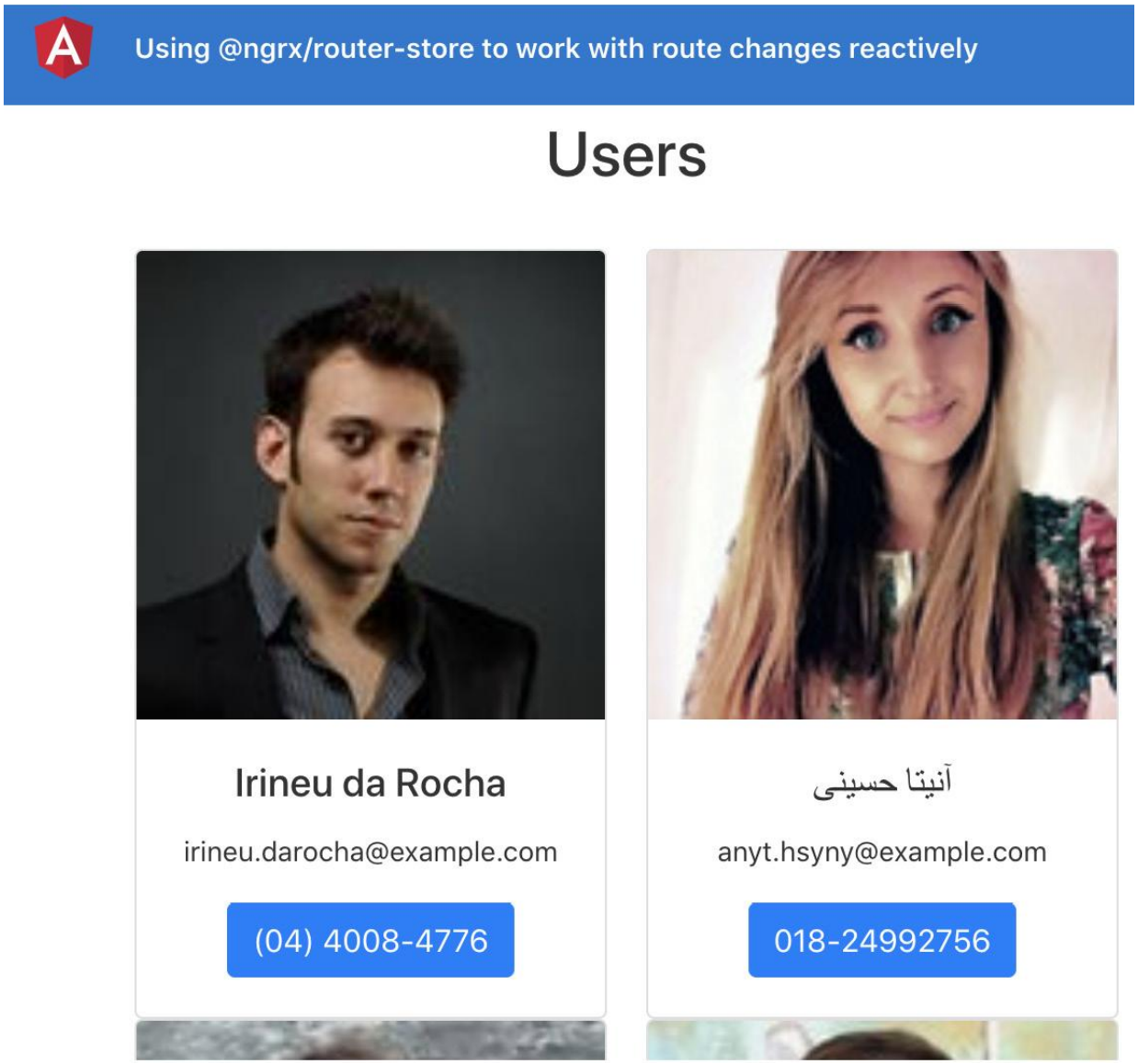


Figure 6.19 – ngrx-router-store app running on http://localhost:4200

Now that the app is running, see the next section for the steps of the recipe.

How to do it...

In order to utilize the power of NgRx even for route changes, we'll utilize the `@ngrx/router-store` package to listen to route changes. Let's begin!

1. First, install the `@ngrx/router-store` package by running the following command in your project root:

```
npm install @ngrx/router-store@12.0.0 --save
```

1. Now, import `StoreRouterConnectingModule` and `routerReducer` from the `@ngrx/router-store` package in your `app.module.ts` file and set up the `imports`, as follows:

```
...
import { StoreRouterConnectingModule, routerReducer } from '@ngrx/router-store';
@NgModule({
  declarations: [...],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    StoreModule.forRoot({
      app: appStore.reducer,
  router: routerReducer
    }),
StoreRouterConnectingModule.forRoot(),
    StoreDevtoolsModule.instrument({
      maxAge: 25, // Retains last 25 states
    }),
    EffectsModule.forRoot([AppEffects])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1. As soon as you refresh the app now and inspect it via the Redux DevTools extension, you should see some additional actions named `@ngrx/router-store/*` being dispatched as well. You should also see that the `router` property in the state has the current routes' information, as depicted in the following screenshot:

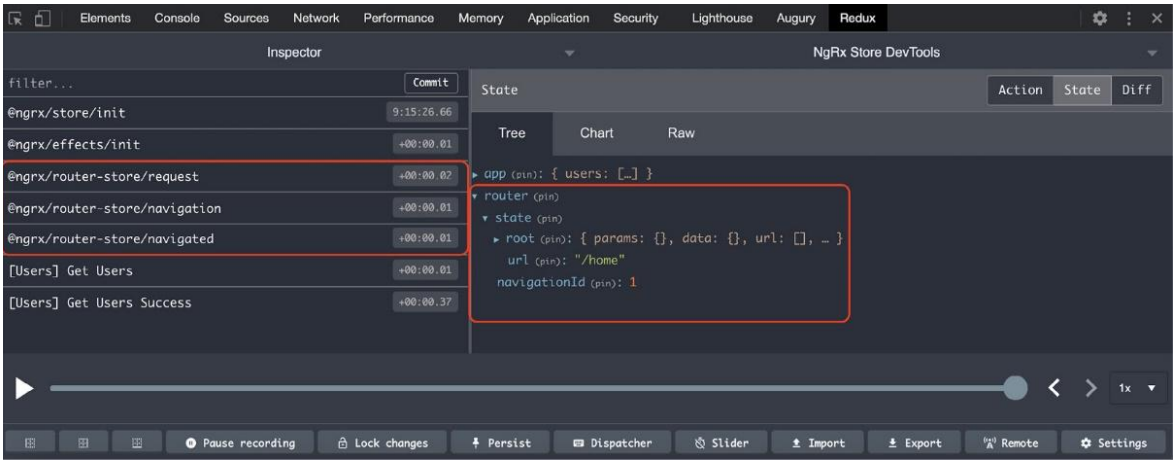


Figure 6.20 – @ngrx/router-store actions and the router state reflected in the NgRx store

2. We now have to modify our reducer—or, more precisely, the `AppState` interface—to reflect that we have the `router` property as well from the `@ngrx/router-store` package. To do so, modify the `store/app.reducer.ts` file, as follows:

```
...
import { getUsersSuccess } from './app.actions';
import { RouterReducerState } from '@ngrx/router-store'
```

```
export interface AppState {
  users: IUser[];
  router: RouterReducerState<any>;
}
const initialState: AppState = {
  users: null,
  router: null
}
...
```

1. Essentially, we have to get rid of the `ActivatedRoute` service's usage from our `UserDetailComponent` class. In order to do so, we'll first modify our selectors to get the params from the router state directly. Modify the `app.selectors.ts` file, as follows:

```
...
import { getSelectors, RouterReducerState } from '@ngrx/router-store';
export const selectApp = createFeatureSelector<AppState>('app');
export const selectUsers = createSelector(
  selectApp,
  (state: AppState) => state.users
);
...
export const selectRouter = createFeatureSelector<
  AppState,
  RouterReducerState<any>
>('router');
const { selectRouteParam } = getSelectors(selectRouter);
export const selectUserUUID = selectRouteParam('uuid');
export const selectCurrentUser = createSelector(
  selectUserUUID,
  selectUsers,
  (uuid, users: IUser[]) => users ? users.find(user => {
    return user.login.uuid === uuid;
  }) : null
);
export const selectSimilarUsers = createSelector(
  selectUserUUID,
  selectUsers,
  (uuid, users: IUser[]) => users ? users.filter(user =>   {
    return user.login.uuid !== uuid;
  }): null
);
```

1. You should see some errors on the console right now. That's because we changed the signature of the `selectSimilarUsers` and `selectCurrentUser` selectors, but it'll be fixed in the next step.

Modify the `user-detail/user-detail.component.ts` file to use the updated selectors correctly, as follows:

```
...
export class UserDetailComponent implements OnInit, OnDestroy {
  ...
  ngOnInit() {
    ...
    this.route.paramMap.pipe(
      takeWhile(() => !!this.isComponentAlive)
    )
    .subscribe(params => {
      const uuid = params.get('uuid');
      this.user$ = this.store.select(selectCurrentUser)
      this.similarUsers$ = this.store.      select(selectSimilarUsers)
    })
  }
  ...
}
```

1. This change should have resolved the errors on the console, and you should actually see the app working perfectly fine, even though we're not passing any `uuid` from the `UserDetailComponent` class anymore.

With the changes from the previous step, we can now safely remove the usage of the `ActivatedRoute` service from the `UserDetailComponent` class, and the code should now look like this:

```
...
import { Observable } from 'rxjs/internal/Observable';
import { first } from 'rxjs/operators';
import { Store } from '@ngrx/store';
...
export class UserDetailComponent implements OnInit, OnDestroy {
  ...
  constructor(
    private store: Store<AppState>
  ) {}
  ngOnInit() {
    this.isComponentAlive = true;
    this.getUsersIfNecessary();
    this.user$ = this.store.select(selectCurrentUser)
    this.similarUsers$ = this.store.    select(selectSimilarUsers)
  }
  ...
}
```

Woohoo! You've finished the recipe now. See the next section to find out how this works.

How it works...

`@ngrx/router-store` is an amazing package that does a lot of magic to make our development a lot easier with NgRx. You saw how we could remove the `ActivatedRoute` service completely from the `UserDetailComponent` class by using the selectors from the package. Essentially, this helped us get the `route params` right in the selectors, and we could use it in our selectors to get and filter out the appropriate data. Behind the scenes, the package listens to the route changes in the entire Angular app and fetches from the route itself. It then stores the respective information in the NgRx Store so that it remains in the Redux state and can be selected via the package-provided selectors easily. In my opinion, it's freaking awesome! I say this because the package is doing all the heavy lifting that we would have to do otherwise. As a result, our `UserDetailComponent` class now relies only on the `Store` service, which makes it even easier to test because of fewer dependencies.

See also

`@ngrx/router-store` documentation (https://ngrx.io/guide/router-store/)

7 Understanding Angular Navigation and Routing

Join our book community on Discord

https://packt.link/EarlyAccess

One of the most amazing things about Angular is that it is an entire ecosystem (a framework) rather than a library. In this ecosystem, the Angular router is one of the most critical blocks to learn and understand. In this chapter, you'll learn some really cool techniques about routing and navigation in Angular. You'll learn about how to guard your routes, listen to route changes, and configure global actions on route changes.The following are the recipes we're going to cover in this chapter:

- Creating routes in an Angular (standalone) app
- Lazily loaded routes in Angular
- Authorized access to routes using route guards
- Working with route parameters
- Showing a global loader between route changes
- Preloading route strategies

Technical requirements

For the recipes in this chapter, make sure you have **Git** and **Node.js** installed on your machine. You also need to have the `@angular/cli` package installed, which you can do with `npm install -g @angular/cli` from your terminal. The code for this chapter can be found at https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter07.

Creating routes in an Angular (standalone) app

If you ask me about how we used to create projects for web applications 7-8 years ago, you'll be astonished to learn how difficult it was. Luckily, the tools and standards have evolved in the software development industry and when it comes to Angular, starting a project is super easy. And with the Angular Standalone apps, the configuration is much smaller for the application's bootstrap process and for routing. In this recipe, you're going to implement some routes in a fresh Angular application.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-basic-routing` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-basic-routing` to serve the project

This should open the app in a new browser tab. And you should see the following:

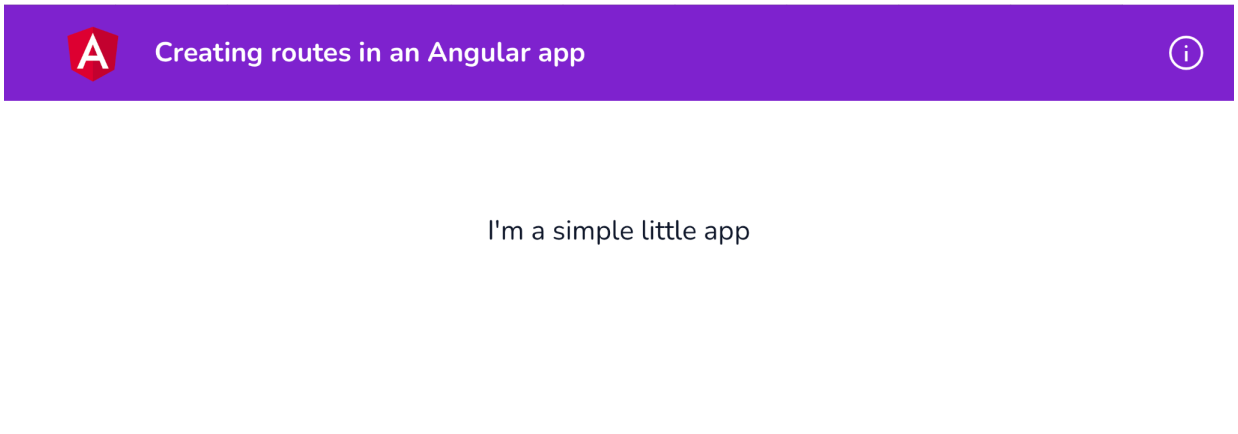


Figure 7.1 – ng-basic-routes app running on http://localhost:4200

How to do it...

We'll configure routing in the app, will add some routes and will add the links to those routes in the headers. Let's get started:

1. First, create a file named `app.routes.ts` inside the `app` folder and add the following code to it:

```
import { Route } from '@angular/router';
export const appRoutes: Route[] = [];
```

1. Now update the `main.ts` file in the `src` folder with the following changes:

```
import { bootstrapApplication } from '@angular/platform-browser';
import {
  provideRouter,
  withEnabledBlockingInitialNavigation,
} from '@angular/router';
import { AppComponent } from './app/app.component';
import { appRoutes } from './app/app.routes';
bootstrapApplication(AppComponent, {
  providers: [provideRouter(appRoutes, withEnabledBlockingInitialNavigation())],
}).catch((err) => console.error(err));
```

1. Now let's create some components (as pages). We're going to create a home page and an about page. Running the following command in the terminal from the `start` folder in the workspace. If asked, you can use the `@nrwl/angular:component` schematics:

```
npx nx g c home --project ng-basic-routing --standalone
npx nx g c about --project ng-basic-routing --standalone
```

You should see two new folders created in the `app` folder of the project now.

1. We will configure the routes now. Update the app.routes.ts file that we created in Step 1 as follows:

```
import { Route } from '@angular/router';
import { AboutComponent } from './about/about.component';
import { HomeComponent } from './home/home.component';
export const appRoutes: Route[] = [{
  path: '',
  pathMatch: 'full',
  redirectTo: 'home'
}, {
  path: 'home',
  component: HomeComponent
}, {
  path: 'about',
```

```
  component: AboutComponent
}
];
```

1. All that remains is to hook the routes in the view is to use the `<router-outlet>` in the template. Update the `app.component.html` as follows:

```
<!-- Toolbar -->
...
<main class="content" role="main">
  <router-outlet></router-outlet>
</main>
```

But wait! That crashes the app. That's because the AppComponent doesn't understand routing yet.

1. Update the `app.component.ts` file to add the `RouterModule` in it as follows:

```
...
import { RouterModule } from '@angular/router';
@Component({
  ...,
  imports: [CommonModule, RouterModule],
})
export class AppComponent {}
```

And voila! You should be able to see the home component now as follows:

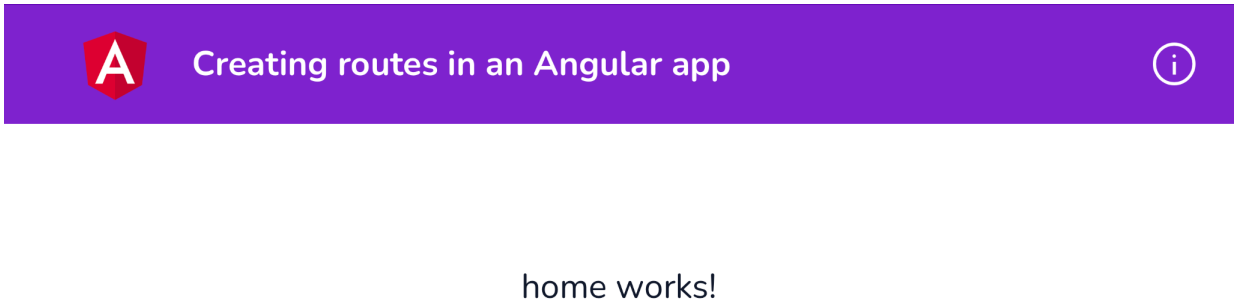


Figure 7.2 – home component in ng-basic-routes app

If you see the URL, it defaults to `http://localhost:4200/home` even if you try to go to `https://localhost:4200`

1. Finally, we'll add the links for the routes in the header. Update the `app.component.html` as follows:

```
<!-- Toolbar -->
<div class="toolbar" role="banner">
  ...
  <div class="spacer"></div>
  <div class="route-links">
    <div
      class="route-links__route-link"
      routerLink="/home"
      routerLinkActive="route-links__route-link--active"
    >
      Home
    </div>
    <div
      class="route-links__route-link"
```

```
      routerLink="/about"
      routerLinkActive="route-links__route-link--active"
    >
      About
    </div>
  </div>
  ...
<main class="content" role="main">...</main>
```

And now if you click the About link on the top, you should be able to see the about route as follows:

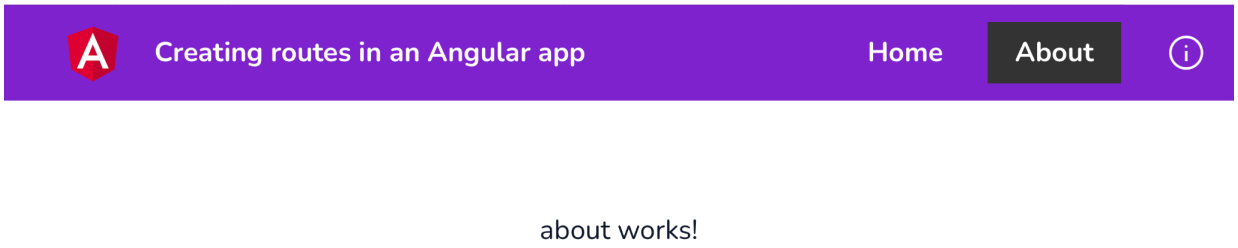


Figure 7.3 – about component in ng-basic-routes app

Awesome! Within a few minutes, and with the help of Angular router and NX CLI, we were able to create a home page, an about page and configured routing as well. The wonders of the modern web!Now that you know how basic routing is implemented, see the next section to understand how it works.

How it works...

When we create an Angular app without the `--routing` argument, we don't get the `app.routes.ts` file or have the configuration added in the `main.ts` file. For this example, we worked with an app with this setting. If you were to start a new project, you might as well just use the `--routing` flag to have them set already. Then we created some standalone components using `nx g c <component name> -standalone` command. If you were using Angular CLI instead of an NX monorepo, you would just replace `` `nx` `` with `` `ng` `` in the mentioned command, and the rest would be the same. Since we have a Standalone App, i.e. the AppComponent is a standalone component (notice the `standalone: true` in the `app.component.ts` file), we had to import the `RouterModule` to it to be able to use the `<router-outlet>` and the `routeLink` attribute on the links. We configured the routes by using the `provideRouter()` method in the `main.ts` file with the routes since we don't have an `AppModule` (because of the app being a standalone app).

See also

Angular router docs (https://angular.io/guide/router)Angular Standalone Components (https://angular.io/guide/standalone-components)

Lazily loaded routes in Angular

In the previous recipe, we learned how to create a basic routing app with eagerly loaded routes. In this recipe, you'll learn how to work with feature modules to lazily load them instead of loading them when the app loads. For this recipe, we'll assume that we already have the routes in place and we just need to load them lazily.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-lazy-routing` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-lazy-routing` to serve the project

This should open the app in a new browser tab. If you open the network tab, you and you should see the following:

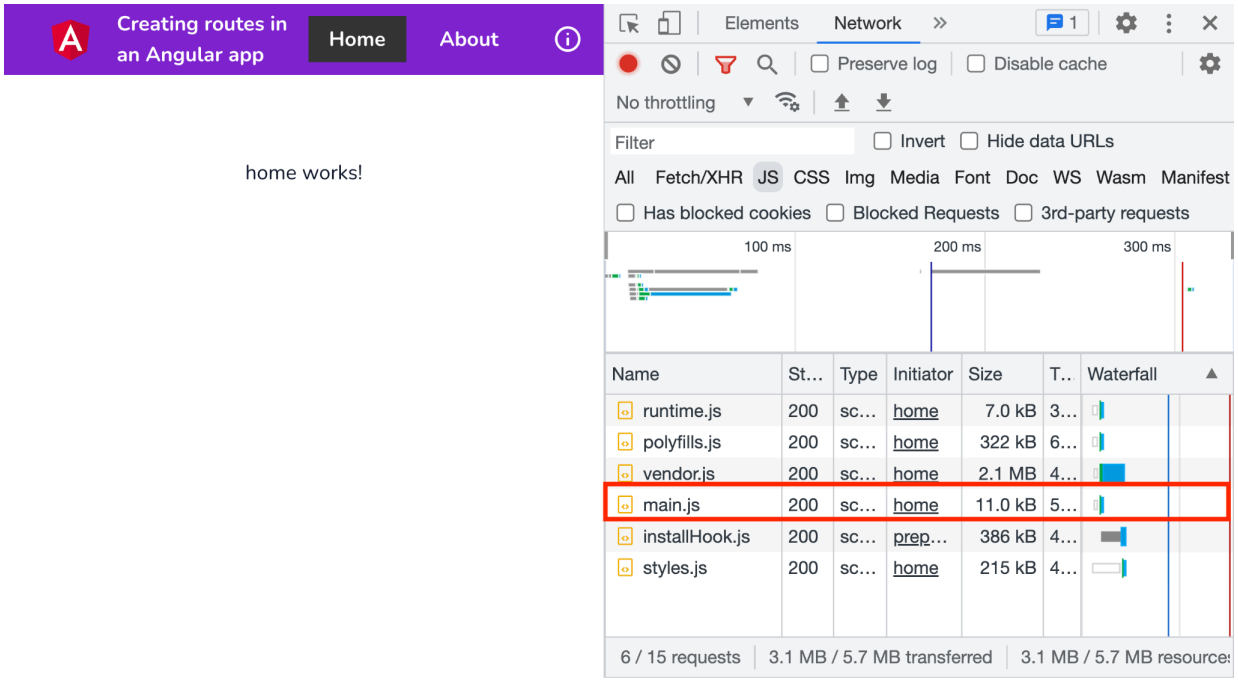


Figure 7.4 – ng-lazy-routing app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

As shown in *Figure 7.4*, we have all the components and modules inside the `main.js` file. Therefore, we have about 11.0 KB in size (this could change based on how Angular further optimizes the framework) for the `main.js` file. We'll modify the code and the routing structure to achieve lazy loading. As a result, we'll have the particular files of the routes loading when we actually navigate to them. Before the standalone components era, this step was pretty difficult and if you even have NgModules now, you'd face it. But since our application has only Standalone Components, see how easy it becomes to change it.:

1. Update the `app.routes.ts` to use the `loadComponent` method instead of the `component` property in the routes as follows

```
import { Route } from '@angular/router';
export const appRoutes: Route[] = [{
  path: '',
  pathMatch: 'full',
  redirectTo: 'home'
}, {
```

```
  path: 'home',
  loadComponent: () => import('./home/home.component').then(m => m.HomeComponent)
}, {
  path: 'about',
  loadComponent: () => import('./about/about.component').then(m => m.AboutComponent)
}];
```

1. Refresh the app and you'll see that the bundle size for the `main.js` file is down to 7.5 KB, which was about 11.0 KB before. See the following screenshot:

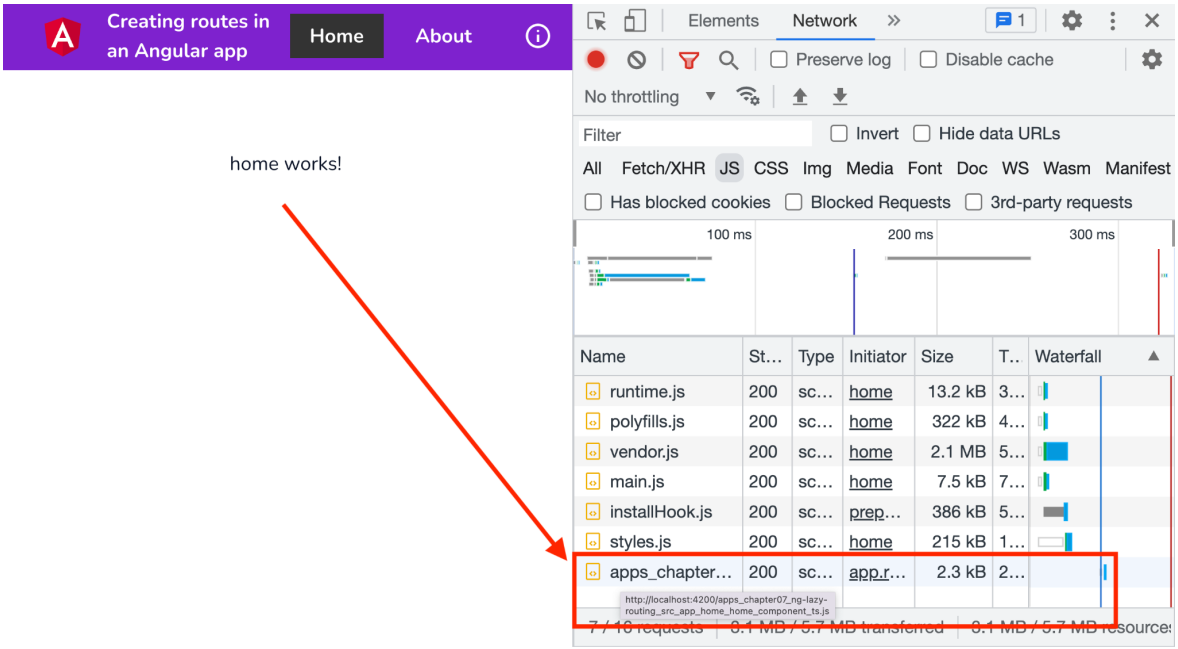


Figure 7.5 – Reduced size of main.js on app load

2. But what about the Home and About routes? And what about lazy loading? Well, tap the `About` route from the header and you'll see a new JavaScript file being downloaded in the **Network** tab specifically for the route. That's lazy loading in action! See the following screenshot:

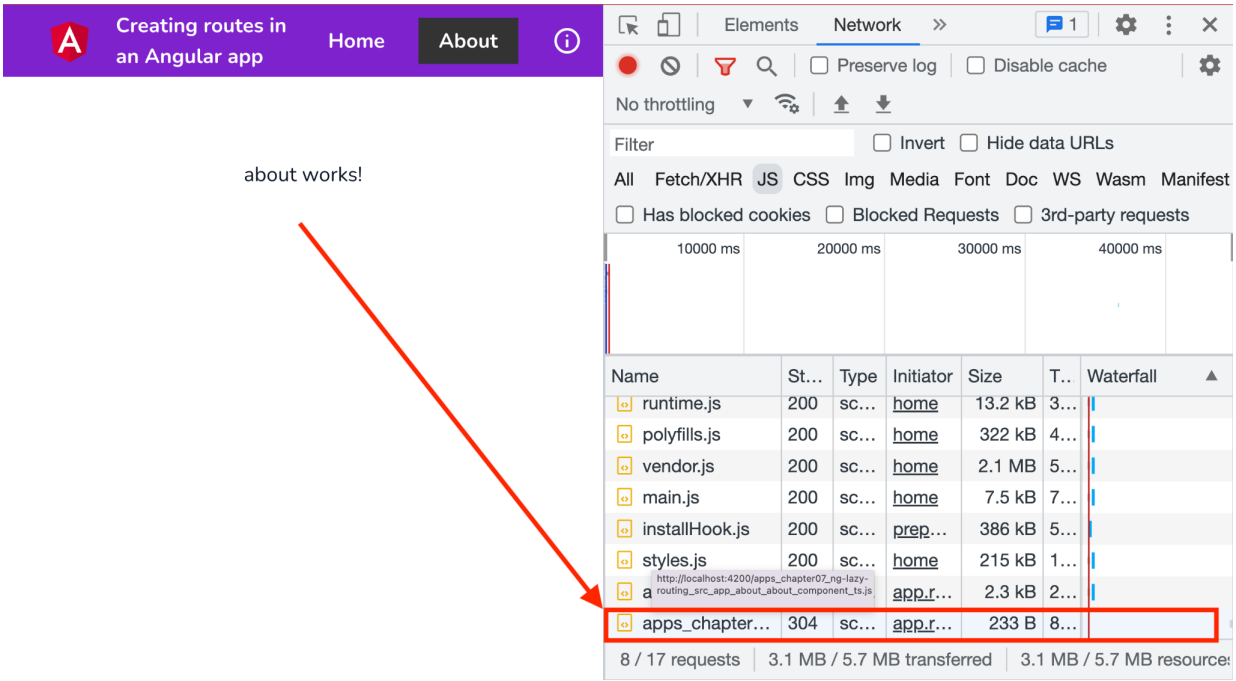


Figure 7.6 – about route being lazily loaded

Awesomesauce! You just became lazy! Just kidding. You just learned the art of lazily loading routes and feature components in your Angular app. You can now show this off to your friends as well.

How it works...

Angular works with modules and components and usually the features are broken down into either NgModules or now with Standalone Components, the components themselves are lazily loaded. We're going to look into the Standalone Components perspective. As we know, `AppComponent` serves as the entry point for the Standalone Angular app and Angular will import and bundle anything that is imported in `AppComponent` during the build process, resulting in the `main.js` file. However, if we want to lazy load our routes/feature components, we need to avoid importing feature components in `AppComponent` directly, or importing them in the Routes even. Rather than that, we can use the `loadChildren` method for lazily loading other modules, or `loadComponent` method to lazily load other Standalone Components. That's what we did in this recipe. It is important to note that the routes stayed the same in `app.routes.ts` file.

See also

Lazy loading modules in Angular (https://angular.io/guide/lazy-loading-ngmodules)Lazy loading Standalone Components (https://angular.io/guide/standalone-components#routing-and-lazy-loading)

Authorized access to routes using route guards

Not all routes in your Angular app should be accessible by everyone in the world. In this recipe, we'll learn how to create route guards in Angular to prevent unauthorized access to routes.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-route-guards` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-route-guards` to serve the project

This should open the app in a new browser tab. And you should see the following:

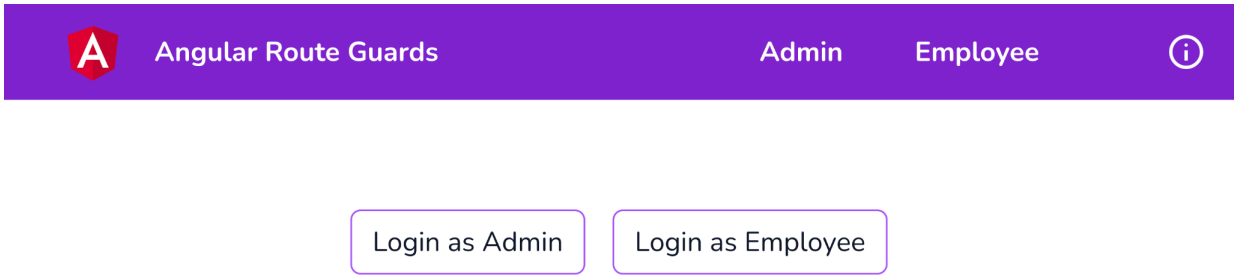


Figure 7.7 – ng-route-guards app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an app with a couple of routes already set up. You can log in as either an employee or an admin to get to the bucket list of the app. However, if you tap any of the two buttons in the header, you'll see that you can navigate to the Admin and Employee sections even without being logged in. This is what we want to prevent from happening. Notice in the `auth.service.ts` file that we already have a way for the user to do a login, and we can check whether the user is logged in or not, using the `isLoggedIn()` method:

1. First, let's create a route guard that will only allow the user to go to the particular routes if the user is logged in. We'll name it `AuthGuard`. Let's create a new file inside the auth folder and name it `auth.guards.ts`. The add the following code:

```
import { inject } from "@angular/core";
import { CanActivateFn, Router } from "@angular/router";
import { AuthService } from "./auth.service";
export const canActivateAdminOrEmployee: CanActivateFn = () => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const isLoggedIn = authService.isLoggedIn();
  if (!isLoggedIn) {
    router.navigate(['/auth']);
    return false;
  }
  return true;
};
```

1. Now we can add this guard to the routes. Update the app.routes.ts as follows:

```
      import { Route } from '@angular/router';
import { canActivateAdminOrEmployee } from './auth/auth.guards';
export const appRoutes: Route[] = [
  {...},
  { path: 'auth', ... },
  {
```

```
    path: 'admin',
    ...,
    canActivate: [canActivateAdminOrEmployee]
  },
  {
    path: 'employee',
    ...,
    canActivate: [canActivateAdminOrEmployee]
  }
];
```

If you try to go to the Admin or Employee pages without being logged in now, you'll see you are not able to do that anymore. Thanks to the route guard.

1. Let's make sure we can't go to the auth page if we're already logged in. Add a function guard in the `auth.guards.ts` file as follows:

```
...
import { UserType } from '../constants/user-type';
export const canActivateLogin: CanActivateFn = () => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const isLoggedIn = authService.isLoggedIn();
  if (router.url === '/' && isLoggedIn) {
    const isAdmin = authService.loggedInUserType === UserType.Admin;
    router.navigate([`/${isAdmin ? 'admin' : 'employee'}`])
    return false
  }
  return !inject(AuthService).isLoggedIn()
}
export const canActivateAdminOrEmployee: CanActivateFn = () => {...};
```

If you log in, and then click the browser back button, you'll see you can't go back to the `/auth` page as well. Even if you open a new tab and go to http://localhost:4200, you'll see that it takes you to the correct page based on the logged-in user's type.Great! You now are an authorization expert when it comes to guarding routes. With great power comes great responsibility. Use it wisely.

How it works...

Angular has switched to functional route guards since Angular v14.2. This makes it much easier to configure the routes compared to the previous version. There are many route guards like `CanActivate`, `CanDeactivate`, `CanActivateChildren` etc. The `provideRouter` method used in the main.ts makes it possible for us to provide the routes with functional route guards. A route can take the guards as an array against the guard property name. You can see how we provided our `canActivateAdminOrEmployee` guard against both the '`/employee`' and '`/admin`' routes in Step 2. A functional guard is supposed to return a `Boolean` value or a `UrlTree`, a `Promise` of them, or an `Observable` of them. We've focused on the Boolean value's usage in our recipe. In the `canActivateAdminOrEmployee` guard, we check if the user is logged in. If that's true, we allow the route to be activated (the routing to happen), or we navigate to the '`/auth`' route. In the `canActivateLogin` guard, we do something a bit more complex. Since someone could land on the homepage (the route is '`/`' which redirects to '`/auth`'), we have to first check if the user is logged in, and if that's the case, then which type of user this is. Based on evaluation, we route the user to either '`/employee`' or '`/admin`' route.

See also

Preventing unauthorized access in Angular routes (https://angular.io/guide/router#preventing-unauthorized-access)Angular CanActivateFn docs (https://angular.io/api/router/CanActivateFn)

Working with route parameters

Whether it is about building a REST API using Node.js or configuring routes in Angular, setting up routes is an absolute art, especially when it comes to working with parameters. In this recipe, you'll create some routes with parameters and will learn how to get those parameters in your components once the route is active.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-route-guards` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-route-guards` to serve the project

This should open the app in a new browser tab. And you should see the following:

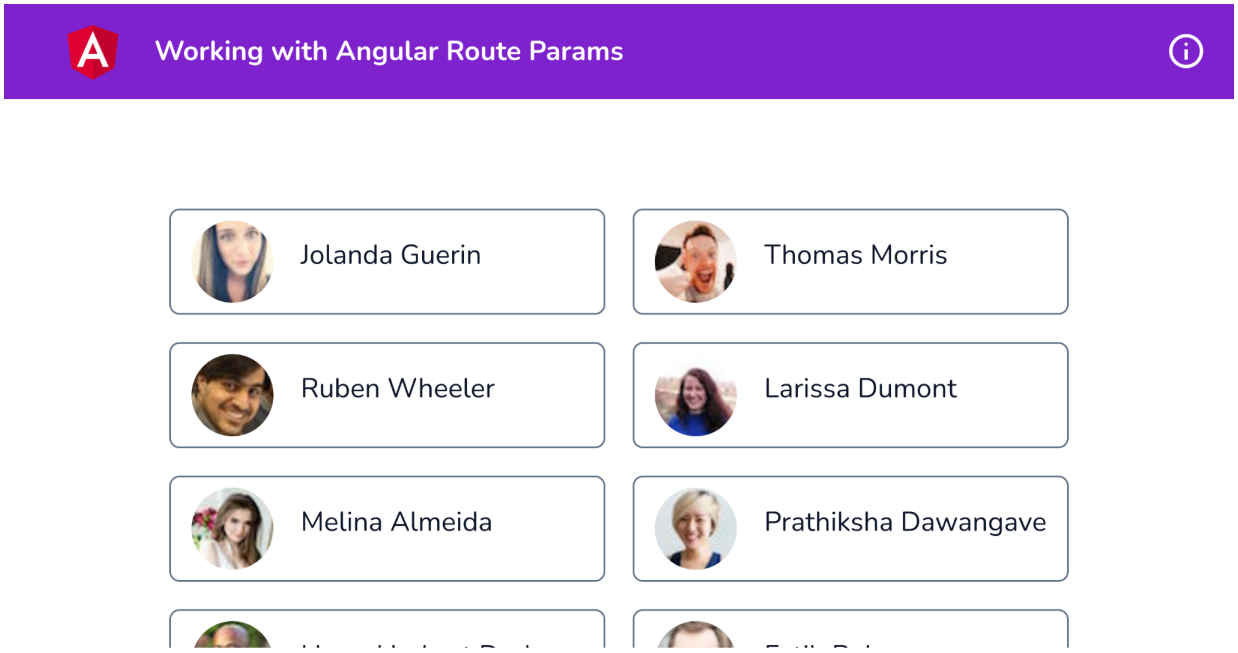


Figure 7.8 – ng-route-params app running on localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

The problem right now is that we have a route for opening the user details, but we don't see anything. I.e. we have a blank page. That's because this route doesn't have any idea about which user to show. Wouldn't it be nice to have a way to pass the clicked user's information from the users list page to the user detail page? That's exactly what we're going to do in this recipe.

1. First, we have to make our user route capable of accepting the route parameter. This is going to be a `required` parameter, which means the route will not work without passing this. Let's modify `app-routes.ts` to add this required parameter to the route definition, as follows:

```
import { Route } from '@angular/router';
export const appRoutes: Route[] = [
  { path: '', ...},
  { path: 'users', ...},
  {
```

```
    path: 'users/:uuid',
    loadComponent: () => ...,
  },
];
```

1. We'll now update the `users-list.component.html` file to change the links to use the `uuid` as follows:

```
<ul>
  <li
    routerLink="/users/{{user.uuid}}"
    *ngFor="let user of users">
    ...
  </li>
</ul>
```

We're now able to navigate to a particular user's route using the uuid, and you should also be able to see it in the address bar as follows:

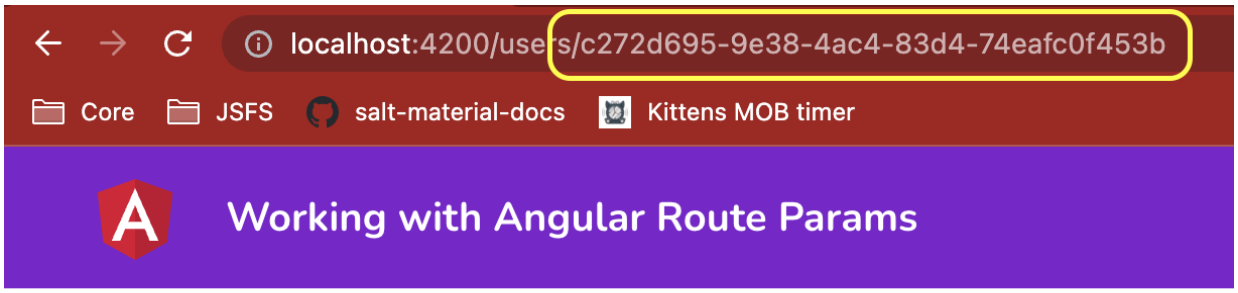


Figure 7.8 – The UUID being shown in the address bar

1. To get the user from `UserService` based on the `uuid`, we need to get the `uuid` value in `UserDetailsComponent` from the route params. Let's update the `user-details.component.ts` as follows:

```
...
import { ActivatedRoute, RouterModule } from '@angular/router';
import { filter, map, Observable} from 'rxjs';
...
@Component({...})
export class UserDetailsComponent {
  route = inject(ActivatedRoute);
  ...
  constructor() {
    this.user$ = this.route.paramMap.pipe(
      map((params) => params.get('uuid')),
      filter((uuid) => !!uuid),
      map((uuid) => {
        return this.userService.getById(uuid as string) || null;
      })
    );
  }
}
```

You should be able to see the user we clicked from the `'/users'` page now. The only thing remaining is showing the similar users.

1. Instead of creating another subscription, we can tap into the one we created in the previous step using the `tap()` operator from `RxJS`. Update the `user-details.component.ts` file further as follows:

```
...
import { filter, map, Observable, of, tap} from 'rxjs';
...
@Component({...})
export class UserDetailsComponent {
  ...
  constructor() {
    this.user$ = this.route.paramMap.pipe(
      map((params) => params.get('uuid')),
      filter((uuid) => !!uuid),
      tap((uuid) => {
        this.similarUsers$ = of(this.userService.getSimilar(uuid as string))
      }),
      map((uuid) => {...})
    );
  }
}
```

And boom, you should be able to see both the clicked user and similar users as follows:

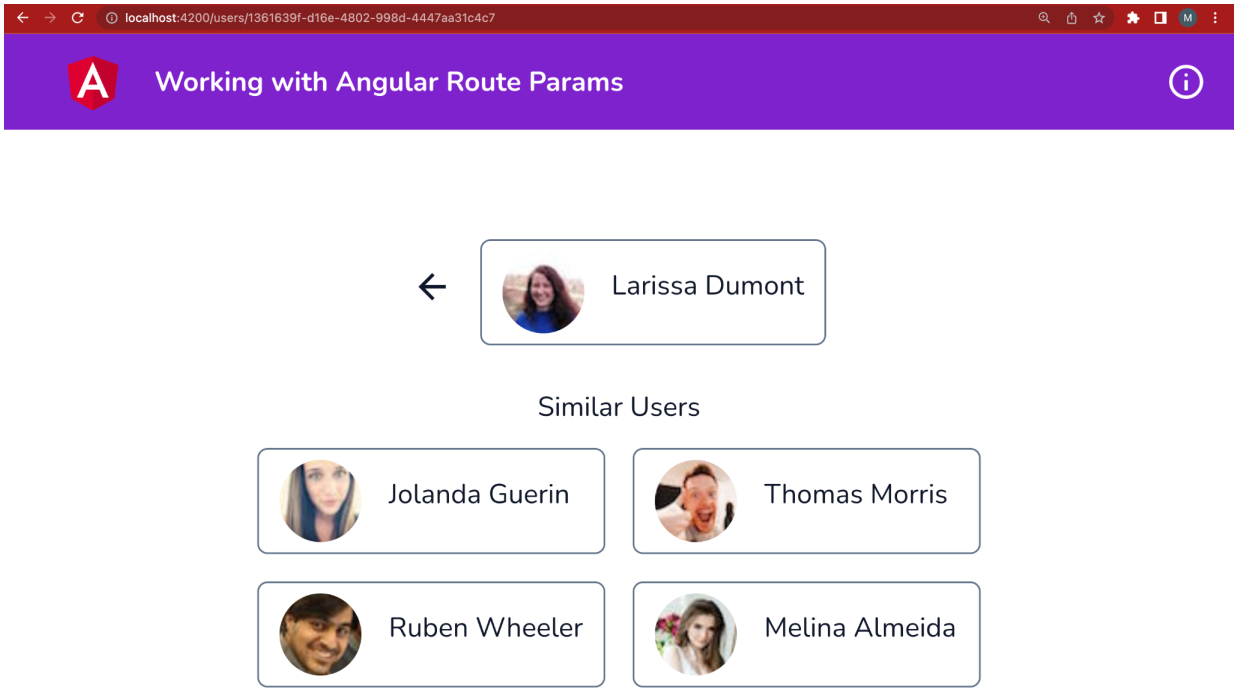


Figure 7.9 – The UUID being shown in the address bar

Grrreat!! With this change, you can try refreshing the app on the home page and then click any user. You should see the current user as well as similar users being loaded. To understand all the magic behind the recipe, see the next section.

How it works...

It all begins when we change our route's path to `user/:userId`. This makes `userId` a parameter for our route. The other piece of the puzzle is to retrieve this parameter in `UserDetailsComponent` and then use it to get the target user, as well as similar users. For that, we use the `ActivatedRoute` service. The `ActivatedRoute` service holds a lot of necessary information about the current route and, therefore, we were able to fetch the current route's `uuid` parameter by subscribing to the `paramMap` Observable, so even if the parameter changes while staying on a user's page, we still execute the necessary operations. Notice that we also use the `tap()` method to do something extra. That is assigning the value of similar

users to the desired observable. Since both of the observables are only used in the template with the `async` pipe, there is no need to unsubscribe them ourselves as Angular takes care of it..

See also

Accessing query parameters and fragments (https://angular.io/guide/router#accessing-query-parameters-and-fragments)Getting route information - Angular Docs (https://angular.io/guide/router#getting-route-information)

Showing a global loader between route changes

Building user interfaces that are snappy and fast is key to winning users. The apps become much more enjoyable for the end users and it could bring a lot of value to the owners/creators of the apps. One of the core experiences on the modern web is to show a loader when something is happening in the background. In this recipe, you'll learn how to create a global user interface loader in your Angular app that shows whenever there is a route transition in the app.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-global-loader` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-global-loader` to serve the project

This should open the app in a new browser tab. And you should see the following:

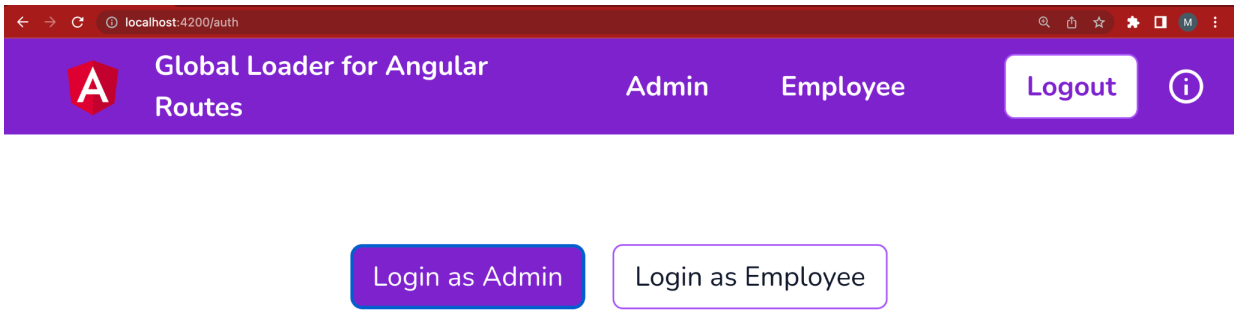


Figure 7.10 – ng-global-loader app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

If you try logging in, or logging out, you'll see that it takes a while before the route changes. We're simulating a delay in route change manually here which could replicate use cases where you need to route based on an HTTP call. In this recipe, we have the `LoaderComponent` already created, which we have to use during the route changes:

1. We'll begin by adding a new property in the `app.component.ts` that says isRouting. We'll initialize it as true for now. We'll also import the `LoaderComponent` class in the `AppComponent` class's `imports` so we can use it in the template. Update the file as follows.:

```
…
@Component({
  ...
  imports: [RouterModule, CommonModule, LoaderComponent],
  providers: [...],
})
export class AppComponent {
  ...
  router = inject(Router);
  isRouting = true;
  ...
}
```

1. Now we'll add the loader in the `app.component.html` to show it conditionally, and we'll wrap the entire app content in another variable. Update the file as follows:

```
<div *ngIf="isRouting; else app" class="loader-container fixed w-full h-full flex items-center ju
  <app-loader></app-loader>
</div>
<ng-template #app>
  <div class="toolbar" role="banner">...</div>
  <main class="content" role="main">...</main>
</ng-template>
```

You should be able to see the loader consistently on the screen now, on every route. However we want to make it work with the Angular Router.

1. We'll now update the `app.component.ts` file to listen to the router service's `events` property, and take an action upon the `NavigationStart` event. Modify the code in the `app.component.ts` file as follows:

```
import { NavigationStart, Router, RouterModule } from '@angular/router';
...
@Component({...})
export class AppComponent {
  constructor() {
    this.router.events.subscribe((event) => {
      if (event instanceof NavigationStart) {
        this.isRouting = true;
      }
    })
  }
  get isLoggedIn() {...}
}
```

If you refresh the app, you'll notice that `<app-loader>` still never goes away. That's because we're not marking the `isRouting` property as `false` anywhere.

1. To mark `isRouting` as `false`, we need to check for three different events: `NavigationEnd`, `NavigationError`, and `NavigationCancel`. Let's add some more logic to handle these three events and mark the property as `false`:

```
...
import { NavigationCancel, NavigationEnd, NavigationError, NavigationStart, Router, RouterModule
...
@Component({...})
export class AppComponent {
  ...
  isRouting = false;
  constructor() {
    this.router.events.subscribe((event) => {
      if (event instanceof NavigationStart) {
        this.isRouting = true;
```

```
      } else if (
        event instanceof NavigationEnd ||
        event instanceof NavigationError ||
        event instanceof NavigationCancel
      ) {
        this.isRouting = false;
      }
    })
  }
  ...
}
```

And boom! We now have a global loader that shows during the route navigation among different pages.Congrats on finishing the recipe. You now can implement a global loader in Angular apps, which will show from the navigation start to the navigation end.

How it works...

The router service is a very powerful service in Angular. It has a lot of methods as well as Observables that we can use for different tasks in our apps. For this recipe, we used the `events` Observable. By subscribing to the `events` Observable, we can listen to all the events that the `Router` service emits through the Observable. For this recipe, we were only interested in the `NavigationStart`, `NavigationEnd`, `NavigationError`, and `NavigationCancel` events. The `NavigationStart` event is emitted when the router starts navigation. The `NavigationEnd` event is emitted when the navigation ends successfully. The `NavigationCancel` event is emitted when the navigation is canceled due to a `route guard` returning `false`, or redirects by using `UrlTree` due to some reason. The `NavigationError` event is emitted when there's an error due to any reason during the navigation. All of these events are of the `Event` type and we can identify the type of the event by checking whether it is an instance of the target event, using the `instanceof` keyword. Notice that since we had the subscription to the `Router.events` property in `AppComponent`, we didn't have to worry about unsubscribing the subscription because there's only one subscription in the app, and `AppComponent` will not be destroyed throughout the life cycle of the app.

See also

Router events docs (https://angular.io/guide/router#router-events)Router service docs (https://angular.io/api/router/Router)

Preloading route strategies

We're already familiar with how to lazy load different feature modules upon navigation. Although sometimes, you might want to preload subsequent routes to make the next route navigation instantaneous or might even want to use a custom preloading strategy based on your application's business logic. In this recipe, you'll learn about the `PreloadAllModules` strategy and will also implement a custom strategy to cherry-pick which modules should be preloaded.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-route-preload-strat` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-route-preload-strat` to serve the project

This should open the app in a new browser tab. If you log in as an admin, you should see the following:

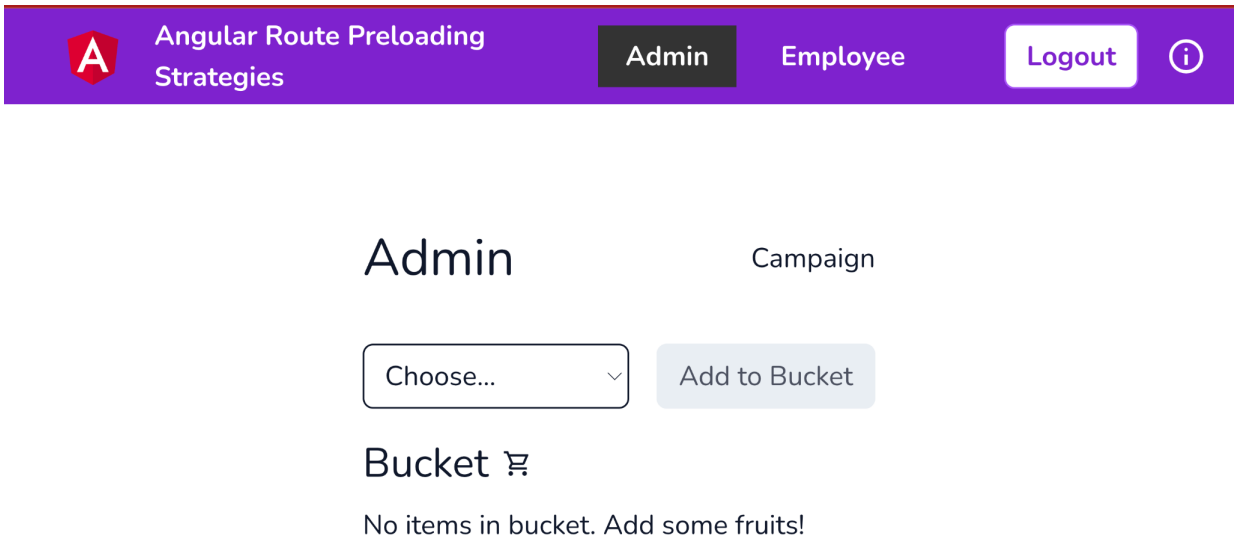


Figure 7.11 – ng-route-preload-strat app running on http://localhost:4200

1. Open Chrome DevTools by pressing *Ctrl* + *Shift* + *C* on Windows or *Cmd* + *Shift* + *C* on Mac.

Navigate to the **Network** tab and filter on JavaScript files only. You should see something like this:

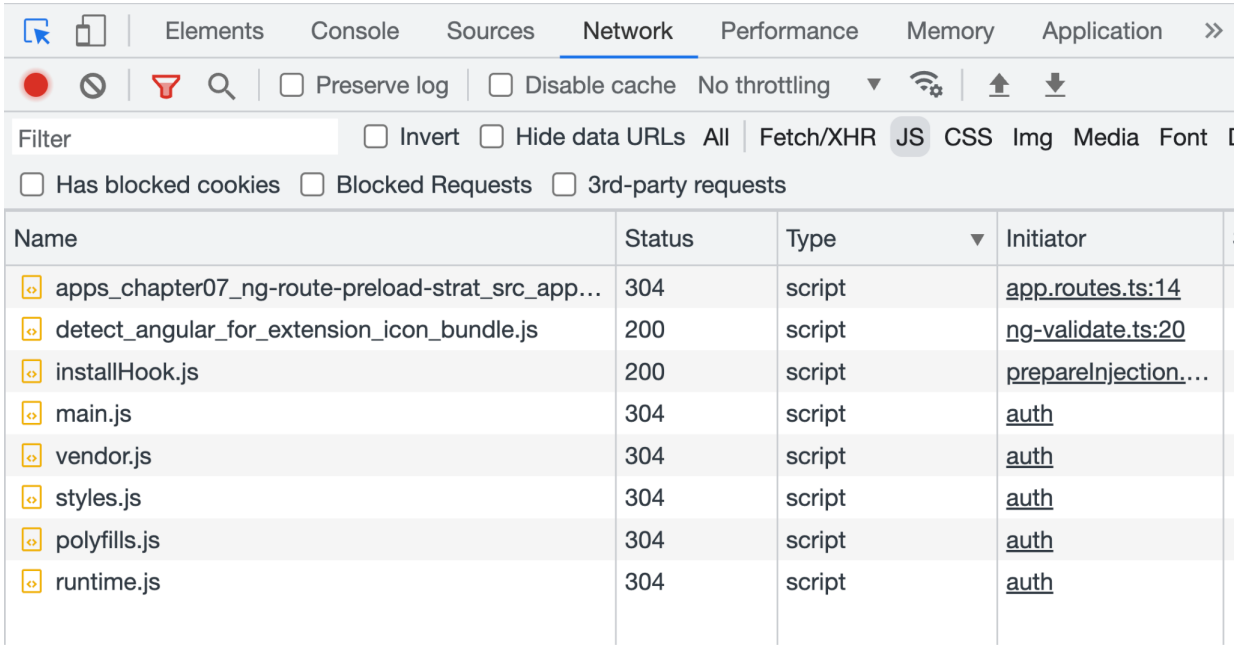


Figure 7.11 – JavaScript files loaded on app load

Now that we have the app running locally, let's see the next section for this recipe

How to do it...

Notice in *Figure 7.11* the first network call. It is the javascript bundle that contains the admin component and the bucket component. Although the routes in `app-routes.ts` are all configured to be

lazily loaded, we can still look into what happens if we use the `PreloadAllModules` strategy, and then a custom preload strategy:

1. We're going to try out the `PreloadAllModules` strategy first. To use it, let's modify the `main.ts` file as follows:

```
import { bootstrapApplication } from '@angular/platform-browser';
import {
  PreloadAllModules,
  provideRouter,
  withEnabledBlockingInitialNavigation,
  withPreloading,
} from '@angular/router';
...
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation(), withPreloading(PreloadAllMo
    provideAnimations()
  ],
}).catch((err) => console.error(err));
```

If you refresh the app, you should all the JavaScript bundles being download for all components as follows:

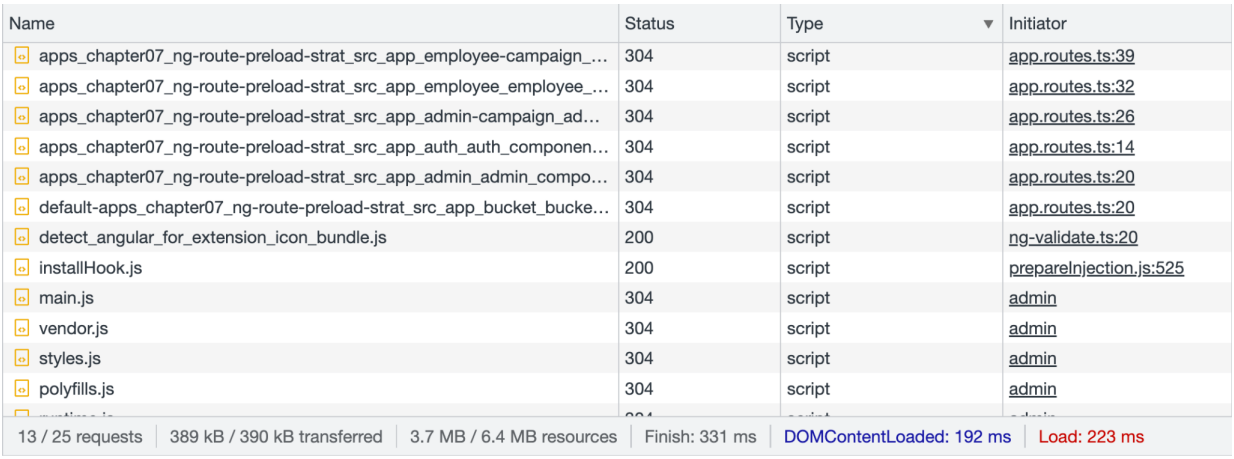


Figure 7.12 – All components being loaded eagerly using PreloadAllModules strategy

So far so good. But what if we wanted to preload only the Admin module, supposing our app is intended for admins mostly? We'll create a custom preload strategy for that.

1. Let's create a service named `CustomPreloadStrategy` by running the following command from within the `start` folder:

```
npx nx g service app-preload-strategy --project ng-route-preload-strat
```

1. In order to use our preload strategy service with Angular, our service needs to implement the `PreloadingStrategy` interface from the `@angular/router` package. Modify the newly created service as follows:

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy } from '@angular/router';
@Injectable({
  providedIn: 'root'
})
export class CustomPreloadStrategyService implements PreloadingStrategy {
}
```

1. Next, we need to implement the `preload` method from the `PreloadingStrategy` interface for our service to work properly. Let's modify `CustomPreloadStrategyService` to implement the `preload` method, as follows:

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';
@Injectable({...})
export class CustomPreloadStrategyService implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>):   Observable<any> {
    return of(null)
  }
  logAndLoad(route: Route, load: () => Observable<any>) {
    console.log(`Preloading route: ${route.path}`);
    return load();
  }
}
```

1. Right now, our `preload` method returns `of(null)` . Instead, in order to decide which routes to preload, we're going to add an object to our route definitions as the `data` object having a Boolean value for admin routes and employee routes to preload. Let's modify the `app-routes.ts` as follows:

```
...
export const appRoutes: Route[] = [
  { path: '', ...},
  { path: 'auth', ...},
  { path: 'admin', ..., data: { loadForAdmin: true } },
  {
    path: 'admin-campaign',
    ...,
    data: { loadForAdmin: true }
  },
  {
    path: 'employee',
    ...,
    data: { loadForEmployee: true },
  },
  {
    path: 'employee-campaign',
    ...,
    data: { loadForEmployee: true },
  },
];
```

1. Now let's add the logic to our `preload()` method in the `AppPreloadStrategyService` class to work with the properties added in Step 5. We're going to first inject the `AuthService` and will create some variables inside the `preload()` method. Update the `app-preload-strategy.service.ts` file as follows:

```
...
import { AuthService } from './auth/auth.service';
import { UserType } from './constants/user-type';
...
export class AppPreloadStrategyService implements PreloadingStrategy {
  auth = inject(AuthService);
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    const isLoggedIn = this.auth.isLoggedIn();
    if (!isLoggedIn) {
      return of(null)
    }
    const isAdmin = this.auth.loggedInUserType === UserType.Admin;
    return of(null)
  }
  logAndLoad(route: Route, load: () => Observable<any>) {...}
}
```

1. Let's modify the preload() method further to work with the `loadForAdmin` and `loadForEmployee` route-data properties as follows:

```
...
export class CustomPreloadStrategyService implements PreloadingStrategy {
...
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    ...
    const isAdmin = this.auth.loggedInUserType === UserType.Admin;
    if (isAdmin && route.data?.['loadForAdmin']) {
      return this.logAndLoad(route, load);
    } else if (!isAdmin && route.data?.['loadForEmployee']) {
      return this.logAndLoad(route, load);
    }
    return of(null)
  }
}
```

1. The final step is to use our custom preload strategy. In order to do so, modify the `main.ts` file as follows:

```
import { bootstrapApplication } from '@angular/platform-browser';
import {
  PreloadAllModules,
  provideRouter,
  withEnabledBlockingInitialNavigation,
  withPreloading,
} from '@angular/router';
...
import { AppPreloadStrategyService } from './app/app-preload-strategy.service';
bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation(), withPreloading(AppPreloadSt
    provideAnimations()
  ],
}).catch((err) => console.error(err));
```

1. Voilà! If you refresh the app now and monitor the **Network** tab, you'll notice when you're logged in as Admin, you can see the console log that admin-campaign route is preloaded as shown in *Figure 7.13* as follows:

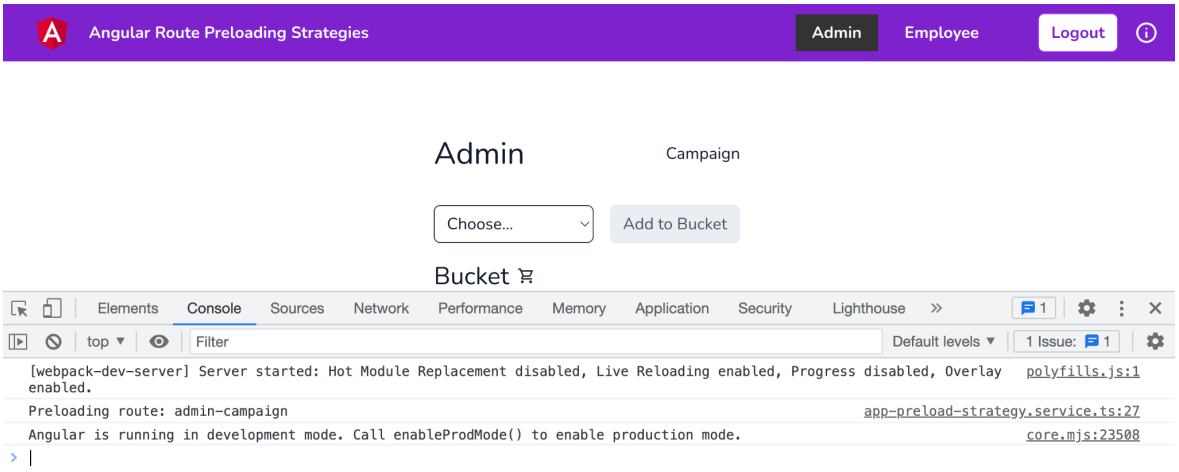


Figure 7.13 – Preloading admin-campaign route when landing on admin route using the custom preload strategy

2. You can also have a look at the network tab to see the bundles being downloaded as follows:

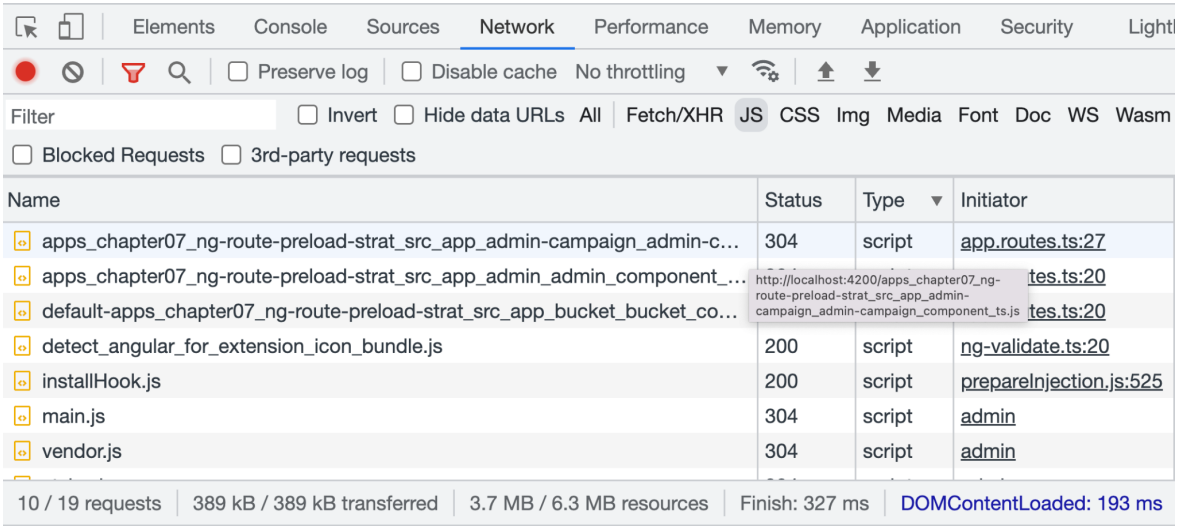


Figure 7.14 – JavaScript bundle for admin-campaign route preloaded

3. Try logging out and logging in as the employee, you'll see the `employee-campaign` route being preloaded. And not only that, if you refresh on the `employee-campaign` route, you'll see the `employee` route being preloaded as well. The same goes with the `admin-campaign` and `admin` routes.

Now that you've finished the recipe, see the next section on how this works.

How it works...

Angular provides a great way to implement our own custom preloading strategy for our feature modules. We can decide easily which modules should be preloaded and which should not. In the recipe, we learned a very simple way to configure the preloading using the `data` object of the routes configuration by adding properties named `loadForAdmin` and `loadForEmployee`. We created our own custom preload strategy service named `AppPreloadStrategyService`, which implements the `PreloadingStrategy` interface from the `@angular/router` package. The idea is to use the `preload` method from the `PreloadingStrategy` interface, which allows us to decide whether a route should be preloaded. That's because Angular goes through each route using our custom preload strategy and let's us decide which routes to preload. And that's it. We can see if the logged in user is Admin and has the `loadForAdmin` property in the route's data, or if the logged in user is an Employee and has the `loadForAdmin` property in the route's data. We then preload the route. Otherwise we don't. And there's no preloading when the user is not logged in.

See also

Route preloading strategies article on `web.dev` (https://web.dev/route-preloading-in-angular/)Route preloading strategies in Angular (https://www.youtube.com/watch?v=RQGLcMnh9k8)

8 Mastering Angular Forms

Getting user inputs is an integral part of almost any modern app that we use. Whether it is authenticating users, asking for feedback, or filling out business-critical forms, knowing how to implement and present forms to end users is always an interesting challenge. In this chapter, you'll learn about Angular forms and how you can create great user experiences using them.Here are the recipes that we're going to cover in this chapter:

- Creating your first Template-Driven form with validation
- Creating your first Reactive Form with form validation
- Testing forms in Angular
- Server side validation using asynchronous validator functions
- Implementing Complex Forms with Reactive Form Arrays
- Writing your own custom form control using `ControlValueAccessor`

Technical requirements

For the recipes in this chapter, make sure you have **Git** and **NodeJS** installed on your machine. You also need to have the `@angular/cli` package installed, which you can do with `npm install -g @angular/cli` from your terminal. The code for this chapter can be found at https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter08.

Creating your first Template-Driven form with validation

Let's start getting familiar with Angular forms in this recipe. In this one, you'll learn about the basic concepts of template-driven forms and will create a basic Angular form using the template-driven forms API.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-tdf` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-tdf` to serve the project

This should open the app in a new browser tab. And you should see the following:

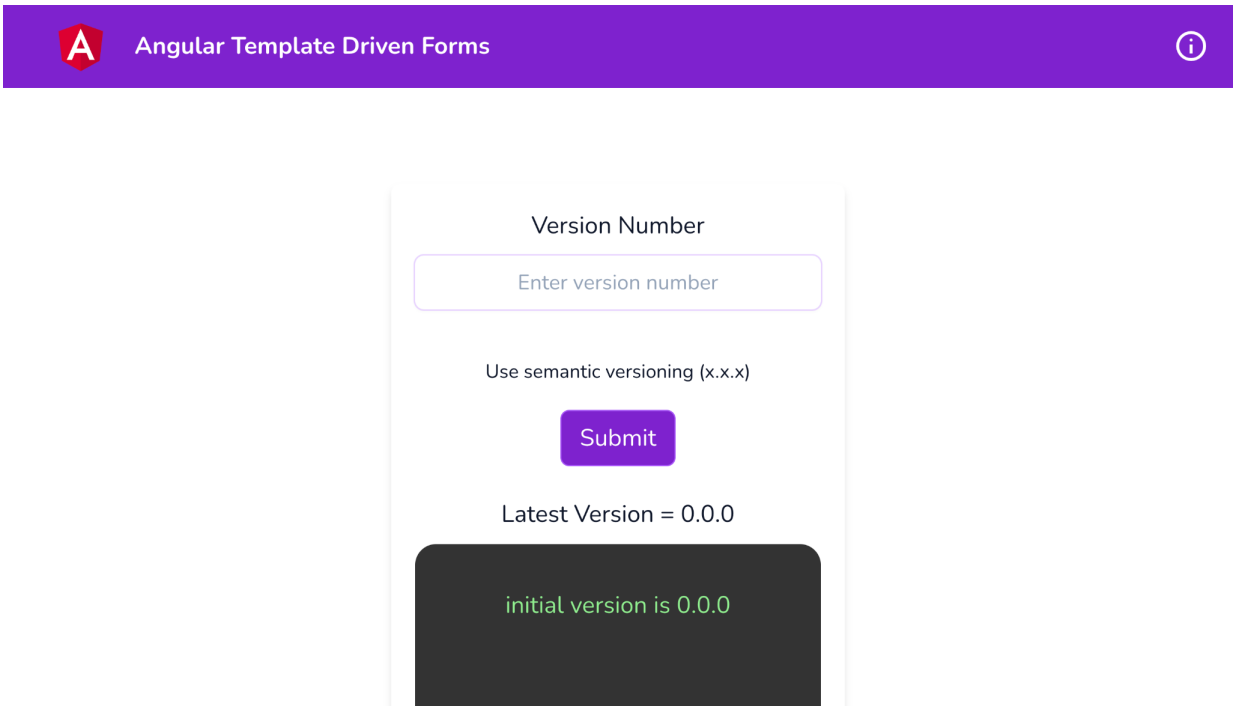


Figure 8.1 – ng-tdf app running on http://localhost:4200

How to do it...

We have an Angular app that already has a bunch of components and some UI setup. need to implement the functionality to add a new version to the logs. We'll use a template-driven form to allow the user to pick an app and submit a release version. Let's get started:

1. First, we'll add the `FormsModule` in the `src/app/components/version-control/version-control.component.ts` file in the project as follows:

```
...
import { FormsModule } from '@angular/forms';
...
@Component({
  ...,
  imports: [CommonModule, FormsModule, VcLogsComponent]
})
export class VersionControlComponent  {...}
```

1. Now we'll use `NgModel` and `NgForm` in the template file to use template driven forms. Update the `version-control.component.html` file as follows:

```
<form #versionForm="ngForm">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input [(ngModel)]="versionInput" #version="ngModel" name="version" type="text" class="form-
    ...
  </div>
  ...
</form>
```

1. Let's create a function which will be triggered when the form is submitted. Update the `version-control.component.ts` file as follows:

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

```
...
export class VersionControlComponent  {
  ...
  formSubmit(form: NgForm) {
    this.versionName = form.controls['version'].value;
  }
}
```

1. Now that we have the form submit handler, let's add the event listener in the template. Update the `version-control.component.html` file as follows:

```
<form #versionForm="ngForm" (ngSubmit)="formSubmit(versionForm)">
  ...
</form>
<app-vc-logs [vName]="versionName"></app-vc-logs>
```

After adding this, you should be able to add new versions. However, you can use any value for the input or even provide nothing and it will add as a new version. See the following screenshot for example:

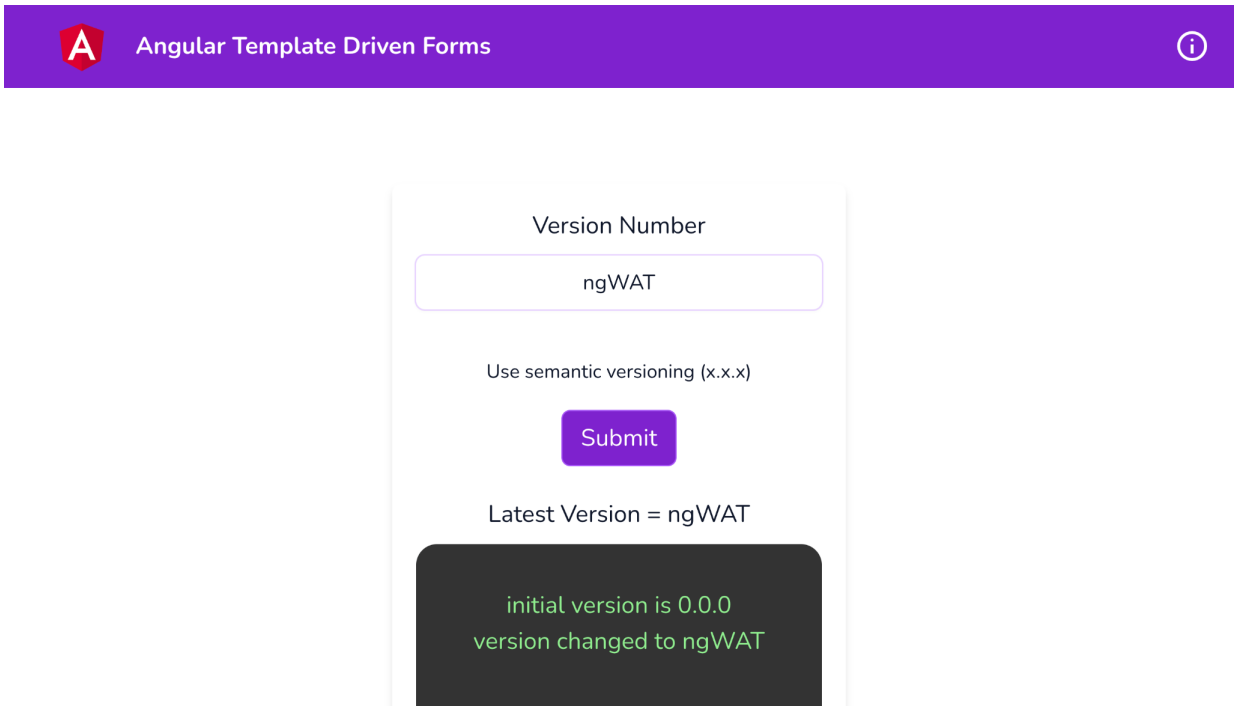


Figure 8.2 – ng-tdf app without form validation

We'll add some form validation from now on. First of all, we'll make sure we only change the value of the `versionName` property when the form has a valid input. Update the `formSubmit()` method in the `version-control.component.ts` . file as follows:

```
export class VersionControlComponent  {
  ...
  formSubmit(form: NgForm) {
    if (!form.valid) {
      return;
    }
    this.versionName = form.controls['version'].value;
  }
}
```

1. We now add some validations in the template file. We'll make the input required and will ensure the version provided follows the semantic versioning. Update the `version-control.component.html` file as follows:

```
<form #versionForm="ngForm" (ngSubmit)="formSubmit(versionForm)">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input [(ngModel)]="versionInput" pattern="([0-9]+)\.([0-9]+)\.([0-9]+)" required #version="n
    ...
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

If you try submitting the form now with an invalid value, you'll see that the version provided doesn't get added to the logs. But if you provide a valid version like 2.1.0, it will be added to the logs. However, this isn't great in terms of UX since it doesn't tell what's wrong.

1. Let's show some error messages based on the form validation we have. Update the `version-control.component.html` as follows:

```
<form #versionForm="ngForm" (ngSubmit)="formSubmit(versionForm)">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input [(ngModel)]="versionInput" pattern="([0-9]+)\.([0-9]+)\.([0-9]+)" required #version="n
    <small id="versionHelp" class="form-text text-muted block mt-6">Use semantic versioning (x.x
    <small class="error block" *ngIf="versionForm.submitted && versionForm.controls['version'].er
      Version number is required
    </small>
    <small class="error block" *ngIf="versionForm.submitted && versionForm.controls['version'].er
      Version number does not match the pattern (x.x.x)
    </small>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
<app-vc-logs [vName]="versionName"></app-vc-logs>
```

If you now submit the form without any input, you should see the error as follows:

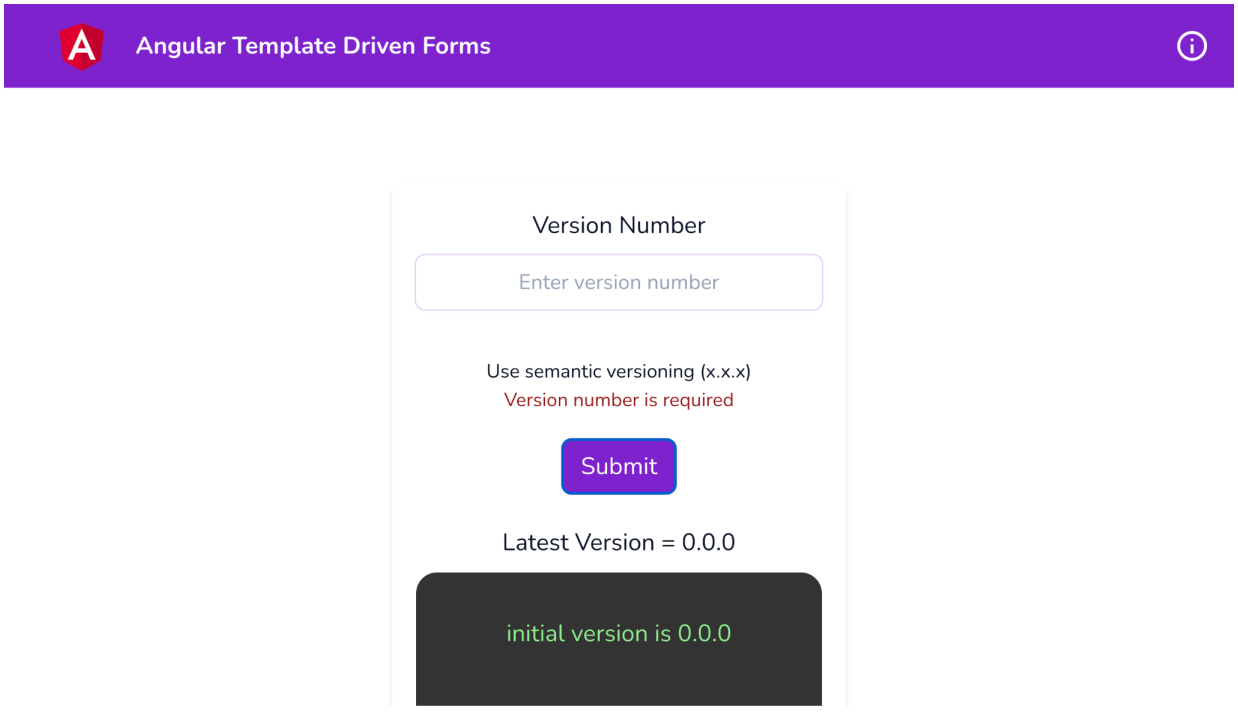


Figure 8.3 – ng-tdf app with form validation

And if you try giving it a wrong value, you'll see a different error.Great! Within a few minutes, we were able to create our first template-driven form in Angular with form validation. If you refresh the app now

and adding some versions, you should see it as follows:

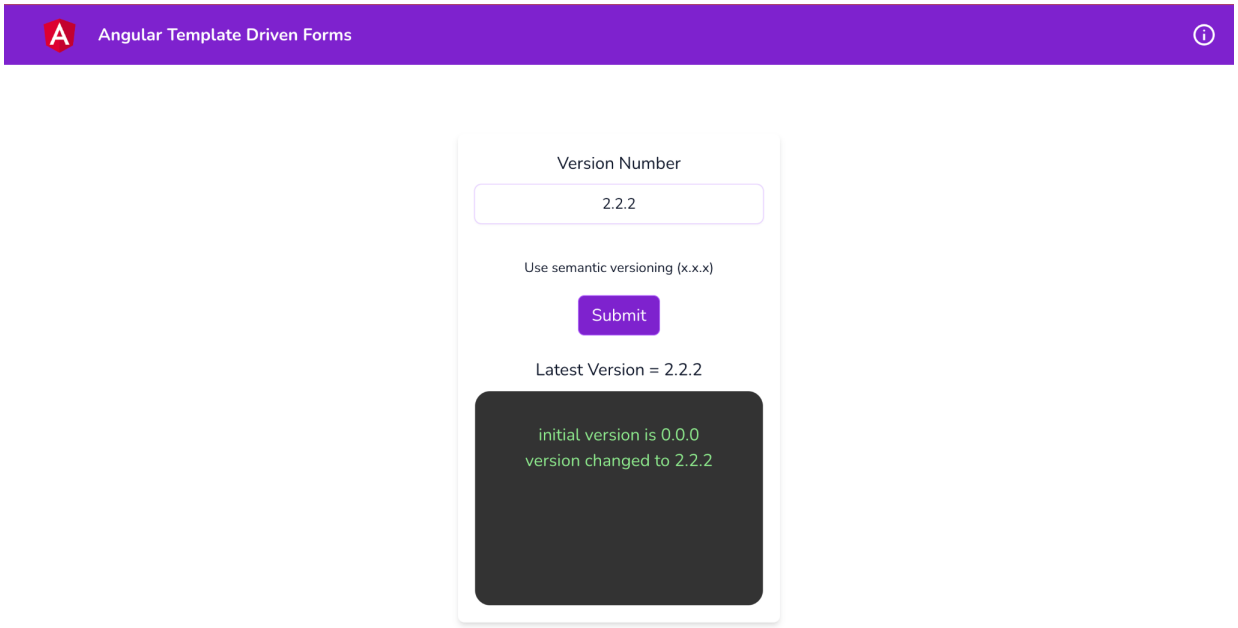


Figure 8.3 – ng-tdf app final output

Now that you know how the template-driven forms are created, let's see the next section to understand how it works.

How it works...

The key to using template-driven forms in Angular resides in `FormsModule`, the `ngForm` directive, and the `ngModel` binding. This would almost always include template variables along with the `name` attributes for inputs in the template. We began by adding the `FormsModule` in the `VersionControlComponent` class which is necessary for using the `ngForm` directive and the `[(ngModel)]` two-way data binding. Then we added the `[(ngModel)]` to the version input and add a template variable named `#version` to it. Notice that when we write `#version="ngModel"`, that means the `#version` template variable is now an `NgModel` instance through which can get the value of it, and can also see if the value in the control is valid. We also added the `ngForm` attribute on the form element similarly creating a `#versionForm` variable so we can use it to see if the entire form is valid. Then we added the `ngSubmit` handler to the `form` element so we can trigger a function when the form is submitted. We also added the respective function named `formSubmit()` in the `VersionControlComponent` class as well. Notice that we pass the `versionForm` variable to the `formSubmit` method, which makes it easier to test the functionality for us. Upon submitting the form, we use the form's value to create a `new version` log item. Notice that if you provide an invalid `version` for the new release log, the app will show the respective errors based on either the empty input, or an invalid version format. This is because we have `required` attribute set on the `<input />` element as well as the `pattern` attribute.

See also

Building a template-driven form in Angular: https://angular.io/guide/forms#building-a-template-driven-formShow and hide validation error messages (Angular Docs): https://angular.io/guide/forms#show-and-hide-validation-error-messages

Creating your first Reactive Form with validation

You've learned about template-driven forms in the previous recipe and are now confident in building Angular apps with them. Now guess what? Reactive forms are even better. Many known engineers and businesses in the Angular community recommend using Reactive forms. The reason is their ease of use when it comes to building complex forms. In this recipe, you'll build your first Reactive form and will learn its basic usage.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-reactive-forms` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-reactive-forms` to serve the project

This should open the app in a new browser tab. And you should see the following:

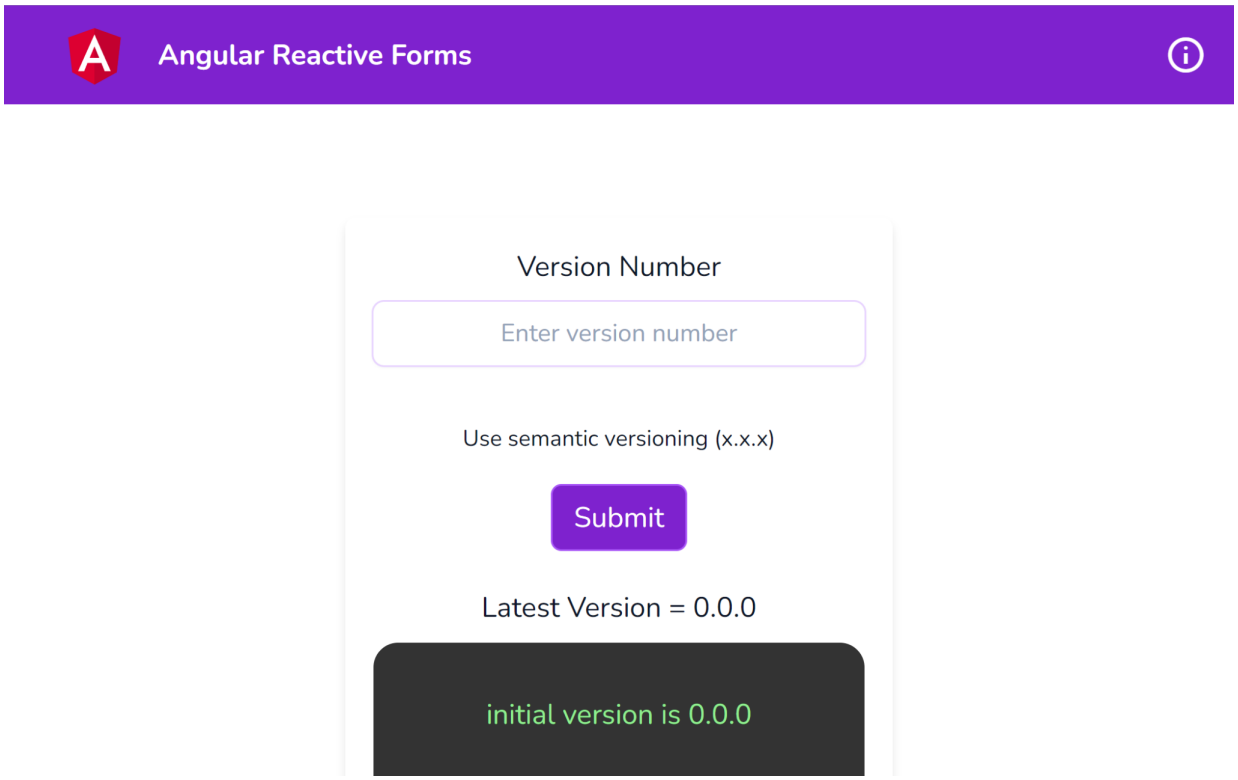


Figure 8.4 – ng-reactive-forms app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

So far, we have an app that has a `VcLogsComponent`, which shows a bunch of version logs that we create. We also have `VersionControlComponent`, which has a form by means of which the release logs will be created. We now have to make our current form a Reactive form using the Reactive forms API. Let's get started:

1. First of all, we need to import `ReactiveFormsModule` into the imports of our `AppModule`. Let's do it by modifying the `app.module.ts` file as follows:

```
...
import { ReactiveFormsModule } from '@angular/forms';
...
@Component({
  ...,
  imports: [CommonModule, VcLogsComponent, ReactiveFormsModule]
})
export class VersionControlComponent  {...}
```

1. We'll create the Reactive form in a bit. First import the required dependencies in the `version-control.component.ts` file as follows. We'll create a `FormGroup` in our `VersionControlComponent` class with a control named `version`. Modify the `version-control.component.ts` file as follows:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import {
  FormControl,
  FormGroup,
  ReactiveFormsModule,
  Validators,
} from '@angular/forms';
...
```

1. Now we'll create a `FormGroup` in our `VersionControlComponent` class with a control named `version`. Modify the `version-control.component.ts` file as follows:

```
@Component({...})
export class VersionControlComponent {
  versionInput = '';
  versionName = '0.0.0';
  versionPattern = '([0-9]+)\.([0-9]+)\.([0-9]+)';
  versionForm = new FormGroup({
    version: new FormControl('', [
      Validators.required,
      Validators.pattern(this.versionPattern),
    ]),
  });
}
```

1. Now that we have the Reactive Form in place, let's bind in the template to the `<form>` element. Update the `version-control.component.html` file as follows:

```
<form [formGroup]="versionForm">
  ...
</form>
```

1. Great! Now that we have the form group bound, we can also bind the `version` form control. Modify the `version-control.component.html` file further as follows:

```
<form [formGroup]="versionForm">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input formControlName="version" name="version" type="text" class="form-control my-2 text-ce
    ...
  </div>
  ...
</form>
...
```

1. Let's decide what will happen when we submit this form. We'll call a method named `formSubmit` in the template and pass `versionForm` to it when the form is submitted. Modify the `version-control.component.html` file as follows:

```
<form
  [formGroup]="releaseForm"
  (ngSubmit)="formSubmit(versionForm)">
```

```
  ...
</form>
```

1. The `formSubmit` method doesn't yet exist. Let's create it now in the `VersionControlComponent` class. Modify the `version-control.component.ts` file as follows:

```
...
@Component({...})
export class VersionControlComponent {
  ...
  formSubmit(form: FormGroup) {
    if (!form.valid) {
      return;
    }
    this.versionName = form.controls['version'].value;
  }
}
```

If you try submitting the form now with an invalid value, you'll see that the version provided doesn't get added to the logs. But if you provide a valid version like 2.1.0, it will be added to the logs. However, this isn't great in terms of UX since it doesn't tell what's wrong.

1. Let's show some error messages based on the form validation we have. Update the `version-control.component.html` as follows:

```
<form [formGroup]="versionForm" (ngSubmit)="formSubmit(versionForm)">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    ...
    <small class="error block" *ngIf="versionForm.dirty && versionForm.controls.version.errors?.
      Version number is required
    </small>
    <small class="error block" *ngIf="versionForm.dirty && versionForm.controls.version.errors?.
      Version number does not match the pattern (x.x.x)
    </small>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
<app-vc-logs [vName]="versionName"></app-vc-logs>
```

If you now type something in the form and clear the input, you should see the error as follows:

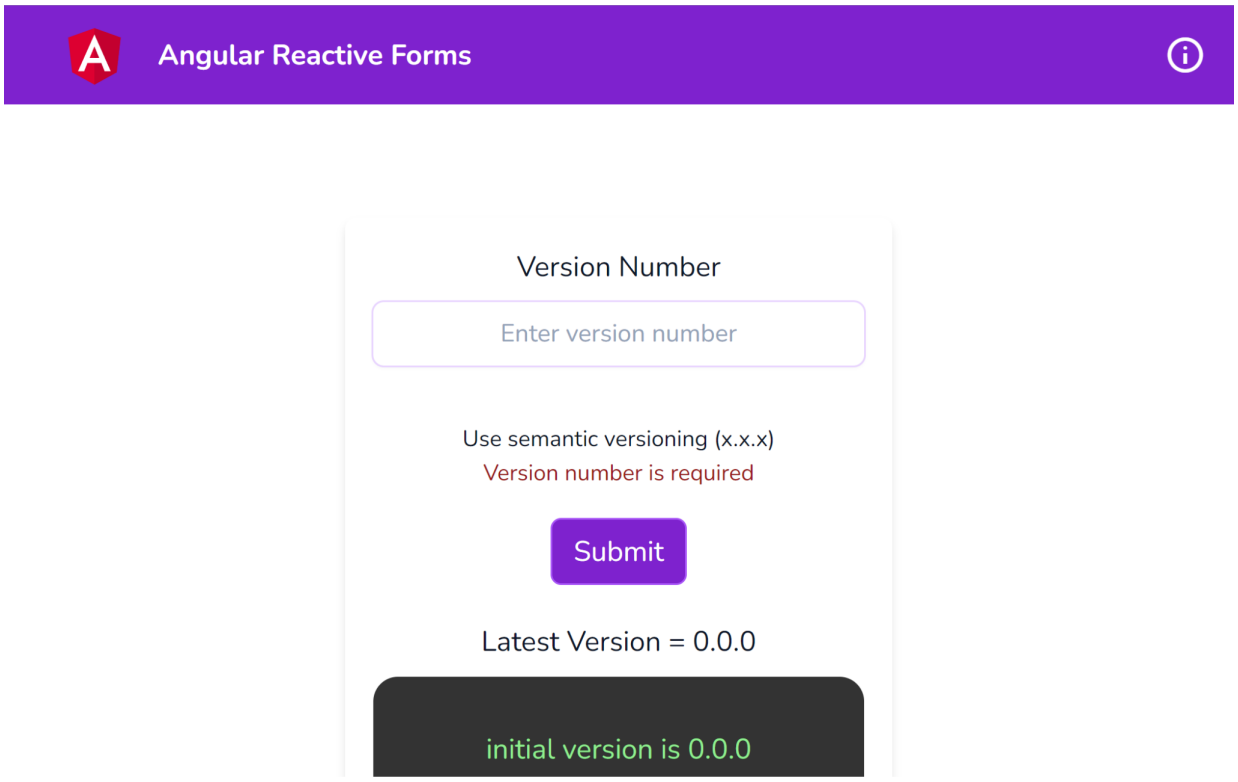


Figure 8.5 – ng-reactive-forms app with form validation

And if you try giving it a wrong value, you'll see a different error.Great! Within a few minutes, we were able to create our Reactive Form in Angular with form validation. See the next section to understand how it works.

How it works...

To work with Reactive Forms in Angular, we need to import the `ReactiveFormsModule` in the components in case of Standalone Components and in the NgModule if you're using modules. In the recipe we do exactly that. I.e. importing the `ReactiveFormsModule` in the `VersionControlComponent`'s `imports` array. After that we create a `FormGroup` which is a Reactive Form. And it takes an object as an argument for the constructor function. The object having the keys can have further Form Controls, nested Form Groups, or Form Arrays etc. In our recipe, we provide a key named `version` against a `FormControl`. The Form Control's constructor method takes two arguments in our recipe, the first one is the default value and the second is an array of `Validators`. Notice that we're using the following validators:

- `Validators.required`
- `Validators.pattern`

Also notice that the pattern we're using to validate semantic versioning is `([0-9]+)\.([0-9]+)\.([0-9]+)`. We then use the Form Group (i.e. the `versionForm` variable) in the template on the `<form>` element using the `[formGroup]` binding. And we then bind the `version` Form Control in the template using the `formControlName` attribute. We handle the form submission using the `(ngSubmit)` listener on the `<form>` element. In the `VersionControlComponent` class, we have the `formSubmit()` method to handle the form submission. This is also what we bind the event listener to in the template. When this function is called, we check if the form is valid. If that's not the case, we just don't do anything. Otherwise we set the value of the `versionName` property which adds a new log to the logs list.Finally, we show the errors based on the form validation in the template using the `errors`

property on the `version` form control. When the form is empty after making any initial change to it, the `errors` object on the form control gets the `required` property set to `true`. When the user adds a value to the input, the `required` property is removed and the validation of `pattern` gets triggered and if the pattern `([0-9]+)\.([0-9]+)\.([0-9]+)` doesn't match the value of the input, we get the `pattern` property added to the form control's `errors` object. Notice that in Step 8, we use these properties from the errors object to show the relevant error.

See also

Angular's guide to Reactive forms: https://angular.io/guide/reactive-forms

Testing forms in Angular

To make sure we build robust and bug-free forms for end users, it is a really good idea to have tests relating to your forms. It makes the code more resilient and less prone to errors. In this recipe, you'll learn how to test your template-driven forms using unit tests.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-testing-forms` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-testing-forms with-server` to serve the project along with the backend server

This should open the app in a new browser tab. And you should see the following:

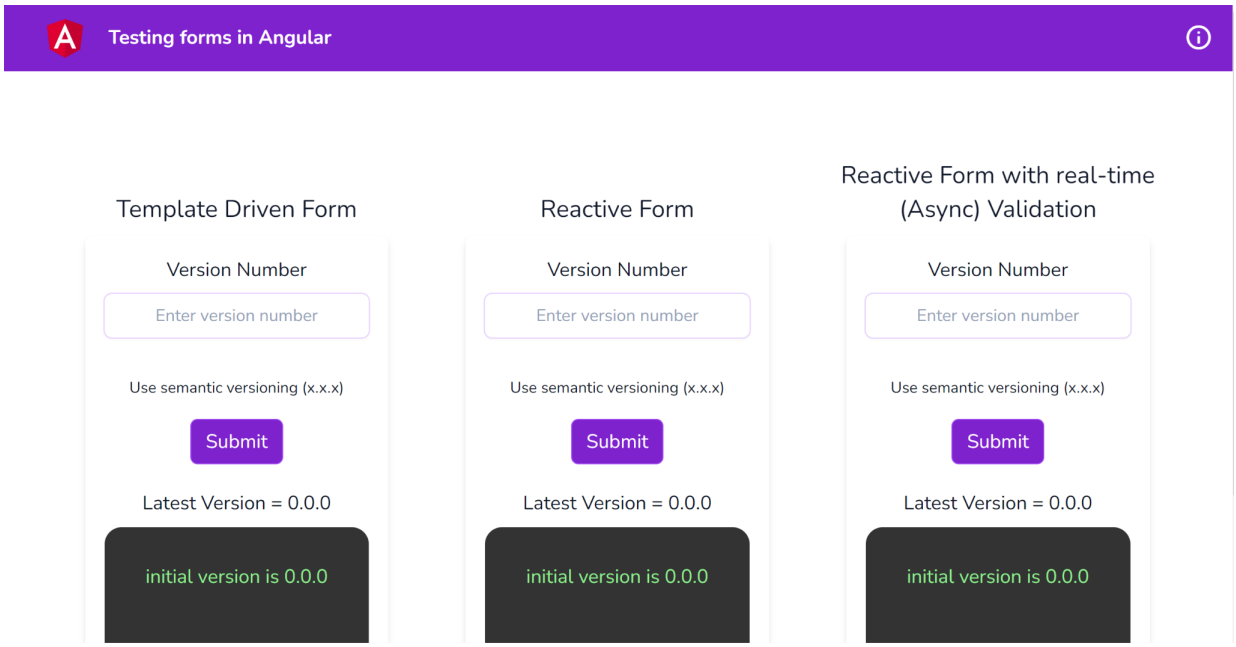


Figure 8.6 – ng-testing-forms app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

We have an app that contains three different components. The first one having a template-driven form, second one having a reactive form with validations, and the third having a reactive form with both synchronous and asynchronous validations. The implementation of all three of them is quite identical with the exception of the third component's validation kicking in as soon as the form is dirty instead of waiting to be submitted. The form also has validations applied to the inputs. Let's start looking into how we can test this form:

1. First of all, run the following command to run the unit tests:

```
npm run test ng-testing-forms
```

Once the command is run, you should see all the tests passing on the console as follows:

Figure 8.7 – Unit tests running with ng-testing-forms

1. Let's write the tests for the template driven form. This is relatively easy. Let's update the `version-control-tdf.component.spec.ts` file as follows:

```
...
describe('VersionControlTdfComponent', () => {
  ...
  it('should create', () => {...});
  it('should submit the form with valid version', () => {
    component.versionForm.controls['version'].setValue('2.2.4');
    fixture.debugElement.nativeElement.querySelector('button').click();
    expect(component.versionName).toBe('2.2.4');
  });
```

If you look at the tests now, you should see all the tests passing as follows:

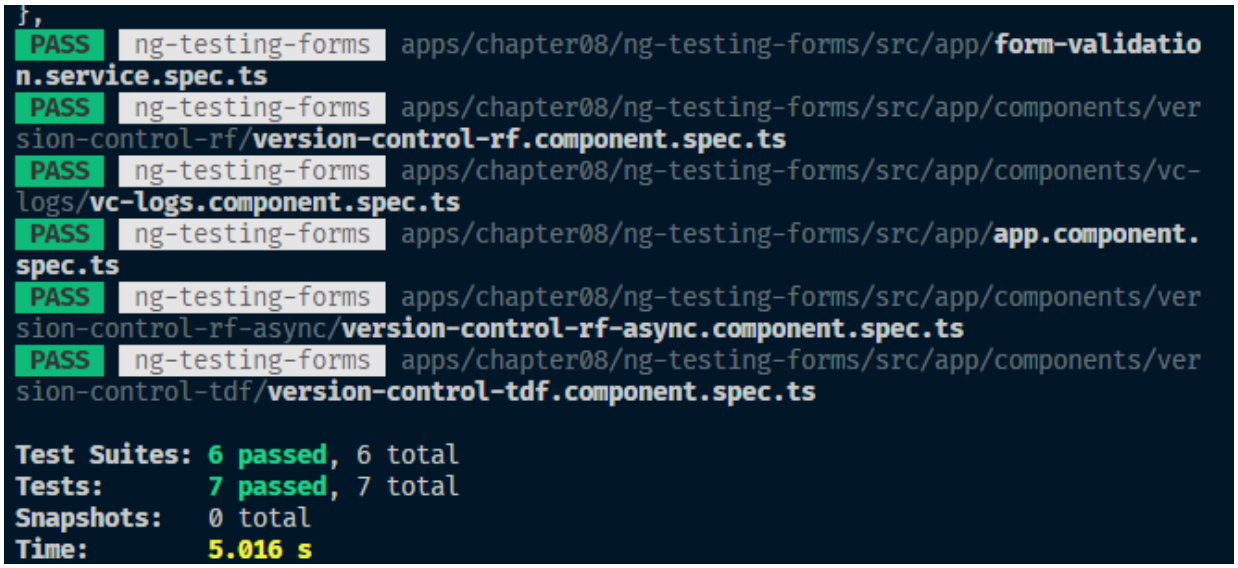


Figure 8.8 – First test for template driven forms passing

Let's add another test to the file. This will test if we're able to see the error message of input being required. Let's update the `version-control-tdf.component.spec.ts` file as follows:

```
it('should show required error', () => {
    component.versionForm.controls['version'].setValue('2.2.4');
    fixture.detectChanges();
    component.versionForm.controls['version'].setValue('');
    fixture.detectChanges();
    fixture.debugElement.nativeElement.querySelector('button').click();
    fixture.detectChanges();
    expect(component.versionName).toBe('0.0.0');
    expect(
      fixture.debugElement.nativeElement
        .querySelector('.error')
        .textContent.trim()
    ).toBe('Version number is required');
  });
```

1. Let's add one more test. This will test if we're able to see the message for the pattern of the version being valid. Add the following test to the same file:

```
it('should show pattern error', () => {
    component.versionForm.controls['version'].setValue('2.2.4');
    fixture.detectChanges();
    component.versionForm.controls['version'].setValue('invalid input');
    fixture.detectChanges();
    fixture.debugElement.nativeElement.querySelector('button').click();
    fixture.detectChanges();
    expect(component.versionName).toBe('0.0.0');
    expect(
      fixture.debugElement.nativeElement
        .querySelector('.error')
        .textContent.trim()
    ).toBe('Version number does not match the pattern (x.x.x)');
  });
```

Great! You should see all the tests still passing at the moment. The fun fact is that the tests are absolutely identical for the reactive form (with synchronous validators only) example as well.

1. Copy all the three tests we added in Step 2 - Step 4 to the file `version-control-rf.component.spec.ts`.

Now comes the tricky part. I.e. working with async validators. We can't just copy paste the tests for this one and expect the tests to pass for the case when we have an incorrect version provided in the form. We need some additional work to do in this case. First,. copy the same tests from the `version-control-tdf.component.spec.ts` file into the `version-control-rf-async.component.spec.ts` file.You will notice that we have 2 tests failing now as follows:

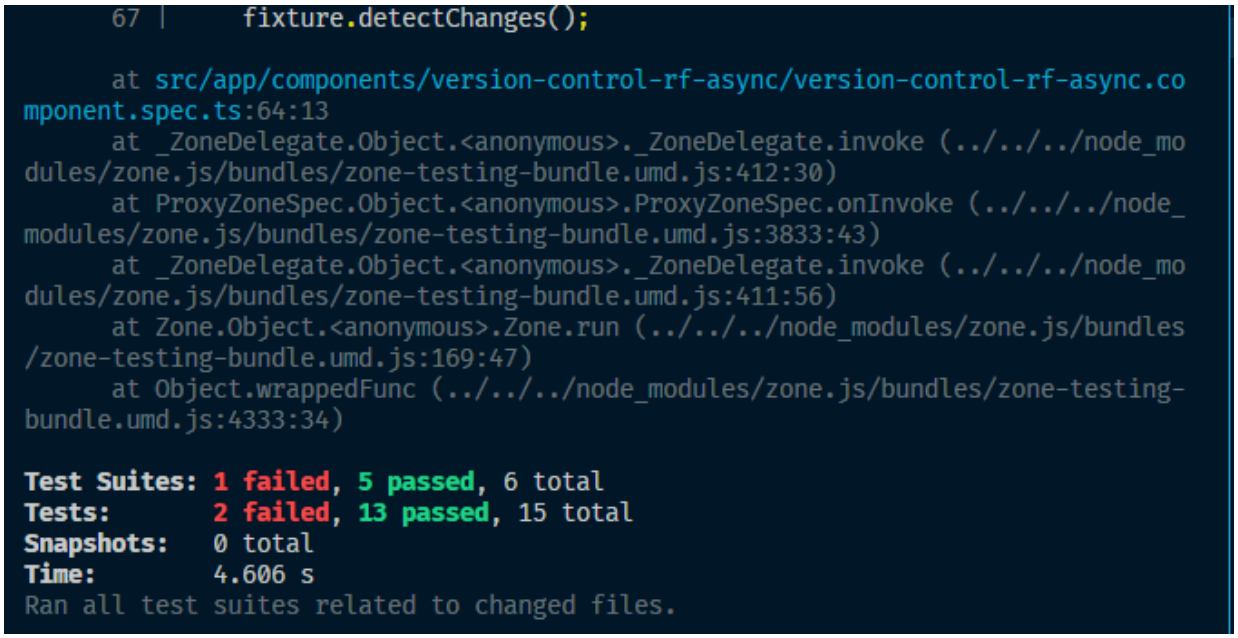


Figure 8.9 – Tests failing for Form with Asyncronous validators using a Service

1. We're going to mock the `FormValidationService` for our tests. Update the `version-control-rf-async.component.spec.ts` as follows:

```
import { provideHttpClient } from '@angular/common/http';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { of } from 'rxjs/internal/observable/of';
import { FormValidationService } from '../../form-validation.service';
import { VersionControlRfAsyncComponent } from './version-control-rf-async.component';
const FormValidationServiceMock = {
  versionValidator() {
    return () => of(null);
  },
  validateVersion() {
    return of(true);
  },
};
```

1. Let's provide our mock service to the `TestBed` for our tests in the same file as follows:

```
...
describe('VersionControlRfAsyncComponent', () => {
 ...
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [VersionControlRfAsyncComponent],
      providers: [
        provideHttpClient(),
        {
          provide: FormValidationService,
          useValue: FormValidationServiceMock,
        },
      ],
    }).compileComponents();
```

```
    ...
  });
});
```

With the above change, the test `'should submit the form with valid version'` and `'should show required error'` should be passing. Yayy! However, the last test fails. Let's fix that.

1. Let's update the final failing test. We're going to update our `FormValidationServiceMock` object to return a `pattern` error in case of an invalid input. Update the file as follows:

```
...
import { AbstractControl } from '@angular/forms';
import { of } from 'rxjs/internal/observable/of';
import { FormValidationService } from '../../form-validation.service';
import { VersionControlRfAsyncComponent } from './version-control-rf-async.component';
const FormValidationServiceMock = {
  versionValidator(control: AbstractControl) {
    return () => {
      if (control.value === 'invalid input') {
        return of({ pattern: true });
      }
      return of(null);
    };
  },
  validateVersion() {
    return of(true);
  },
};
```

And viola! All of the tests should be passing now as follows:

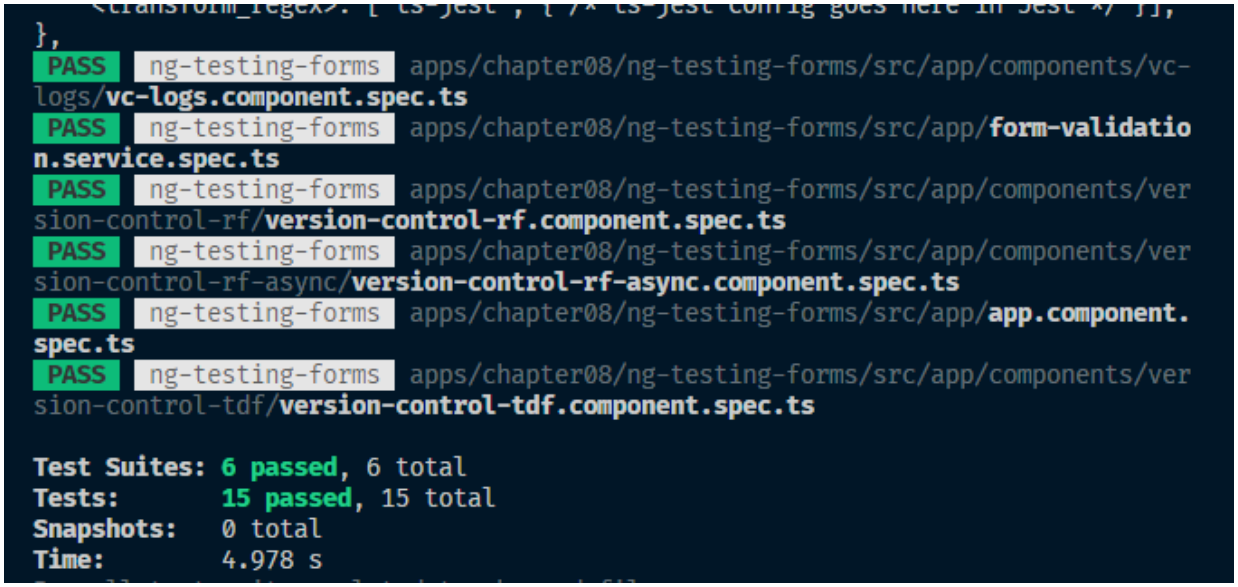


Figure 8.10 – All tests passing for the recipe

Awesome! You now know a bunch of techniques for testing your Angular forms. Some of these techniques might still require some explanation. See the next section to understand how it all works.

How it works...

Testing Angular forms can be a bit of a challenge as it depends on how complex the form is, what use cases you want to test, and how complex those use cases are. And this is true for the cases where you have several dependencies to work with. In our recipe, we started with the Template Driven forms and it was easy for us to test them since the component has a `ViewChild()` of the type `NgForm`. This makes it

easier to write our first test that sets the value of the `version` form control. Then we click the submit button and expect the component's `versionName` property's value to be the one we entered in the form. Easy peasy!For the second test, we first set the value of the form to `'2.2.4'` and then we empty the form by setting the value to an empty string. Then we submit the form and expect the value of the component's `versionName` property to not be changed and to still be `'0.0.0'`. But what we also check is that inside the form, we are able to see the error that says `'Version number is required'`. Notice that we use `fixture.debugElement.nativeElement.querySelector()` method to get the desired error element. And then we check its `textContent` value. For the test to check the pattern, we do the same as for the second test, but instead of emptying the input, we provide `'invalid input'` as the new value to the form control. The template driven form picks it and shows the error `'Version number does not match the pattern (x.x.x)'` which is what we expect in the test as well.The same three tests work exactly the same in the `Reactive Forms` example we have. The difference this time is that the `versionForm` property in the component is not `NgForm`, but rather a `FormGroup`. It is amazing that Angular has the same API (same methods) in both `NgForm` and `FormGroup` to set and retrieve the values so it makes our tests exactly the same.For the tests with Async validators, we did something special. Apart from copy pasting the tests from the other examples, we had to mock the `FormValidationService`. This is because this example had an asynchronous validator function, which comes from the `FormValidationService` class. We start by creating a stub named `FormValidationServiceMock` (we'll learn more about stubs in Chapter 10). Then we provide this stub in the `TestBed` against the `FormValidationService` class. Finally, we make sure that our `versionValidator` method in the stub returns the correct error when the input is not valid. And that's it. Without much changes in the tests themselves, we are able to test all three kinds of forms with minor adjustments. I hope you learnt quite a few techniques in this recipe.

See also

Testing template-driven forms: https://angular.io/guide/forms-overview#testing-template-driven-forms

Server side validation using asynchronous validator functions

Form validations are pretty straightforward in Angular, the reason being the super-awesome validators that Angular provides out of the box. These validators are synchronous, meaning that as soon as you change the input, the validators kick in and provide you with information about the validity of the values right away. But sometimes, you might rely on some validations from a backend API, or have some asynchronous logic that you need to execute to validate the form value. These situations would require something called asynchronous validators. In this recipe, you're going to create your first asynchronous validator.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-rf-async-validator` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-rf-async-validator with-server` to serve the project along with the backend server

This should open the app in a new browser tab. And you should see the following:

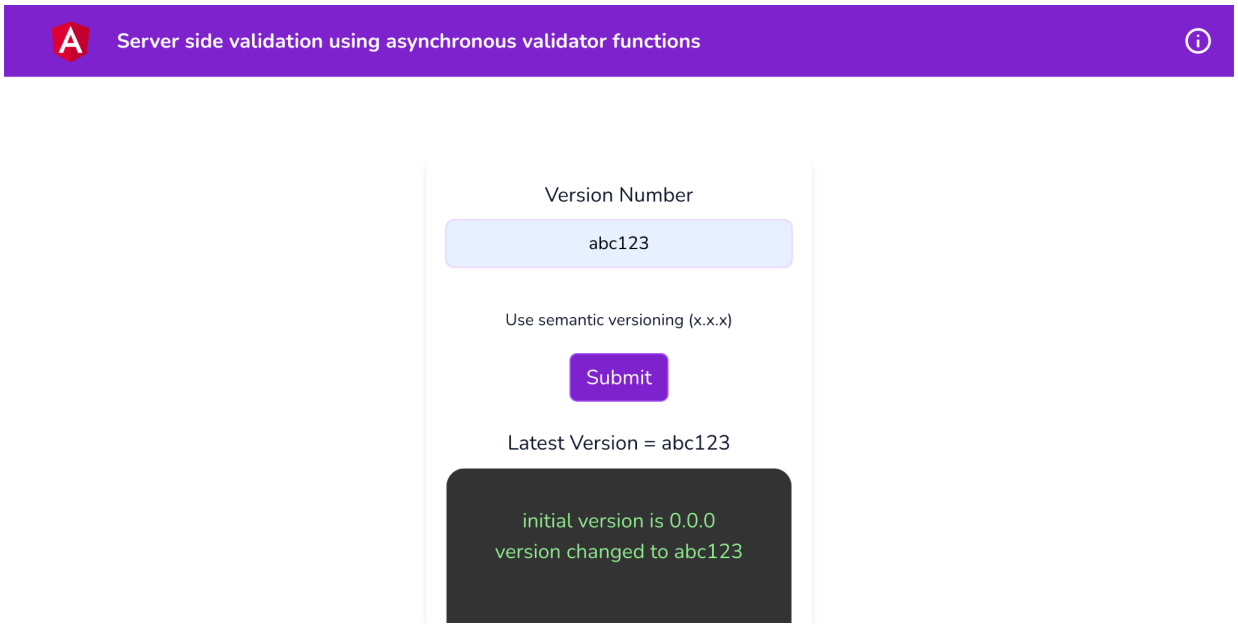


Figure 8.11 – Asynchronous validators app running on http://localhost:4200

Now that we have the app running, let's see the steps involved in this recipe in the next section.

How to do it...

We already have the backend having an end point that can accept a version as a query parameter and returns a boolean for if the provided version is a valid version according to the pattern (x.x.x) and is a new version than the last provided one. We'll create an async validator to make sure that the new releases have a valid version. Let's begin:

1. First, we will modify the `VersionService` class in the `version.service.ts` file as follows to add a method to validate the form input via the `http://localhost:3333/api/version/validate` endpoint.

```
...
export class VersionService {
  ...
  validateVersion(version: string): Observable<{ error: ValidationError }> {
    return this.http.get<{ error: ValidationError }>(
      `${this.apiBaseUrl}/validate?val=${version}`
    );
  }
  submitVersion(version: string): Observable<{ success: boolean }> {...};
  }
}
```

1. We will now create an **Async Validator Function** (`AsyncValidatorFn`) inside the `VersionService` class. This is something we'll bind later to the form in the app. Let's update the `version.service.ts` file as follows:

```
import { HttpClient } from '@angular/common/http';
import { Injectable, inject } from '@angular/core';
import { AbstractControl, AsyncValidatorFn, ValidationErrors } from '@angular/forms';
import { timer, switchMap, map } from 'rxjs';
import { Observable } from 'rxjs/internal/Observable';
...
export class VersionService {
  ...
```

```
  versionValidator(): AsyncValidatorFn {
    return (control: AbstractControl): Observable<ValidationErrors> => {
      return timer(500).pipe(
        switchMap(() => this.validateVersion(control.value)),
        map(({ error }) => {
          const errors: ValidationErrors = {};
          if (error !== null) {
            errors[error] = true;
          }
          return errors;
        })
      );
    };
  }
  ...
}
```

1. We will use the `versionValidator()` method we just created inside the `VersionControlComponent` class with the `versionForm`. To do that, let's modify the `version-control.component.ts` file as follows:

```
import { Component, OnInit, inject } from '@angular/core';
...
import { VersionService } from '../../version.service';
...
export class VersionControlComponent implements OnInit {
  ...
  versionService = inject(VersionService);

  ngOnInit() {
    this.versionForm.controls.version.setAsyncValidators(
      this.validationService.versionValidator()
    )
  }

  formSubmit(form: FormGroup) {...}
}
```

If you now try submitting a version equal to or lower than the last submitted one, you will receive an error as follows.

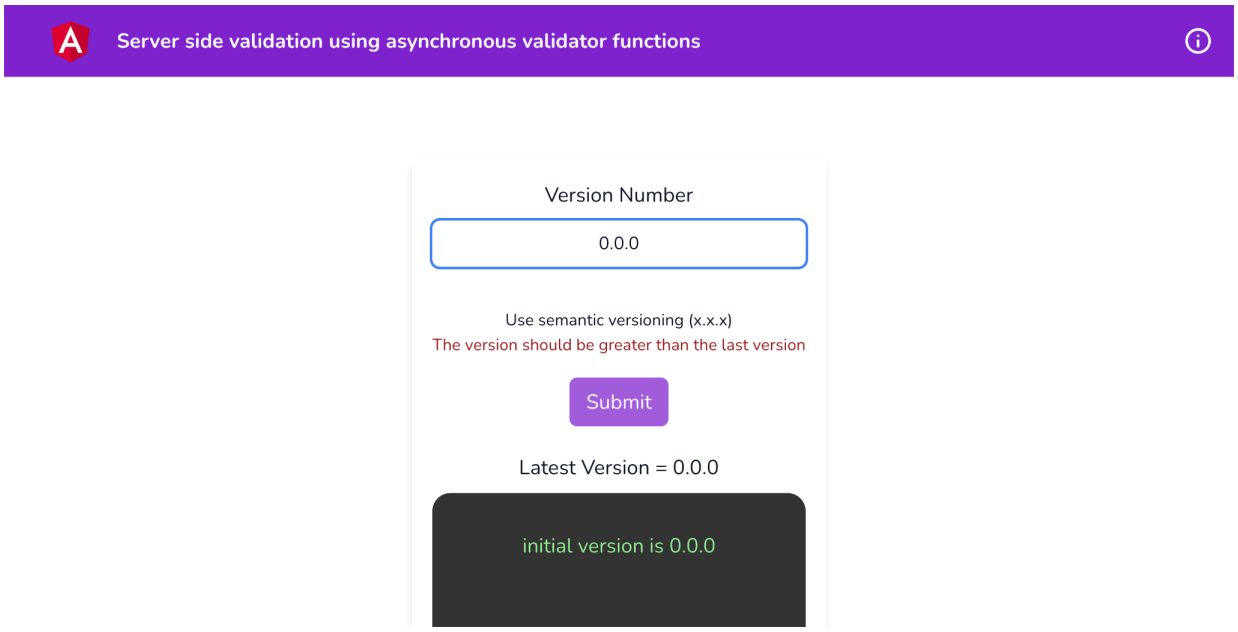


Figure 8.12 – Error being shown when a lower or equal version number is provided

Cool! So, you now know how to create an asynchronous validator function in Angular for form validation within Reactive forms. Since you've finished the recipe, refer to the next section to see how this works.

How it works

Angular provides an easy way to create async validator functions, and they're pretty handy too. We use an async validator function when there needs to be a validation that can be time consuming and we don't want to block the main thread. Or when we rely on a backend service for validation. For example, in this recipe, we started by creating the validator function named `versionValidator` inside a new service called `VersionService`. Notice that we are using a couple of RxJS operators in the function including `timer`, `switchMap`, and `map`. We use the `timer` operator to debounce and wait for the user to stop typing for `500` milliseconds. We then use the `switchMap` operator combined with the `validateVersion` method to make an HTTP call to the backend to validate the version. The benefit of `switchMap` here is that it cancels out existing HTTP calls, if any. And then we use the `map` operator to take out the error from the HTTP call. If there's no error, we return an empty object from the map function's callback. If we have an error, we set the `[error]` as `true` in the `errors` object as shown in **Step 4**.After creating the validator function, we attached it to the form control for the version name by using the `FormControl.setAsyncValidators()` method in the `VersionControlComponent` class. We then used the validation errors named `pattern` and `oldVersion` in the template to show the relevant error message on the UI.

See also

AsyncValidator Angular docs: https://angular.io/api/forms/AsyncValidator#provide-a-custom-async-validator-directive

Implementing Complex Forms with Reactive Form Arrays

This is definitely one of the most requested from the readers of the first edition of the Angular Cookbook. In this recipe, we'll work with Reactive Forms and specifically, the `FormArray` class from Reactive Forms. We're going to implement a complex form which has a list of projects that can be submitted with the forms. The user will be able to add as many projects as they want and will be able to remove the ones they don't want.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-form-arrays` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-form-arrays` to serve the project

This should open the app in a new browser tab. And you should see the following:

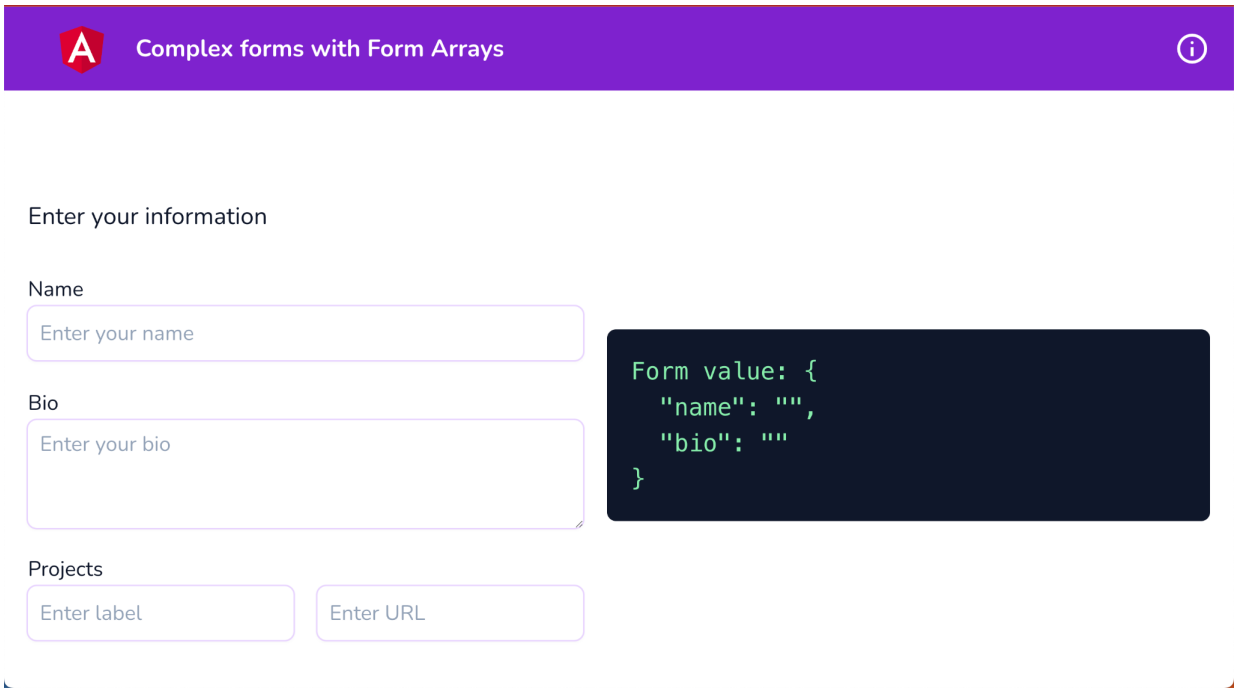


Figure 8.13 – Reactive Form Arrays app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

We have an app that already implements a Reactive Form. However, we can only add one project so far. Let's use the Reactive Form Arrays to allow the user to submit multiple projects:

1. First of all, let's use the Reactive Form Builder to add a form array to the existing Reactive Form in the `portfolio-form.component.ts` file as follows:

```
...
export class PortfolioFormComponent {
  fb = inject(FormBuilder);
  portfolioForm = this.fb.group({
    name: ['', Validators.required],
    bio: [''],
    projects: this.fb.array<FormGroup>([])
  })
  get projectsFormArr() {
    return this.portfolioForm.controls.projects;
  }
  ...
}
```

1. Now that we have the `projects` Form Array, let's use that in the template so we can show the form inputs for the projects according to the form array. Let's modify the portfolio-form.component.html file as follows:

```
<div class="flex gap-4 items-center flex-col md:flex-row">
  <form [formGroup]="portfolioForm" class="flex-1 w-full md:w-auto">
    ...
    <section formArrayName="projects">
      <label class="text-sm">Projects</label>
      <fieldset [formGroup]="projectControl" class="flex gap-4 mb-4 justify-between" *ngFor="let
        <input formControlName="label" type="text" class="form-control" placeholder="Enter label'
        <input formControlName="url" type="url" class="form-control" placeholder="Enter URL">
```

```
      </fieldset>
    </section>
  </form>
  ...
</div>
```

1. You'll notice that the object on the right in the app has a `projects` array now. However, we see no inputs for the projects at the moment. That's because the form array is empty as shown in *Figure 8.14*.

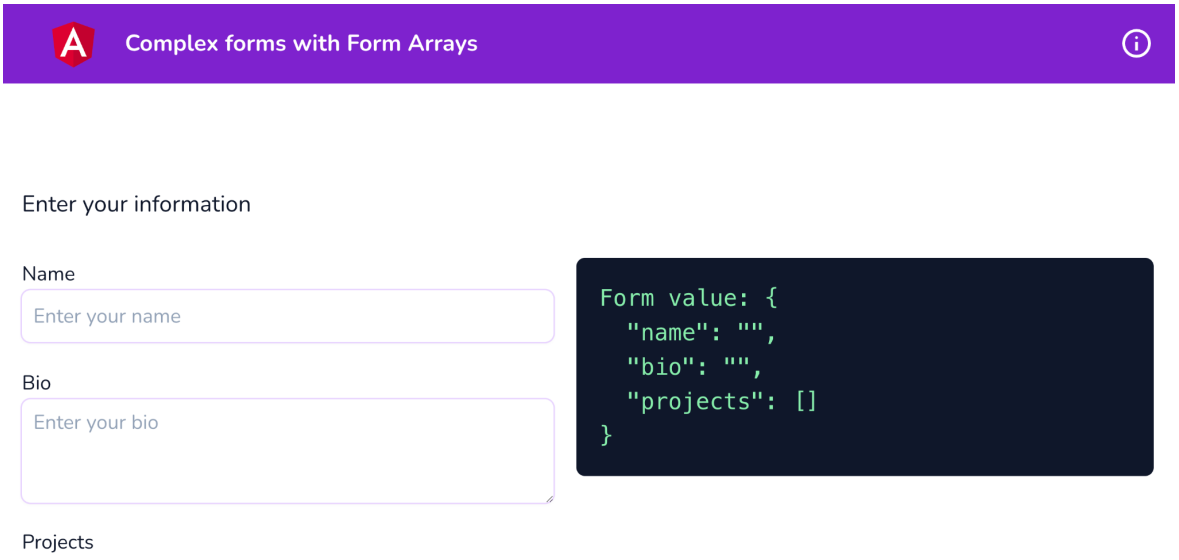


Figure 8.14 – Reactive Form Arrays app running on http://localhost:4200

2. We'll add one form group to the form array by default when the app starts. To do this, modify the `portfolio-form.component.ts` file as follows:

```
import { Component, OnInit, inject } from '@angular/core';
...
export class PortfolioFormComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.addNewProject();
  }
  addNewProject() {
    this.projectsFormArr.push(
      this.fb.group({
        label: ['', Validators.required],
        url: ['', Validators.required]
      })
    )
  }
  ...
}
```

If you look at the app now, you should see one set of inputs for the projects as shown in *Figure 8.15*.

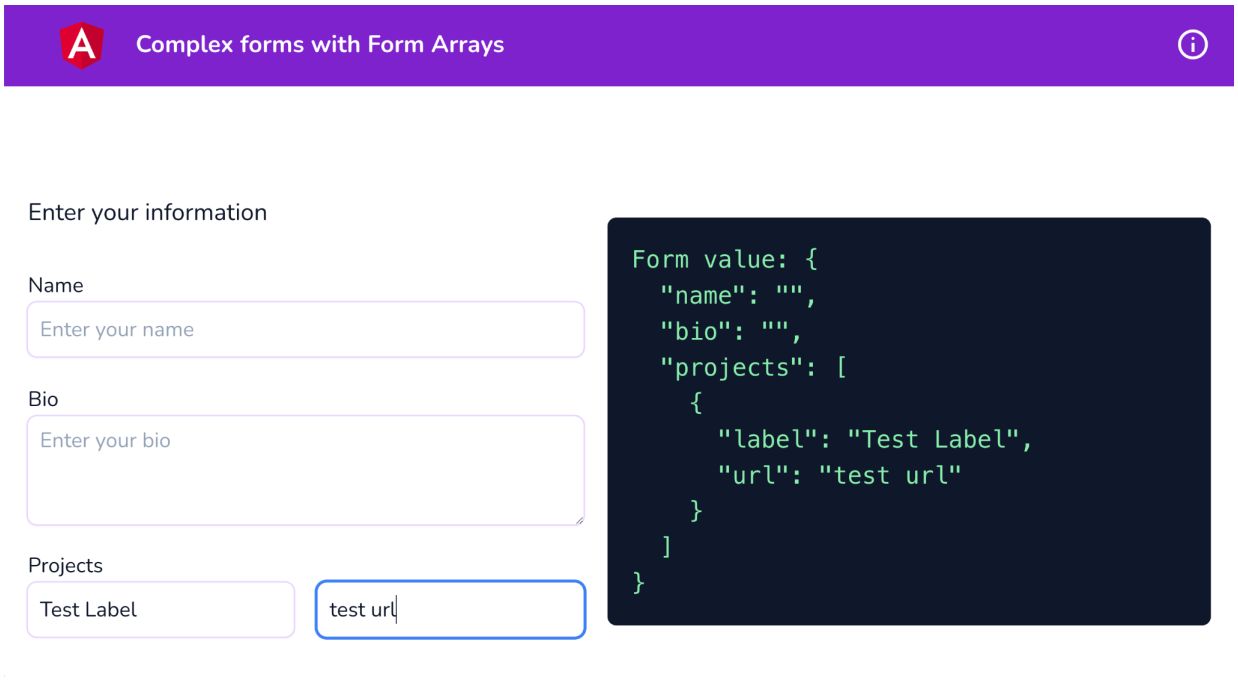


Figure 8.15 – One set of inputs for projects by default

1. Let's add a button for the user to be able to add more projects. We'll modify the `portfolio-form.component.html` file as follows:

```
<section formArrayName="projects">
      <label class="text-sm">Projects</label>
      <fieldset [formGroup]="projectControl" class="flex gap-4 mb-4 justify-between" *ngFor="let
        ...
        <input formControlName="url" type="url" class="form-control" placeholder="Enter URL">
        <button type="button" [style.visibility]="isLast ? 'visible' : 'hidden'" [hidden]="!isLas
      </fieldset>
</section>
```

With the code above, you should see new project inputs appearing on the click of the + button.

1. Let's add our final feature for this recipe. And that is to show buttons to remove the particular project's input row. We'll first add a method in the `portfolio-form.component.ts` file, that receives the index of the row to be removed as an argument:

```
...
export class PortfolioFormComponent implements OnInit {
  ...
  addNewProject() {...}
  removeProject(index: number) {
    this.projectsFormArr.removeAt(index);
  }
  ...
}
```

1. Now we'll update the template to add the remove button. Update the `portfolio-form.component.html` file as follows:

```
<section formArrayName="projects">
      <label class="text-sm">Projects</label>
      <fieldset [formGroup]="projectControl" class="flex gap-4 mb-4 justify-between" *ngFor="let
        <input formControlName="label" type="text" class="form-control" placeholder="Enter label"
        <input formControlName="url" type="url" class="form-control" placeholder="Enter URL">
        <button type="button" [hidden]="isLast" (click)="removeProject(i)">-</button>
```

```
      <button type="button" [style.visibility]="isLast ? 'visible' : 'hidden'" [hidden]="!isLas
    </fieldset>
  </section>
```

If you now look at the app, you should be able to add or remove projects to the form as you like, as shown in *Figure 8.16*.

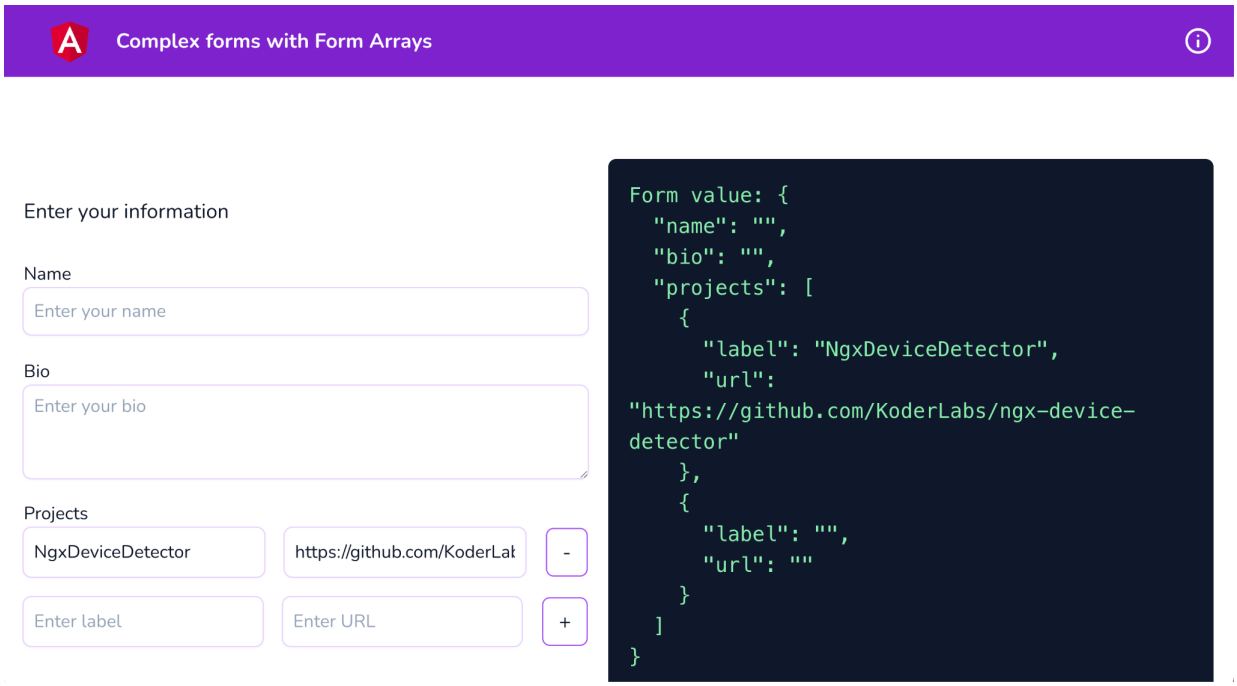


Figure 8.16 – Final result with form arrays

Great! You now know how to work with Reactive Form Arrays. Refer to the next section to understand how it all works.

How it works...

Angular Form Array is an amazing tool built into the Reactive Forms in Angular. It aggregates all values of each child Form Control or Form Group into an Array. The beauty of the Form Array is that if any one of the child Form Control is invalid, the whole array becomes invalid. In this recipe, we start by using the `FormBuilder` from the `ReactiveFormsModule` in the `PortfolioFormComponent` class. We initialize it with an empty `FormArray` and then push one `FormGroup` to it when the component mounts (using `ngOnInit()` method). This is so we can see one set of form inputs for the projects. Notice that we use the `addNewProject` method for this. Notice that we are using a `getter` function named `projectsFormArr` so we can easily access the `FormArray` to be able to loop over it, to add Form Groups to it and to remove the desired Form Groups from it when needed. We then loop over the `projectsFormArr.controls` array so we can display the inputs via the HTML template using the `*ngFor` directive.Then we modify the template to add the + button so we can add more projects to the form array. Notice the statement in the `*ngFor` having `let isLast = last`. The `ngFor` directive provides us the first and last variables automatically so we can use them in our template. This statement creates a variable named `isLast` for us and assigns the value of the `last` variable provided by `ngFor` directive to it. This is so we can show the + button only on the last row instead of on every row.Finally, we add the remove button to each row so we can remove the respective `FormGroup` from the `FormArray`. Notice that here we also add `let i = index` to the template in the ngFor directive. This is because our `removeProject` method in the TypeScript file expects the index of the `FormGroup` in the `FormArray` to be passed in it, when being called. Hence we're able to get the index for each `FormGroup` from the `ngFor` directive and can pass the index to the `removeProject` method. And we use `[hidden]="isLast"` on each remove button so it is hidden for

the last row only. Because we show the + button on the last row. And because we need to have at least one row for the projects.

See also

Angular Form Array: https://angular.io/api/forms/FormArray

Writing your own custom form control using ControlValueAccessor

Angular forms are great. While they support the default HTML tags like `input`, `textarea` etc., sometimes, you would want to define your own components that take a value from the user. It would be great if the variables of those inputs were a part of the Angular form you're using already.In this recipe, you'll learn how to create your own custom Form Control using the `ControlValueAccessor` API, so you can use the Form Control with both Template Driven forms and Reactive Forms.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-form-cva` inside the cloned repository:

1. Open the code repository in your Code Editor.
2. Open the terminal, navigate to the code repository directory and run `npm run serve ng-form-cva` to serve the project

This should open the app in a new browser tab. And you should see the following:

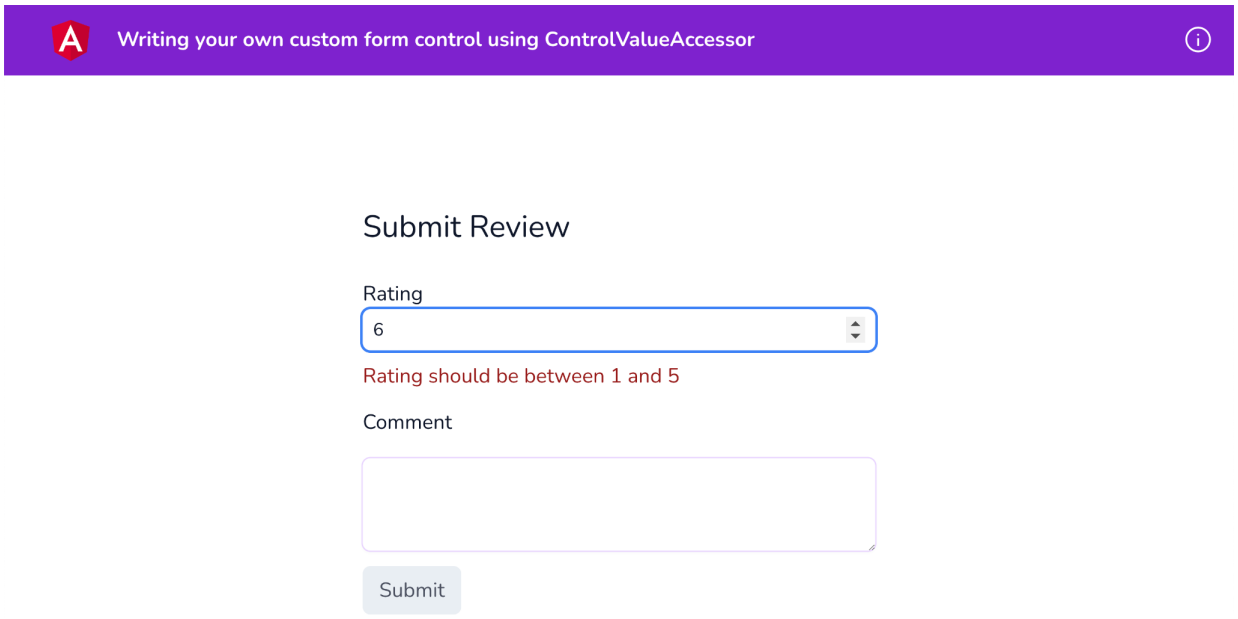


Figure 8.17 – Custom form control app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

We have a simple Angular app. It has two inputs and a **Submit** button. The inputs are for a review and they ask the user to provide a value for the rating of this imaginary item and any comments the user

wants to provide. We'll convert the Rating input into a custom Form Control using the `ControlValueAccessor` API. Let's get started:

1. Let's create a component for our custom form control. Open the terminal in the project root and run the following command:

```
cd start && npx nx g c components/rating --standalone
```

1. We'll now create the stars UI for the rating component. Modify the `rating.component.html` file as follows:

```
<div class="rating">
  <div class="rating__star" [ngClass]="{
      'rating__star--active':
        (!isMouseOver && value >= star) ||
        (isMouseOver && hoveredRating >= star)
    }" (mouseenter)="onRatingMouseEnter(star)" (mouseleave)="onRatingMouseLeave()" (click)="sele
    *ngFor="let star of [1, 2, 3, 4, 5]; let i = index">
    <span class="material-symbols-outlined">
      star
    </span>
  </div>
</div>
```

1. Add the styles for the rating component to the `rating.component.scss` file as follows:

```
.rating {
  display: flex;
  margin-bottom: 10px;
  &__star {
    cursor: pointer;
    color: grey;
    padding: 0 6px;
    &:first-child {
      padding-left: 0;
    }
    &:last-child {
      padding-right: 0;
    }
    &--active {
      color: orange;
    }
  }
}
```

1. We also need to modify the `RatingComponent` class to introduce the necessary methods and properties. Let's modify the `rating.component.ts` file as follows:

```
/* eslint-disable @typescript-eslint/no-empty-function */
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
@Component({
  ...
  standalone: true,
  imports: [CommonModule]
})
export class RatingComponent {
  value = 2;
  hoveredRating = 2;
  isMouseOver = false;
  onRatingMouseEnter(rating: number) {
    this.hoveredRating = rating;
    this.isMouseOver = true;
  }
  onRatingMouseLeave() {
    this.hoveredRating = 0;
    this.isMouseOver = false;
  }
```

```
  selectRating(rating: number) {
    this.value = rating;
  }
}
```

1. Let's import the `RatingComponent` class in the `HomeComponent` class because Rating Component is a `standalone` component. Update the `home.component.ts` file as follows:

```
import { RatingComponent } from '../components/rating/rating.component';
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss'],
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule, RatingComponent]
})
export class HomeComponent {
  ...
}
```

1. Now we need to use this rating component instead of the input that we already have in the `home.component.html` file. Modify the file as follows:

```
<div class="home">
  <div class="review-container">
    ...
    <form class="input-container" [formGroup]=    "reviewForm" (ngSubmit)="submitReview(reviewFo
      <div class="mb-3">
        <label for="ratingInput" class="form-        label">Rating</label>
        <app-rating formControlName="rating"></app-rating>
      </div>
      <div class="mb-3">
        ...
      </div>
      <button id="submitBtn" [disabled]="reviewForm.      invalid" class="btn btn-dark" type="sub
    </form>
  </div>
</div>
```

If you refresh the app now and hover on the stars, you can see the color changing as you hover over the stars. The selected rating is also highlighted as follows:

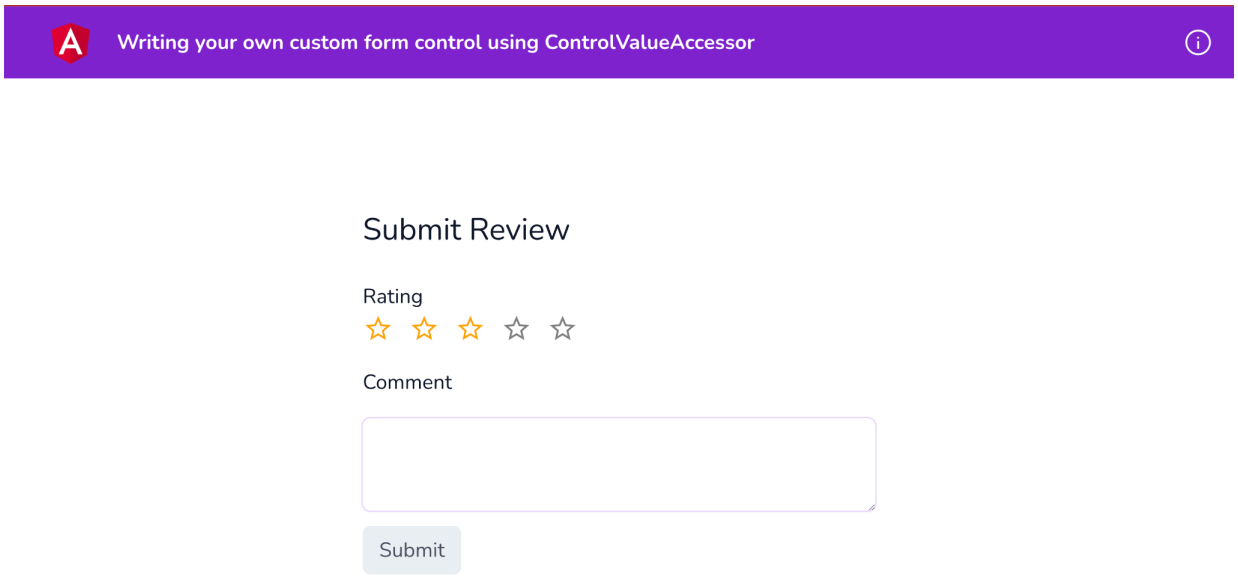


Figure 8.18 – Rating component with hovered stars

1. Let's now implement the `ControlValueAccessor` interface for our rating component. It requires a couple of methods to be implemented and we'll start with the `onChange()` and `onTouched()` methods. Modify the `rating.component.ts` file as follows:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { ControlValueAccessor } from '@angular/forms';
@Component({...})
export class RatingComponent implements ControlValueAccessor {
  ...
  isMouseOver = false;
  onChange: any = () => { };
  onTouched: any = () => { };
  ...
  registerOnChange(fn: any){
    this.onChange = fn;
  }
  registerOnTouched(fn: any) {
    this.onTouched = fn;
  }
}
```

1. We'll now add the required methods to disable the input when required and to set the value of the form control, in other words, the `setDisabledState()` and `writeValue()` methods. We'll also add the `disabled` and `value` properties to our `RatingComponent` class as follows:

```
...
import { Component, Input } from '@angular/core';
...
export class RatingComponent implements ControlValueAccessor {
  ...
  isMouseOver = false;
  @Input() disabled = false;
  ...
  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }
  writeValue(value: number) {
    this.value = value;
  }
}
```

1. We need to use the `disabled` property to prevent any UI changes when it is `true`. The value of the `value` variable shouldn't be updated either. Modify the `rating.component.ts` file to do so as follows:

```
...
@Component({...})
export class RatingComponent implements OnInit, ControlValueAccessor {
  ...
  isMouseOver = false;
  @Input() disabled = true;
  ...
  onRatingMouseEnter(rating: number) {
    if (this.disabled) return;
    this.hoveredRating = rating;
    this.isMouseOver = true;
  }
  ...
  selectRating(rating: number) {
    if (this.disabled) return;
    this.value = rating;
  }
  ...
}
```

With the above change, you'll notice that hovering the stars doesn't do anything now; because of the `disabled` property being set to `true`.

1. Let's make sure that we send the value of the `value` variable to `ControlValueAccessor` because that's what we want to access later. Also, let's set the `disabled` property back to `false`. Update the `selectRating` method in the `RatingComponent` class as follows:

```
...
export class RatingComponent implements ControlValueAccessor {
  ...
  @Input() disabled = false;
  constructor() { }
  ...
  selectRating(rating: number) {
    if (this.disabled) return;
    this.value = rating;
    this.onChange(rating);
  }
  ...
}
```

1. We need to tell Angular that our `RatingComponent` class has a value accessor, otherwise using the `formControlName` attribute on the `<app-rating>` element will throw errors. Let's add an `NG_VALUE_ACCESSOR` provider to the `RatingComponent` class's decorator as follows:

```
import { Component, forwardRef, Input, OnInit } from '@angular/core';
import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';
@Component({
  ...
  imports: [CommonModule],
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => RatingComponent),
    multi: true
  }]
})
...
```

If you refresh the app now, select a rating, and hit the **Submit** button, you should see the values being logged as follows:

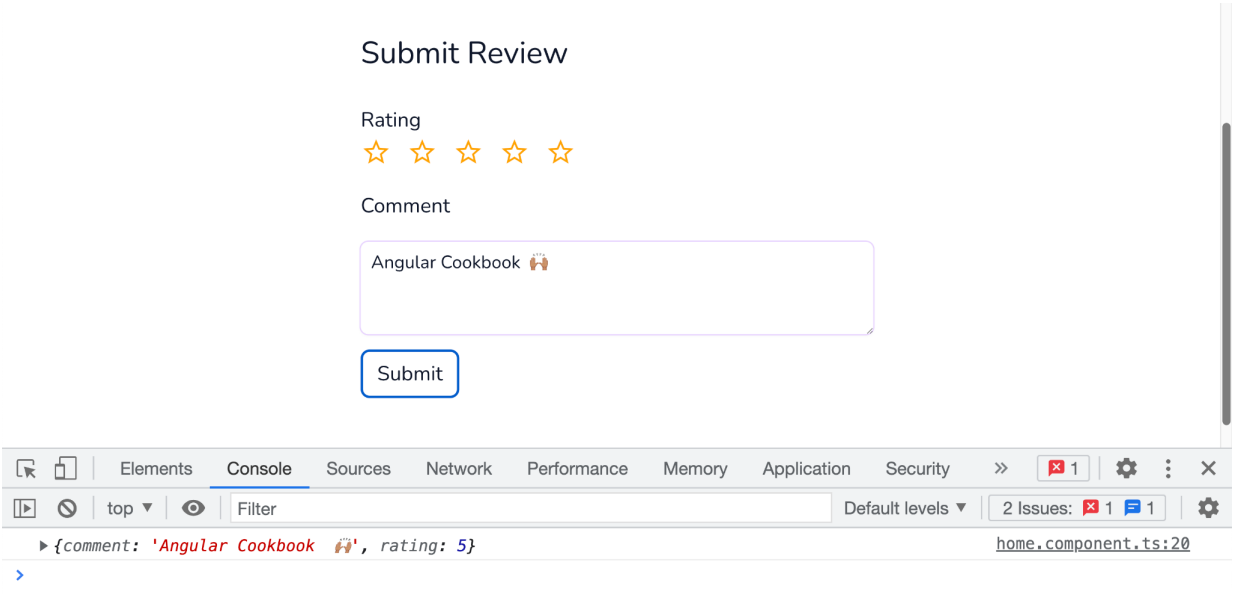


Figure 8.19 – Form value logged using the custom form control

Voilà! You just learned how to create a custom form control using `ControlValueAccessor`. Refer to the next section to understand how it works.

How it works...

We started the recipe by creating a component that we can use to provide a rating for the review we have to submit. We started off by adding the template and the styles for the rating component. Notice that we are using an `[ngClass]` directive on each of the star elements to add the `rating__star--active` class conditionally. Let's discuss each condition now: `(isMouseOver && hoveredRating >= star)`: This condition relies on the `isMouseOver` and `hoveredRating` variables. The `isMouseOver` variable becomes `true` as soon as we mouse over any star and is turned back to `false` when we move away from the star. This means that it is only `true` when we're hovering over a star. `hoveredRating` tells us which star we're hovering over at the moment and is assigned the star's value, in other words, a value from `1` to `5`. So, this condition is only true when we're doing a mouseover, and the hovered star's rating is greater than the value of the current star. So, if we're hovering over the fourth star, all the stars from value `1` to `4` will be highlighted as they'll have the `rating__star--active` class conditionally assigned to them. `(!isMouseOver && value >= star)`: This condition relies on the `isMouseOver` variable that we discussed previously and the `value` variable. The `value` variable holds the value of the selected rating, which is updated when we click on a star. So, this condition is applied when we're not doing a mouseover and we have the value of the `value` variable greater than the current star. This is especially beneficial when you have a greater value assigned to the `value` variable and try to hover over a star with a lesser value, in which case, all the stars with values greater than the hovered star will not be highlighted.

1. Then we used three events on each star: `mouseenter`, `mouseleave`, and `click`, and then used our `onRatingMouseEnter`, `onRatingMouseLeave`, and `selectRating` methods, respectively, for these events. All of this was designed to ensure that the entire UI is fluent and has a good user experience. We then implemented the `ControlValueAccessor` interface for our rating component. When we do that, we need to define the `onChange` and `onTouched` methods as empty methods, which we did as follows:

```
onChange: any = () => { };
onTouched: any = () => { };
```

1. Then we used the `registerOnChange` and `registerOnTouched` methods from `ControlValueAccessor` to assign our methods as follows:

```
registerOnChange(fn: any){
  this.onChange = fn;
}
registerOnTouched(fn: any) {
  this.onTouched = fn;
}
```

1. We registered these functions because whenever we do a change in our component and want to let `ControlValueAccessor` know that the value has changed, we need to call the `onChange` method ourselves. We do that in the `selectRating` method as follows, which makes sure that when we select a rating, we set the form control's value to the value of the selected rating:

```
selectRating(rating: number) {
  if (this.disabled) return;
  this.value = rating;
  this.onChange(rating);
}
```

1. The other way around is when we need to know when the form control's value is changed from outside the component. In this case, we need to assign the updated value to the `value` variable. We do that in the `writeValue` method from the `ControlValueAccessor` interface as follows:

```
writeValue(value: number) {
  this.value = value;
}
```

1. What if we don't want the user to provide a value for the rating? In other words, we want the rating form control to be disabled. For this, we did two things. First, we used the `disabled` property as an `@Input()`, so we can pass and control it from the parent component when needed. Secondly, we used the `setDisabledState` method from the `ControlValueAccessor` interface, so whenever the form control's `disabled` state is changed, apart from `@Input()`, we set the `disabled` property ourselves.
2. Finally, we wanted Angular to know that this `RatingComponent` class has a value accessor. This is so that we can use the Reactive forms API, specifically, the `formControlName` attribute with the `<app-rating>` selector, and use it as a form control. To do that, we provide our `RatingComponent` class as a provider to its `@Component` definition decorator using the `NG_VALUE_ACCESSOR` injection token as follows:

```
@Component({
  ...
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => RatingComponent),
    multi: true
  }]
})
```

1. Note that we're using the `useExisting` property with the `forwardRef()` method providing our `RatingComponent` class in it. We need to provide `multi: true` because Angular itself registers some value accessors using the `NG_VALUE_ACCESSOR` injection token, and there may also be third-party form controls.
2. Once we've set everything up, we use `formControlName` on our rating component in the `home.component.html` file as follows:

```
<app-rating formControlName="rating"></app-rating>
```

See also

Custom form control in Angular by Thoughtram:
https://blog.thoughtram.io/angular/2016/07/27/custom-form-controls-in-angular-2.htmlControlValueAccessor docs: https://angular.io/api/forms/ControlValueAccessor