Jak wdrażać wzorce reaktywne w aplikacjach **Angulara**

Przewodnik po RxJS 7

Lamis Chebbi

Tytuł oryginału: Reactive Patterns with RxJS for Angular: A practical guide to managing your Angular application's data reactively and efficiently using RxJS 7

Tłumaczenie: Grzegorz Skorczyński

ISBN: 978-83-8322-489-3

Copyright © Packt Publishing 2022. First published in the English language under the title 'Reactive Patterns with RxJS for Angular – (9781801811514)'

Polish edition copyright © 2023 by Helion S.A.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/jawdwz
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/jawdwz.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorce ... 11

O recenzentach ... 12

Przedmowa ... 13

CZĘŚĆ I. Wstęp

Rozdział 1. Potęga podejścia reaktywnego ... 19
- Wymagania techniczne ... 20
- Filary programowania reaktywnego ... 20
  - Strumienie danych ... 20
  - Wzorzec Obserwator ... 21
- Wykorzystanie RxJS w Angularze i jego zalety ... 22
  - Moduł klienta HTTP ... 23
  - Moduł routera ... 24
  - Formularze reaktywne ... 28
  - Emiter zdarzeń ... 28
  - Potok asynchroniczny — AsyncPipe ... 29
- Diagram marmurkowy — nasza tajna broń ... 30
- Podsumowanie ... 32

Rozdział 2. RxJS 7 — główne funkcje ... 33
- Wymagania techniczne ... 33
- Ulepszenia wpływające na rozmiar pakietu ... 34
- Przegląd zmian ułatwiających pisanie w TypeScript ... 36
- Dlaczego funkcja toPromise() została uznana za przestarzałą ... 36
  - Metoda firstValueFrom() ... 38
  - Metoda lastValueFrom() ... 39
  - Puste błędy ... 40
- Ulepszenia spójności API ... 41
- Podsumowanie ... 42

Kup ksi k   Pole ksi k

Rozdział 3. Przewodnik po aplikacji ... 43
Wymagania techniczne ... 43
Historyjki użytkowników naszej aplikacji ... 44
Widok pierwszy — strona główna ... 44
Widok drugi — interfejs dodawania nowego przepisu ... 45
Widok trzeci — interfejs „Moje przepisy” ... 46
Widok czwarty — interfejs „Ulubione” ... 46
Widok piąty — interfejs modyfikowania przepisu ... 47
Widok szósty — interfejs szczegółów przepisu ... 48
Architektura naszej aplikacji ... 48
Przegląd komponentów ... 49
Podsumowanie ... 50

CZĘŚĆ II. W stronę wzorców reaktywnych

Rozdział 4. Pobieranie danych jako strumieni ... 53
Wymagania techniczne ... 53
Definiowanie wymagań ... 54
Opis klasycznego wzorca pobierania danych ... 55
Definiowanie struktury danych ... 55
Tworzenie usługi we frameworku Angular ... 56
Pobieranie danych za pomocą metody ... 56
Wstrzykiwanie i subskrybowanie usługi w komponencie ... 57
Wyświetlanie danych w szablonie ... 57
Zarządzanie anulowaniem subskrypcji ... 58
Wzorzec reaktywny dla pobierania danych ... 61
Pobieranie danych w postaci strumieni ... 61
Definiowanie strumienia w komponencie ... 62
Używanie potoku asynchronicznego w szablonie ... 62
Podkreślenie zalet wzorca reaktywnego ... 64
Stosowanie podejścia deklaratywnego ... 64
Korzystanie ze strategii wykrywania zmian OnPush ... 65
To, co najważniejsze ... 67
Podsumowanie ... 67

Kup książkę Poleć książkę

Rozdział 5. Obsługa błędów 68
Wymagania techniczne 68
Zrozumienie anatomii obiektu obserwowalnego 69
Badanie wzorców i strategii obsługi błędów 69
Operatory obsługi błędów 70
Operator catchError 71
Operator delayWhen 78
Przykłady obsługi błędów 80
Podsumowanie 82

Rozdział 6. Łączenie strumieni 83
Wymagania techniczne 83
Definiowanie wymagań 83
Imperatywny wzorzec filtrowania danych 85
Komponent filtrujący 85
Komponent listy przepisów 86
Deklaratywny wzorzec filtrowania danych 88
Operator combineLatest 88
Filary wzorca deklaratywnego 89
Emitowanie wartości po wystąpieniu akcji 92
Podsumowanie 95

Rozdział 7. Przekształcanie strumieni 96
Wymagania techniczne 96
Definiowanie wymagania 97
Wzorzec imperatywny dla autozapisu 97
Wzorzec deklaratywny dla autozapisu 100
Obiekt obserwowalny wyższego rzędu 101
Operatory mapowania wyższego rzędu 101
Operator concatMap 102
Inne przydatne operatory mapowania wyższego rzędu 106
Operator mergeMap 106
Operator switchMap 107
Operator exhaustMap 108
Podsumowanie 109

Kup ksi k Pole ksi k

CZĘŚĆ III. Multiemisja zabierze Cię w nowe miejsca

Rozdział 8. Podstawy multiemisji 113
Wymagania techniczne 113
Różnice między multiemisją a pojedynczą emisją 113
Producent 113
Zimny obiekt obserwowalny 114
Gorący obiekt obserwowalny 115
Tematy RxJS 117
Subject 118
ReplaySubject 119
BehaviourSubject 120
Operatory multiemisji 121
Podsumowanie 122

Rozdział 9. Buforowanie strumieni 123
Wymagania techniczne 123
Definiowanie wymagania 123
Użycie wzorca reaktywnego do buforowania strumieni 125
Zalecany wzorzec RxJS 7 do buforowania strumieni 129
Przypadki użycia buforowania strumieni 130
Podsumowanie 132

Rozdział 10. Udostępnianie danych między komponentami 133
Wymagania techniczne 133
Definiowanie wymagania 134
Badanie reaktywnego wzorca udostępniania danych 134
Krok pierwszy — tworzenie wspólnej usługi 135
Krok drugi — aktualizacja ostatnio wybranego przepisu 136
Krok trzeci — konsumowanie ostatniego wybranego przepisu 137
Przedstawienie innych sposobów udostępniania danych 139
Podsumowanie 139

Rozdział 11. Operacje zbiorcze 140
Wymagania techniczne 140
Definiowanie wymagania 140

Reaktywny wzorzec dla operacji zbiorczych 142
Operator forkJoin 142
Wzorzec w akcji 144
Reaktywny wzorzec śledzenia postępów 147
Podsumowanie 148

Rozdział 12. Przetwarzanie aktualizacji w czasie rzeczywistym 149
Wymagania techniczne 149
Definiowanie wymagania 149
Reaktywny wzorzec konsumowania wiadomości w czasie rzeczywistym 151
Zachowanie WebSocketSubject 151
Zarządzanie połączeniem 154
Implementacja wzorca 155
Reaktywny wzorzec obsługi ponownego połączenia 158
Ponawianie próby ponownego połączenia 160
Podsumowanie 161

CZĘŚĆ IV. Ostatnie szlify

Wymagania techniczne 165
Wzorzec subskrypcji i zapewnienia 166
Wzorzec testowania marmurkowego 169
Zrozumienie składni 169
Implementacja testów marmurkowych 170
Testowanie strumieni przy użyciu modułu HttpClientTestingModule 175
Podsumowanie 177

Skorowidz 179

Kup ksi k
Pole ksi k

Pobieranie danych jako strumieni

Rozdział 4

Dane są najważniejsze! Sposób zarządzania danymi aplikacji ma ogromny wpływ na wydajność interfejsu użytkownika, a także wrażenia użytkownika (UX). Według mnie obecnie wrażenia użytkownika i wydajne interfejsy nie są już tylko opcją. W rzeczywistości są to kluczowe wyznaczniki satysfakcji użytkowników. Poza tym efektywne zarządzanie danymi pozwala na optymalizację kodu i poprawia jego przejrzystość, co w konsekwencji minimalizuje nakłady na utrzymanie i koszty wprowadzania zmian.

Jak więc efektywnie zarządzać naszymi danymi? Cóż, tym właśnie będziemy się zajmować w tym rozdziale. Istnieje kilka reaktywnych wzorców, które przydadzą się w wielu zastosowaniach, a zaczniemy od zbadania wzorca, który pomoże nam obsłużyć częsty i niezwykle istotny przypadek, czyli wyświetlanie wartości otrzymanych z REST API, aby użytkownicy mogli je przeczytać i na nich pracować.

Zaczniemy od wyjaśnienia wymagania, które zamierzamy zaimplementować w naszej aplikacji. Następnie użyjemy klasycznego wzorca pobierania danych i pokażemy różne podejścia, których można użyć do zarządzania rezygnacjami z subskrypcji, oraz wszystkie ważne koncepcje techniczne, które są z tym związane. Potem wyjaśnimy reaktywny wzorzec pobierania danych. Na koniec podkreślimy, w czym wzorzec reaktywny jest lepszy od klasycznego podejścia. Zaczynajmy!

W tym rozdziale omówimy następujące tematy:

- Definiowanie wymagań.
- Opis klasycznego wzorca pobierania danych.
- Opis reaktywnego wzorca pobierania danych.
- Podkreślenie zalet wzorca reaktywnego.

Wymagania techniczne

W tym rozdziale zakładamy, że masz podstawową wiedzę na temat klienta HTTP.

Kup ksi k Pole ksi k

Będziemy stosować symulowane REST API przy użyciu JSON Server, który może być rozwinięty do postaci w pełni działającej aplikacji serwerowej z REST API. Nie będziemy się uczyć, jak korzystać z JSON Server, ale jeśli chcesz dowiedzieć się więcej na ten temat, możesz znaleźć dodatkowe informacje pod adresem *https://github.com/typicode/json-server*.

Kod źródłowy projektu dla tego rozdziału znajdziesz w archiwum dostępnym pod adresem *https://ftp.helion.pl/przyklady/jawdwz.zip*.

Projekt składa się z dwóch folderów:

- *recipes-book-api*: zawiera uproszczony serwer;
- *recipes-book-front*: zawiera aplikację frontendową, która została utworzona z wykorzystaniem Angulara 12 i RxJS7, aby pomóc nam szybko zbudować piękne komponenty interfejsu użytkownika; sprawdź też wymagania techniczne dotyczące poprzedniego rozdziału.

Projekt jest zgodny ze standardowymi wytycznymi stylu Angulara, które można znaleźć pod adresem *https://angular.io/guide/styleguide*.

Aby uruchomić serwer, należy wykonać następujące polecenie w folderze *recipes-book-api*:

```
npm run server:start
```

Serwer zostanie uruchomiony pod adresem *http://localhost:3001*.

W dalszej kolejności należy uruchomić frontend, wykonując następujące polecenie w folderze *recipes-book-front*:

```
ng serve --proxy-config proxy.config.json
```

Możesz dowiedzieć się więcej o parametrze `--proxy-config` pod adresem *https://angular.io/guide/build#proxying-to-a-backend-server.*

Definiowanie wymagań

Najpierw zdefiniujmy wymaganie, które zamierzamy zrealizować w sposób reaktywny. Aby dobrze zdefiniować wymagania, powinniśmy odpowiedzieć sobie na pytanie, co chcemy robić, lub innymi słowy, jak powinna zachowywać się aplikacja. Cóż, chcemy wyświetlić listę przepisów na stronie głównej.

Komponentem odpowiedzialnym za wyświetlanie listy przepisów jest `RecipesList` ↪`Component`. Wyświetlimy więc ten komponent na stronie głównej. Zamierzamy stopniowo implementować historyjkę użytkownika opisaną w punkcie „Widok pierwszy — strona domowa” w rozdziale 3., „Przewodnik po aplikacji”.

Kup ksi k Pole ksi k

Oczywiście musimy najpierw pobrać listę przepisów, aby móc ją wyświetlić użytkownikowi. Tak więc lista przepisów to pierwsze *dane*, które musimy pobrać. Są one dostępne na zdalnym serwerze z RESTful API. Zamierzamy użyć żądania `GET` i wywołać endpoint *api/recipes* zaimplementowany w naszym backendzie *recipes-book-api*, który uruchomiliśmy w podrozdziale „Wymagania techniczne".

W kolejnych sekcjach przyjrzymy się zarówno klasycznym, jak i zalecanym wzorcom pobierania danych.

Opis klasycznego wzorca pobierania danych

Zacznijmy od opisania klasycznego wzorca zbierania naszych danych. W naszym scenariuszu żądane dane to lista przepisów.

Definiowanie struktury danych

Przede wszystkim musimy zdefiniować strukturę naszych danych, abyśmy mogli dla nich użyć silnego typowania.

Możemy zastosować Angular CLI do wygenerowania modelu `Recipe` w folderze *src/app/core/model*:

```
$ ng g i Recipe
```

Dla zachowania konwencji zmieńmy nazwę generowanego pliku *recipe.ts* na *recipe.model.ts*. Następnie zdefiniujemy interfejs za pomocą potrzebnych parametrów:

```
export interface Recipe {
  id: number;
  title: string;
  ingredients: string;
  tags?: string;
  imageUrl: string;
  cookingTime?: number;
  prepTime?: number;
  yield: number;
  steps?: string;
  rating: number;
}
```

Definiując interfejs `Recipe`, którego będziemy używać, wypisujemy kolejno nazwę i typ właściwości. Szczegółowy opis każdej właściwości znajduje się w punkcie „Widok drugi — interfejs dodawania nowego przepisu" w rozdziale 3., „Przewodnik po aplikacji".

Kup ksi k Pole ksi k

W przypadku właściwości opcjonalnych umieściliśmy znak zapytania (`?`) tuż przed nazwą właściwości.

Tworzenie usługi we frameworku Angular

Następnym krokiem jest utworzenie usługi Angulara o nazwie `RecipesService`, która będzie odpowiedzialna za zarządzanie wszystkimi operacjami związanymi z przepisem. Ta usługa enkapsuluje **operacje tworzenia**, **odczytu**, **aktualizacji** i **usuwania** (**CRUD** — ***Create, Read, Update, and Delete***) i udostępnia je dla różnych komponentów interfejsu użytkownika. W tym rozdziale zaimplementujemy tylko operację odczytu.

A dlaczego robimy to w ten sposób? Cóż, przede wszystkim dlatego, by zwiększyć modularność i zapewnić możliwość ponownego wykorzystania usługi w innych komponentach.

Użyliśmy Angulara CLI do wygenerowania usługi w folderze *core/services*:

```
$ ng g s Recipes
```

Pobieranie danych za pomocą metody

Wstrzykniemy usługę `HttpClient` i zdefiniujemy metodę pobierania danych. Usługa będzie wyglądać tak:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Recipe } from '../model/recipe.model';
import { environment } from 'src/environments/environment';
const BASE_PATH = environment.basePath

@Injectable({
  providedIn: 'root'
})
export class RecipesService {

constructor(private http: HttpClient) { }

getRecipes(): Observable<Recipe[]> {
    return this.http.get<Recipe[]>('${BASE_PATH}/recipes');
  }
}
```

Wyjaśnijmy teraz, co dzieje się w tej usłudze. To nic nadzwyczajnego!

Mamy metodę `getRecipes()`, która pobiera listę przepisów przez HTTP i zwraca silnie typowaną odpowiedź HTTP: `Observable<Recipe[]>`. Jest to powiadomienie obserwowalne, które reprezentuje strumień danych, jaki zostanie utworzony po wysłaniu

żądania HTTP `GET`. Zgodnie z dobrymi praktykami `BASE_PATH` jest zdefiniowana w pliku zewnętrznym *environment.ts*, ponieważ w wielu przypadkach zależy ona od środowiska.

Wstrzykiwanie i subskrybowanie usługi w komponencie

Następnie wstrzykniemy usługę do komponentu `RecipesListComponent`, który odpowiada za wyświetlenie listy przepisów oraz wywołanie metody `getRecipes()` w `ngOnInit()` (gdy komponent został zainicjowany). Wykonujemy operację odczytu z serwera API.

Aby dane zostały wyemitowane, musimy subskrybować obiekt obserwowalny zwrócony z metody `getRecipes()`. Następnie wiążemy dane z lokalną właściwością utworzoną w naszym komponencie; w naszym przypadku jest to tablica przepisów.

Kod komponentu będzie miał taką postać:

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { Recipe } from '../core/model/recipe.model';
import { RecipesService } from '../core/services/recipes.service';


@Component({
 selector: 'app-recipes-list',
 templateUrl: './recipes-list.component.html',
 styleUrls: ['./recipes-list.component.css']
})
export class RecipesListComponent implements OnInit {

 recipes!: Recipe[];
 constructor(private service: RecipesService) { }

 ngOnInit(): void {
  this.service.getRecipes().subscribe(result => {
   this.recipes = result;
  });
 }
}
```

Teraz, gdy pobraliśmy dane i zapisaliśmy je lokalnie w polu komponentu, zobaczmy, jak wyświetlać te informacje w interfejsie użytkownika.

Wyświetlanie danych w szablonie

Na koniec, aby wyświetlić listę przepisów w naszym interfejsie użytkownika, możemy użyć pola `recipes` (które jest dostępne w komponencie) w naszym szablonie HTML.

Kup ksi k Pole ksi k

W naszym przypadku do wyświetlania listy przepisów jako kart w układzie siatki używamy komponentu `DataView` z pakietu PrimeNG. Więcej informacji można znaleźć na stronie *https://primeng.org/dataview*.

Oczywiście naszym celem nie jest skupienie się na kodzie szablonu, ale działanie na danych. Jak widać, w poniższym przykładzie przekazaliśmy tablicę przepisów jako `value` do komponentu `dataView`. Jeżeli nie chcesz używać zewnętrznych zależności, to możesz także użyć dyrektyw strukturalnych i wykorzystać czysty HTML:

```
<div class="card">
  <p-dataView #dv [value]="recipes" [paginator]="true"
    [rows]="9" filterBy="name" layout="grid">
  /** Tu może się znajdować dodatkowy kod **/
  </p-dataView>
</div>
```

Jest to podstawowy wzorzec pobierania danych, który poznaliśmy na początku nauki Angulara. Mogłeś(-łaś) więc już go widzieć.

Zostało tylko jedno! Trzeba obsłużyć anulowanie subskrypcji obiektu obserwowalnego, ponieważ ten kod zarządza subskrypcjami ręcznie. W przeciwnym razie po zniszczeniu komponentu obiekt obserwowalny nie zostanie zakończony, a odniesienie w pamięci nie zostanie zwolnione, co spowoduje wycieki pamięci.

Dlatego zawsze powinieneś (powinnaś) zachować ostrożność, kiedy ręcznie subskrybujesz obiekty obserwowalne wewnątrz komponentów Angulara.

Zarządzanie anulowaniem subskrypcji

Istnieją dwa powszechnie stosowane sposoby zarządzania anulowaniem subskrypcji: wzorzec imperatywny oraz wzorzec deklaratywny wraz ze wzorcami reaktywnymi. Przyjrzyjmy się szczegółowo obu tym wzorcom.

Imperatywne zarządzanie anulowaniem subskrypcji

Imperatywne anulowanie subskrypcji oznacza, że ręcznie wywołujemy metodę `unsubscribe()` na obiekcie subskrypcji, którym sami zarządzamy. Poniższy fragment kodu ilustruje to. Po prostu przechowujemy subskrypcję w zmiennej o nazwie `subscription` i anulujemy subskrypcję w metodzie cyklu życia Angulara `ngOnDestroy()`:

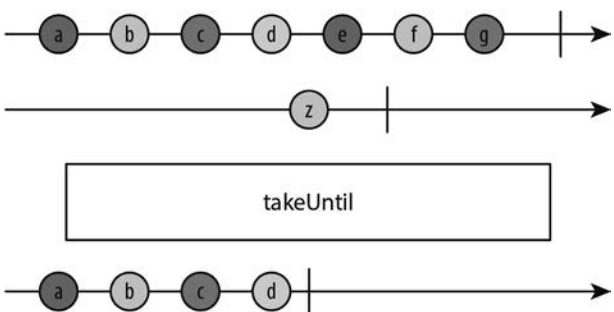
```
export class RecipesListComponent implements OnInit, OnDestroy {
    recipes!: Recipe[];
    subscription!: Subscription;
    constructor(private service: RecipesService) {}

    ngOnInit(): void {
        this.subscription = this.service.getRecipes().subscribe(result => {
```

```
            this.recipes = result;
        });
    }
    ngOnDestroy(): void {
        this.subscription.unsubscribe();
    }
}
```

Cóż, działa dobrze, ale nie jest to zalecany wzorzec. Są lepsze sposoby użycia mocy RxJS.

Deklaratywne zarządzanie anulowaniem subskrypcji

Drugi sposób jest o wiele bardziej deklaratywny i bardziej przejrzysty — używa się w nim operatora `takeUntil` RxJS. Zanim zagłębimy się w ten wzorzec, dowiedzmy się, jaką pełni rolę `takeUntil`, a nie ma lepszego sposobu na zrozumienie tego niż diagram marmurkowy (rysunek 4.1).



Rysunek 4.1. Diagram marmurkowy operatora takeUntil

Operator `takeUntil()` pobiera wartości ze źródła obserwowalnego (pierwsza oś czasu) do chwili, kiedy obiekt obserwowalny, który jest podany jako dane wejściowe (druga oś czasu), wyemituje wartość. Na poprzednim diagramie marmurkowym obserwowalne źródło emitowało wartości *a*, *b*, *c* i *d*. Tak więc `takeUntil()`wyemituje je w tej kolejności. Następnie obiekt obserwowalny wyemituje *z*, a zatem `takeUntil()` przestanie emitować wartości i zostanie zakończone.

Operator `takeUntil()` pomoże nam utrzymać subskrypcję przez zdefiniowany przez nas okres. Chcemy, aby był on aktywny, dopóki komponent nie zostanie zniszczony, więc stworzymy temat RxJS, który wyemituje wartość, gdy komponent zostanie zniszczony. Następnie przekażemy ten temat do operatora `takeUntil()` jako dane wejściowe:

```
recipes!: Recipe[];
destroy$ = new Subject < void > ();
constructor(private service: RecipesService) {}
```

```
ngOnInit(): void {
    this.service.getRecipes().pipe(
    takeUntil(this.destroy$)).subscribe(result => {
        this.recipes = result;
    });
}

ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
}
```

Uwaga

Znak *$* to nieformalna konwencja używana do wskazania, że zmienna jest obiektem. Możesz przeczytać więcej na ten temat na stronie internetowej Angulara pod adresem *https://angular.io/guide/rx-library#naming-conventions-for-observables*.

I to wszystko! Skończyliśmy.

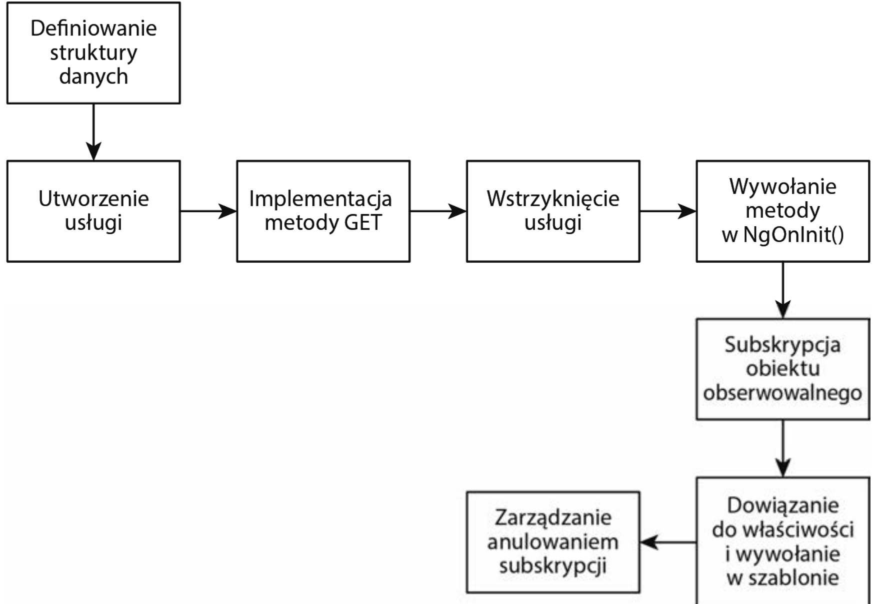
Pierwszą rzeczą, którą możesz zauważyć, jest to, że napisaliśmy tu mniej kodu niż w przypadku pierwszego podejścia. Ponadto gdy wywołamy funkcję `unsubscribe()` na zwróconym obiekcie subskrypcji (pierwszy sposób), nie ma możliwości powiadomienia nas o anulowaniu subskrypcji. Jednak przy użyciu funkcji `takeUntil()` zostaniesz powiadomiony(-na), że obiekt obserwowalny został zakończony przez metodę obsługującą zakończenie.

Dodatkowo możesz skorzystać z usług innych operatorów, które zajmą się zarządzaniem rezygnacją z subskrypcji w bardziej reaktywny sposób. Na przykład spójrz na następujące elementy:

- `take(`*x*`)`: emituje *x* wartości, a następnie jest kończony; trzeba jednak pamiętać, że przy wolnych połączeniach sieciowych, jeżeli element numer *x* nie został wyemitowany, to musisz anulować subskrypcję;
- `first()`: funkcja emituje pierwszą wartość, a następnie kończy się;
- `last()`: funkcja ta emituje ostatnią wartość, a następnie kończy się.

To był klasyczny wzorzec, którego nauczyliśmy się na początku i jest on względnie poprawny. Podsumowując, rysunek 4.2 przedstawia wszystkie kroki, przez które przeszliśmy.

Kup ksi k Pole ksi k



Rysunek 4.2. Przepływ we wzorcu klasycznym

Możemy jednak użyć innego wzorca, który jest znacznie bardziej **deklaratywny** i **reaktywny** i ma wiele zalet. Odkryjmy go!

Wzorzec reaktywny dla pobierania danych

Ideą tego reaktywnego wzorca jest zachowanie i wykorzystanie obiektu obserwowalnego jako strumienia w całej aplikacji. Nie martw się, zaraz wszystko Ci się rozjaśni.

Pobieranie danych w postaci strumieni

Zamiast definiować metodę pobierającą nasze dane, zadeklarujemy zmienną wewnątrz naszej usługi:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Recipe } from '../model/recipe.model';
import { environment } from 'src/environments/environment';
const BASE_PATH = environment.basePath
```

```
@Injectable({
    providedIn: 'root'
})
export class RecipesService {
    recipes$ = this.http.get < Recipe[] > (
     '${BASE_PATH}/recipes');
    constructor(private http: HttpClient) {}
}
```

W tym przypadku deklarujemy zmienną `recipes$` jako wynik HTTP `GET`, która może być zarówno obiektem obserwowalnym, jak i strumieniem danych.

Definiowanie strumienia w komponencie

Teraz w `RecipesListComponent` zrobimy to samo co we wzorcu klasycznym, czyli zadeklarujemy zmienną przechowującą strumień zwrócony z naszego serwisu. Tym razem zmienna jest obiektem obserwowalnym:

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { RecipesService } from '../core/services/recipes.service';

@Component({
    selector: 'app-recipes-list',
    templateUrl: './recipes-list.component.html',
    styleUrls: ['./recipes-list.component.css']
})
export class RecipesListComponent implements OnInit {
    recipes$ = this.service.recipes$;
    constructor(private service: RecipesService) {}

    ngOnInit(): void {
    }
}
```

Ale poczekaj! Musimy przecież subskrybować, aby otrzymywać emitowane dane, prawda? Jak najbardziej tak.

Używanie potoku asynchronicznego w szablonie

Nie będziemy subskrybować ręcznie; zamiast tego użyjemy **potoku asynchronicznego** (ang. *async pipe*).

Potok asynchroniczny ułatwia renderowanie wartości emitowanych z elementu obserwowalnego. Przede wszystkim automatycznie subskrybuje obserwowalne wejście. Następnie zwraca ostatnią wyemitowaną wartość. A co najlepsze, gdy komponent zostanie zniszczony, automatycznie anuluje subskrypcję, by uniknąć potencjalnych wycieków pamięci.

Oznacza to, że nie ma potrzeby czyszczenia żadnych subskrypcji, gdy komponent zostanie zniszczony. I to jest niesamowite. Za każdym razem, gdy emitowana jest nowa wartość, komponent jest oznaczany do sprawdzenia pod kątem zmian.

Tak więc w szablonie wiążemy dane z obiektem obserwowalnym za pomocą potoku asynchronicznego. Tutaj `as recipes` opisuje zmienną tablicową, do której emitowane są wartości. Możemy jej użyć w szablonie w ten sposób:

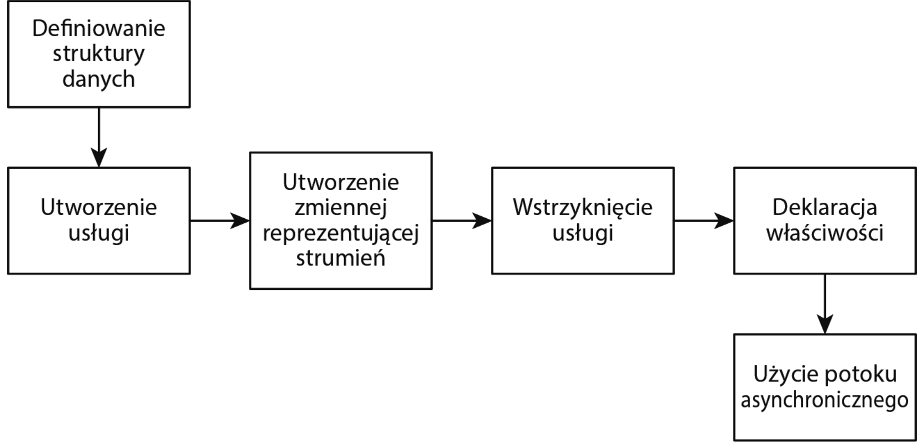
```
<div *ngIf="recipes$ | async as recipes" class="card">
    <p-dataView #dv [value]="recipes" [paginator]="true"
      [rows]="9" filterBy="name" layout="grid">
    /** Tu może się znajdować dodatkowy kod **/
    </p-dataView>
</div>
```

W ten sposób nie musimy podpinać się do metody cyklu życia `ngOnInit()`. Nie będziemy subskrybować obiektu obserwowalnego w `ngOnInit()` i nie będziemy anulować subskrypcji w `ngOnDestroy()`, tak jak to robiliśmy w podejściu klasycznym. Po prostu ustawiamy właściwość lokalną w naszym komponencie i to wystarczy.

Nie trzeba więc ręcznie subskrybować ani anulować subskrypcji.

Cały kod szablonu HTML jest dostępny w archiwum pod adresem *https://ftp.helion.pl/przyklady/jawdwz.zip.*

Podsumowując, rysunek 4.3 przedstawia wszystkie kroki, przez które przeszliśmy.



Rysunek 4.3 Przepływ we wzorcu reaktywnym

Teraz, gdy został przedstawiony wzorzec reaktywny w działaniu, przejdźmy do następnego podrozdziału i poznajmy jego zalety.

Podkreślenie zalet wzorca reaktywnego

Pewnie już znasz pierwszą zaletę: nie musimy ręcznie zarządzać subskrypcją i jej anulowaniem, a co to za ulga! Ale jest też wiele innych zalet; nie chodzi tylko o potok asynchroniczny. Spójrzmy szczegółowo na inne zalety.

Stosowanie podejścia deklaratywnego

Zwróćmy uwagę na fakt, że nie używamy jawnie metody `subscribe()`. Co jest w niej złego? Cóż, subskrybowanie strumienia w naszym komponencie oznacza, że pozwalamy na to, że kod imperatywny przecieka do naszego kodu funkcyjnego i reaktywnego. Samo użycie obiektów obserwowalnych z RxJS nie sprawia, że Twój kod jest reaktywny i deklaratywny. Jednak co właściwie oznacza deklaratywny?

Cóż, najpierw wyjaśnimy kilka kluczowych terminów:

- **Deklaratywny** odnosi się do użycia zadeklarowanych funkcji do wykonywania akcji. Polegasz na czystych funkcjach, które mogą zdefiniować przepływ zdarzeń. Używając RxJS, możesz to zobaczyć w postaci obiektów obserwowalnych i operatorów.
- **Funkcja czysta** (ang. *pure function*) to funkcja, która zawsze zwraca identyczne wyniki dla tych samych danych wejściowych, bez względu na to, ile razy zostanie wywołana. Innymi słowy, zawsze w sposób przewidywalny da ten sam wynik.

Ale dlaczego jest to takie ważne? Cóż, należy to mieć na uwadze, ponieważ deklaratywne podejście przy wykorzystaniu operatorów RxJS i obiektów obserwowalnych ma wiele zalet, a mianowicie:

- Sprawia, że Twój kod jest czystszy i bardziej czytelny.
- Ułatwia testowanie kodu, ponieważ jest on przewidywalny.
- Umożliwia cache'owanie strumienia wyjściowego przy określonym wejściu, co zwiększa wydajność. Przyjrzymy się temu bardziej szczegółowo w rozdziale 8., „Podstawy multiemisji”, rozdziale 9., „Buforowanie strumieni” i rozdziale 10., „Udostępnianie danych między komponentami”.
- Umożliwia najlepsze wykorzystanie operatorów RxJS oraz przekształcanie i łączenie strumieni pochodzących z różnych usług lub z tej samej usługi. To jest to, co zrobimy w rozdziale 7., „Przekształcanie strumieni”.
- Pomaga łatwo reagować na działania użytkownika w celu wykonania akcji.

Kup książkę Poleć książkę

Tak więc bardziej deklaratywne oznacza bardziej reaktywne. Zachowaj jednak ostrożność. To nie znaczy, że nie możesz w ogóle wywołać metody `subscribe()`. W niektórych sytuacjach nie da się inaczej sprawić, by obiekt obserwowalny zaczął nadawać. Ale spróbuj zadać sobie pytania: Czy naprawdę muszę tu subskrybować? Czy zamiast tego mogę połączyć wiele strumieni lub użyć operatorów RxJS, aby osiągnąć to, czego potrzebuję, bez konieczności subskrybowania? Poza przypadkami, w których jest to nieuniknione, nigdy nie używaj metody `subscribe()`.

Przejdźmy teraz do zalet związanych z wydajnością.

Korzystanie ze strategii wykrywania zmian OnPush

Inną naprawdę fajną rzeczą jest to, że możemy użyć strategii `changeDetection`, `OnPush`.

Wykrywanie zmian to jedna z potężnych funkcji Angulara. Chodzi o wykrycie, kiedy zachodzi zmiana danych komponentu, a następnie automatyczne ponowne renderowanie widoku lub aktualizację drzewa DOM, aby odzwierciedlić tę zmianę. Domyślna strategia „sprawdzaj zawsze” oznacza, że zawsze kiedy jakiekolwiek dane zostaną zmienione, Angular uruchomi mechanizm wykrywający zmiany, aby zaktualizować drzewo DOM. Dzieje się to więc automatycznie, dopóki nie zostanie jawnie dezaktywowane.

W strategii `OnPush` Angular uruchamia detektor zmian tylko wtedy, gdy występuje któryś z warunków:

- *Warunek 1.*: Zmienia się odwołanie do właściwości `@Input` komponentu (pamiętaj, że gdy obiekt będący właściwością wejściową jest zmieniany bezpośrednio, wówczas odwołanie do obiektu nie ulegnie zmianie, w związku z czym detektor zmian nie uruchomi się. W tym przypadku powinniśmy zwrócić nowe odwołanie do obiektu właściwości, aby wywołać wykrycie zmian).
- *Warunek 2.*: Procedura obsługi zdarzeń (ang. *event handler*) zostanie wywołana lub wyemituje wartość.
- *Warunek 3.*: Obiekt obserwowalny powiązany przez potok asynchroniczny emituje nową wartość.

Dlatego stosowanie strategii `changeDetection OnPush` minimalizuje zakres wykrywania zmian, tak by sprawdzać tylko wymienione wcześniej przypadki w celu ponownego renderowania komponentów. Ta strategia ma zastosowanie do wszystkich dyrektyw podrzędnych i nie można jej przesłonić.

W naszym scenariuszu detektor zmian jest potrzebny tylko wtedy, gdy otrzymamy nową wartość; w przeciwnym razie otrzymujemy bezużyteczne aktualizacje. Zatem

nasz scenariusz spełnia *Warunek 3*. Dobra wiadomość jest taka, że możemy użyć strategii wykrywania zmian `onPush` w następujący sposób:

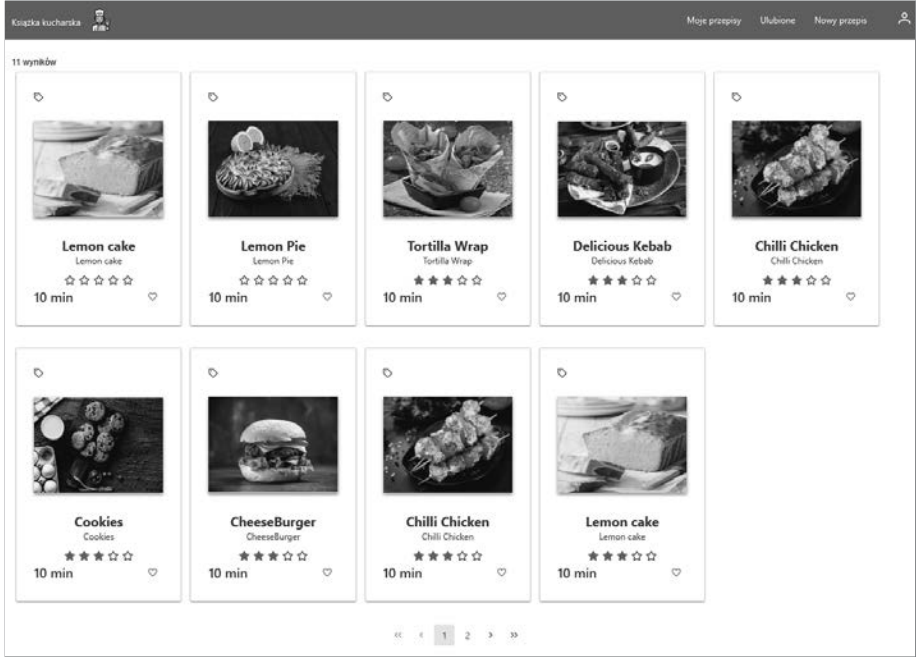
```
import { ChangeDetectionStrategy, Component} from '@angular/core';

@Component({
    selector: 'app-recipes-list',
    templateUrl: './recipes-list.component.html',
    styleUrls: ['./recipes-list.component.scss'],
    changeDetection: ChangeDetectionStrategy.OnPush
})
```

Tak więc, jeśli będziemy pamiętać o używaniu asynchronicznego potoku tak często, jak to możliwe, dostrzeżemy kilka zalet:

- Ułatwimy przejście z domyślnej strategii wykrywania zmian na `OnPush`, jeśli zajdzie taka potrzeba.
- Będziemy mieć mniej cykli wykrywania zmian przy użyciu `OnPush`.

Korzystanie z potoku asynchronicznego pomoże Ci uzyskać wydajny interfejs użytkownika, co będzie bardzo istotne, gdy nasz widok będzie wykonywał wiele zadań.

Na rysunku 4.4. widzimy nasz interfejs użytkownika.



Rysunek 4.4. Widok listy przepisów

Kup ksi k Pole ksi k

To, co najważniejsze

W skrócie, użycie reaktywnego wzorca pobierania danych ma następujące zalety:

- poprawia wydajność Twojej aplikacji,
- poprawia strategię wykrywania zmian,
- zwiększa przejrzystość i czytelność kodu,
- sprawia, że kod jest bardziej deklaratywny i bardziej reaktywny,
- ułatwia wykorzystanie operatorów RxJS,
- ułatwia reagowanie na działania użytkownika.

Podsumowanie

W tym rozdziale przyjrzeliśmy się klasycznemu i reaktywnemu wzorcowi pobierania danych. Nauczyliśmy się imperatywnego podejścia do zarządzania anulowaniem subskrypcji wraz ze wzorcem reaktywnym. Wyjaśniliśmy kilka przydatnych operatorów RxJS, a także rzuciliśmy światło na zalety korzystania ze wzorca reaktywnego i dowiedzieliśmy się o wszystkich technicznych aspektach z tym związanych.

Skoro już pobraliśmy nasze dane w postaci strumieni RxJS, przejdźmy do następnych rozdziałów, gdzie dzięki temu będziemy mogli reagować na działania użytkownika za pomocą strumieni RxJS i w konsekwencji zbudujemy naszą aplikację *Książka kucharska* w sposób reaktywny. W następnym rozdziale skupimy się na wzorcach reaktywnych służących do obsługi błędów i różnych dostępnych w tym celu strategiach.

Kup ksi k Pole ksi k

Skorowidz

A

akcje
  emitowanie wartości, 92
aktualizacje w czasie rzeczywistym, 149
anatomia obiektu obserwowalnego, 69
Angular, 22
  tworzenie usługi, 56
anulowanie subskrypcji, 58
API, 41
architektura aplikacji, 48
AsyncPipe, 29
autozapis, 96, 97, 100

B

BehaviourSubject, 120
biblioteka PrimeNG, 43
błędy, 40, 68, 80, 153
Bootstrap, 43
buforowanie strumieni, 123
  przypadki użycia, 130
  wzorzec reaktywny, 125
  wzorzec RxJS 7, 129

C

CRUD, 56

D

deklaratywny, 64
diagram marmurkowy, marble diagram, 30, 59

E

elementy obserwowalne, observables, 21
emiter zdarzeń, 28

F

filary
  programowania reaktywnego, 20
  wzorca deklaratywnego, 89
filtrowanie danych
  deklaratywne, 88
  imperatywne, 85
formularze reaktywne, 28
framework Angular, 22
funkcja, 33
  czysta, pure function, 64
  toPromise(), 36

H

historyjki użytkowników, 44
HttpClientTestingModule
  testowanie strumieni, 175

I

implementacja
  testów marmurkowych, 170
  wzorca, 155
interfejs, 44–48

J

język TypeScript, 36

K

klasa
- ActivatedRoute, 26, 27
- Component, 28
- EventEmitter, 28
- FormControl, 28
- RecipesService, 80

klient, 23
komponenty aplikacji, 49
- definiowanie strumienia, 62
- filtrowanie, 85
- lista przepisów, 86
- subskrybowanie usługi, 57

komunikacja WebSocket, 150

Ł

łączenie strumieni, 83, 91

M

metoda
- firstValueFrom(), 38
- lastValueFrom(), 39

metody
- pobieranie danych, 56

moduł
- HttpClientTestingModule, 175
- klienta HTTP, 23
- routera, 24

multiemisja, multicasting, 113

N

narzędzia Bootstrap, 43
nasłuchiwanie wiadomości, 152

O

obiekty obserwowalne, 69
- gorące, 115
- powiadamiające, 76
- testowanie, 165
- wyższego rzędu, 101
- zimne, 114

obietnice, promises, 37
obserwatorzy, observers, 22
obsługa
- błędów, 68, 80, 153
- połączenia, 158

odpytywanie, polling, 125
operacje zbiorcze, 140, 142
operator
- catchError:, 71, 73
- combineLatest, 88
- concatMap, 102
- delayWhen, 78
- exhaustMap, 108
- forkJoin, 142
- map:, 72
- mergeMap, 106
- retry, 74
- retryWhen, 75
- switchMap, 107
- takeUntil, 59

operatory
- mapowania wyższego rzędu, 101, 106
- multiemisji, 121
- obsługi błędów, 70

otwarcie połączenia, 152

P

pobieranie danych, 53
- w postaci strumieni, 61
- wzorzec, 55
- wzorzec reaktywny, 61
- za pomocą metody, 56

połączenia, 154
- otwieranie, 152
- ponowne, 160
- zamykanie, 153

potok asynchroniczny, 29, 62
producent, 113
programowanie reaktywne, 20
protokół Websocket, 150

Kup ksi k Pole ksi k

R

ReplaySubject, 119
router, 24
routing, 26
rozmiar pakietu, 34
RxJS 7, 21, 22, 33

S

stan błędu, 69
status ukończenia, 69
strategia
  obsługi błędów, 69
  ponawiania, 74
  ponownego rzutu, 73
  wykrywania zmian OnPush, 65
  zastępowania, 71
struktura danych, 55
strumienie, 62
  buforowanie, 123
  łączenie, 83, 91
  pobieranie danych, 53
  przekształcanie, 96
  testowanie, 175
strumień
  akcji, 90
  danych, 90
Subject, 118
subskrybowanie usługi, 57
szablon
  potok asynchroniczny, 62
  wyświetlanie danych, 57

T

technika
  ściągania, pull, 150
  wypychania, push, 150
tematy RxJS, 117
testowanie
  marmurkowe, 169, 170
  obiektów obserwowalnych RxJS, 165
  strumieni, 175
tworzenie
  strumienia akcji, 90
  usługi, 56
  WebSocketSubject, 151
TypeScript, 36

U

udostępnianie danych, 134, 139
usługi
  wstrzykiwanie, 57

W

WebSocket, 150
widoki aplikacji, 44–48
  „Moje przepisy”, 46
  „Ulubione”, 47
  dodawania nowego przepisu, 45
  edycji przepisu, 47
  strony głównej, 44
  szczegółów przepisu, 48
wstrzykiwanie usługi, 57
wypychanie wiadomości, 152
wyświetlanie danych, 57
wywołania zwrotne, callback, 69
wzorce
  reaktywne, 19, 61, 64
  deklaratywne, 89, 100
wzorzec
  buforowania strumieni, 125, 129
  dla autozapisu
    deklaratywny, 100
    imperatywny, 97
  dla operacji zbiorczych, 142
  filtrowania danych, 85, 88
  klasyczny, 60, 61
  konsumowania wiadomości, 151
  Obserwator, 21
  obsługi ponownego połączenia, 158
  pobierania danych, 55
  subskrypcji i zapewnienia, 166
  śledzenia postępów, 147
  testowania marmurkowego, 169
  udostępniania danych, 134

Kup ksi k Pole ksi k

Z

zachowanie WebSocketSubject, 151
zamykanie połączenia, 153
zarządzanie
  anulowaniem subskrypcji, 58
  połączeniem, 154
zdarzenia
  routera, 24
  w WebSocketSubject, 153

Ź

źródło obserwowalne, 76

Ż

żądanie http
  GET, 124
  POST, 99

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

Musisz zacząć myśleć w reaktywny sposób!

Angular jest frameworkiem napisanym w języku TypeScript i rozwijanym przez Google. Służy do tworzenia nowoczesnych, wydajnych aplikacji. RxJS to niezawodna biblioteka do obsługi programów asynchronicznych i opartych na zdarzeniach. Jest typem pierwszoklasowym w Angularze. Umożliwia poprawę wydajności aplikacji i jakości kodu, polepsza też doświadczenia użytkownika. Zastosowanie wzorców reaktywnych przy tworzeniu stron internetowych za pomocą Angulara jest jedną ze skuteczniejszych metod usprawniania interakcji użytkownika z aplikacjami Angulara.

Ten przewodnik zawiera wszystko, co trzeba wiedzieć o RxJS i reaktywności. Zrozumiesz znaczenie paradygmatu reaktywnego i nauczysz się korzystać z nowych funkcji RxJS 7. Poznasz różne wzorce reaktywne, a także rzeczywiste sposoby ich używania. Książka przeprowadzi Cię przez proces tworzenia kompletnej aplikacji, dzięki czemu poznasz techniki reaktywnego zarządzania danymi aplikacji. Przybliżysz sobie też różne wzorce poprawiające komfort użytkowania i jakość kodu. Dowiesz się, jak z zastosowaniem najlepszych praktyk przetestować strumienie asynchroniczne i poprawić wydajność aplikacji. W efekcie zaczniesz tworzyć aplikacje Angulara poprzez implementację wzorców reaktywnych.

W książce:

- praca z diagramem marmurkowym
- korzystanie z RxJS 7 podczas budowy i wdrażania reaktywnej aplikacji Angulara
- koncepcja strumieni (przekształcanie, łączenie, komponowanie)
- strategie testowania aplikacji RxJS
- wycieki pamięci w aplikacjach internetowych i techniki ich unikania
- multiemisja w RxJS i rozwiązywanie złożonych problemów

Lamis Chebbi jest ekspertką w zakresie technologii Google dla Angulara, a także prelegentką i trenerką. Założyła Angular Tunisia, która należy do społeczności WWCode. Od kilku lat zajmuje się Angularem. Jej pasją są również muzyka i podróże. Mówi o sobie, że jest wieczną studentką.