Wydanie IV

Poznaj Angular

Rzeczowy przewodnik po tworzeniu aplikacji webowych z użyciem frameworku Angular 15

Aristeidis Bampakos
Pablo Deeleman

Helion

‹packt›

Tytuł oryginału: Learning Angular: A no-nonsense guide to building web applications with Angular 15, 4th Edition

Tłumaczenie: Anna Mizerska

ISBN: 978-83-289-0385-2

Copyright © Packt Publishing 2023. First published in the English language under the title 'Learning Angular - Fourth Edition – (9781803240602)

Polish edition copyright © 2024 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/pozan4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/pozan4.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

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Komponenty interfejsu użytkownika

Rozdział 4

Dotychczas mieliśmy okazję spojrzeć na framework Angular z lotu ptaka. Nauczyliśmy się tworzyć nową aplikację za pomocą CLI Angulara i grupować funkcje aplikacji w moduły. Widzieliśmy, jak używać modułów Angulara i organizować naszą aplikację w wydajny i spójny sposób, dzięki któremu można łatwo skalować i testować aplikację.

Jak się dowiedzieliśmy, moduły Angulara rozszerzają nasze aplikacje tworzone w tym frameworku przez dodawanie nowych funkcjonalności. Wspomnieliśmy już, że funkcjonalność modułu jest głównie reprezentowana przez komponenty Angulara. Wydaje się, że wiemy już wszystko, by dalej odkrywać możliwości oferowane przez Angular w kwestii interaktywnych komponentów i ich sposobu komunikowania się ze sobą.

W tym rozdziale omówimy następujące tematy:

- Tworzenie naszego pierwszego komponentu.
- Interakcja z szablonem.
- Wzajemne porozumiewanie się komponentów.
- Enkapsulacja stylów CSS.
- Wybór strategii wykrywania zmian.
- Wprowadzenie komponentu cyklu życia.

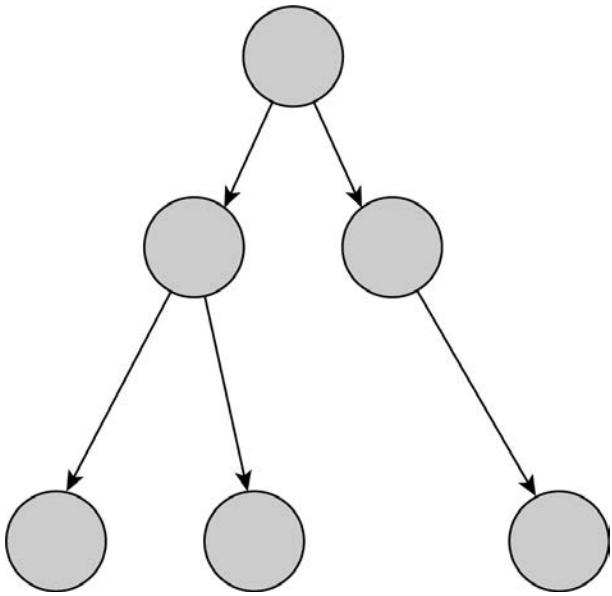
Wymagania techniczne

W tym rozdziale znajdziesz różne przykłady kodu, które pomogą Ci przejść przez koncepcję komponentów Angulara. Kod wykorzystywany w tym rozdziale znajdziesz w folderze *r04* w materiałach do pobrania dla tej książki, pod adresem: *https://ftp.helion.pl/przyklady/pozan4.zip*.

Kup książkę Poleć książkę

Tworzenie naszego pierwszego komponentu

Komponenty to podstawowy budulec aplikacji napisanej w Angularze. Sterują różnymi częściami strony internetowej zwanej **widokami**, na przykład listą produktów lub formularzem finalizacji zamówienia. Są odpowiedzialne za logikę związaną z prezentacją aplikacji napisanej w Angularze, są poukładane w formie hierarchicznego drzewa i mogą ze sobą współpracować, co pokazano na rysunku 4.1.



Rysunek 4.1. Architektura komponentów

Architektura aplikacji tworzonej w Angularze opiera się na komponentach. Każdy komponent Angulara może się komunikować i współpracować z jednym komponentem lub większą liczbą komponentów w drzewie komponentów. Jak widzimy na rysunku 4.1, komponent może być nadrzędny dla kilku komponentów, a zarazem podrzędny innemu komponentowi nadrzędnemu.

W tym podrozdziale odkryjemy następujące tematy związane z komponentami Angulara:

- Budowa komponentu Angulara.
- Rejestrowanie komponentów w modułach.
- Tworzenie **odrębnych** komponentów.

Naszą podróż zaczniemy od omówienia struktury komponentu Angulara.

Kup książkę Poleć książkę

Budowa komponentu Angulara

Jak dowiedzieliśmy się z rozdziału 1. „Budowa pierwszej aplikacji opartej na Angularze", typowa aplikacja napisana w Angularze zawiera przynajmniej jeden główny komponent składający się kilku plików. Plik TypeScript tego komponentu to *app.component.ts*.

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

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   title = 'Poznaj Angular';
}
```

Wyrażenie `import` na początku pliku służy do zaimportowania artefaktu `Component` z pakietu *@angular/core* `npm`. Artefakt `Component` to dekorator Angulara użyty w celu konfiguracji komponentu Angulara. Zawiera następujące właściwości:

- `selector` — selektor CSS mówiący Angularowi, by wczytał komponent w miejscu, w którym znajdzie odpowiadający mu znacznik w szablonie HTML. CLI Angulara domyślnie dodaje przedrostek `app`, ale możesz to ustawić inaczej za pomocą opcji `--prefix` podczas tworzenia projektu w Angularze.
- `templateUrl` — definiuje ścieżkę zewnętrznego pliku HTML zawierającego szablon HTML tego komponentu. Możesz również określić szablon w kodzie za pomocą właściwości `template`.
- `styleUrls` — definiuje listę ścieżek wskazujących na zewnętrzne pliki ze stylami CSS. Styl możesz również określić w kodzie za pomocą właściwości `styles`.

Plik TypeScript komponentu również ma klasę TypeScript o nazwie `AppComponent` zawierającą właściwość `title`. Dekorator `@Component` nad klasą jest informacją, że jest to komponent Angulara. Gdyby tego dekoratora brakowało, framework Angular traktowałby tę klasę jak zwykłą klasę TypeScriptu.

Rejestrowanie komponentów w modułach

Poza głównym komponentem aplikacji możemy utworzyć komponenty Angulara, które zapewniają modułowi Angulara określoną funkcjonalność. W poprzednim rozdziale stworzyliśmy moduł Angulara do zarządzania produktami w naszej aplikacji sklepu internetowego i użyliśmy wymyślonego komponentu listy produktów. Teraz nadszedł czas, by utworzyć ten komponent naprawdę!

Kup książkę Poleć książkę

Nowy komponent w aplikacji tworzymy za pomocą polecenia `generate` w CLI Angulara, przekazując nazwę komponentu jako parametr. Uruchom to polecenie w folderze `products`, który utworzyliśmy w poprzednim rozdziale:

```
ng generate component product-list
```

Ta komenda doda komponent `product-list` i zarejestruje go w module `products`.

Wskazówka

Gdybyśmy uruchomili polecenie `generate` w folderze *src\app*, komponent zostałby zarejestrowany w głównym module aplikacji. Chcemy tego uniknąć, gdyż nie byłoby to zgodne z regułą modułowości i zasadą wielokrotnego wykorzystania modułów Angulara.

Tworzenie komponentu Angulara to proces składający się z dwóch kroków. Polega na utworzeniu niezbędnych plików danego komponentu i zarejestrowaniu go w module Angulara. Polecenie pokazane powyżej utworzy folder *product-list* zawierający pojedyncze pliki komponentów, o których była mowa w rozdziale 1. „Budowa pierwszej aplikacji opartej na Angularze". Jednocześnie CLI Angulara zarejestruje określony komponent w module `products` przez dodanie klasy `ProductListComponent` w tablicy `declarations` w pliku *products.module.ts*.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/
↪product-list.component';

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

Uwaga

Komponenty Angulara mogą być zarejestrowane tylko w jednym module Angulara.

Gdy rejestrujemy komponent w module Angulara, nadajemy mu **kontekst kompilacji**. Komponent może znaleźć wszystko, czego potrzebuje, co musi być wczytane w tym kontekście. Jednak możemy również tworzyć komponenty, które nie istnieją w kontekście żadnego szczególnego modułu Angulara.

Kup książkę Poleć książkę

Tworzenie odrębnych komponentów

Komponent niezarejestrowany w module Angulara to odrębny moduł. Odrębne komponenty nie potrzebują kontekstu kompilacji z modułu Angulara, ponieważ samodzielnie importują artefakty Angulara, których potrzebują. Aby utworzyć samodzielny komponent za pomocą CLI Angulara, w poznanym już poleceniu `generate` ustawiamy opcję `standalone`.

```
ng generate component product --standalone
```

Plik TypeScript samodzielnego komponentu nieco różni się od pliku *product.component.ts*.

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

@Component({
   selector: 'app-product',
   standalone: true,
   imports: [CommonModule],
   templateUrl: './product.component.html',
   styleUrls: ['./product.component.css']
})
export class ProductComponent {}
```

Dekorator `@Component` zawiera następujące dodatkowe właściwości:

- `standalone` — wskazuje na to, czy komponent jest samodzielny, czy nie;
- `imports` — zawiera moduły Angulara lub inne *samodzielne* komponenty, które muszą być odpowiednio wczytane. CLI Angulara domyślnie dodaje `CommonModule` przy okazji tworzenia nowych odrębnych komponentów.

Gdy przyjrzysz się bliżej, pewnie zauważysz, że dekorator `@Component` wykonuje względem zaimportowanych artefaktów tę samą pracę co moduł Angulara. Wygląda na to, że przenieśliśmy tablicę `imports` z modułu Angulara do dekoratora komponentu. Ponadto można zauważyć, że wytworzenie komponentu nie zmieniło żadnego modułu Angulara.

Samodzielne komponenty mogą importować moduły Angulara i na odwrót. Jedyne, co musimy zrobić, to dodać samodzielny komponent do tablicy `imports` modułu, tak jakby był to moduł.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.
component';
import { ProductComponent } from './product/product.component';
```

Kup książkę Poleć książkę

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

Importowanie `ProductComponent` do modułu `ProductsModule` powoduje, że samodzielny komponent jest dostępny w całym module.

Uwaga

Odrębny komponent nie powinien być dodawany do tablicy `declarations` modułu Angulara, ponieważ w ten sposób zarejestrowalibyśmy go w tym module.

Samodzielne komponenty stanowią rewolucyjny sposób przyjęcia prostszego podejścia do budowania aplikacji, stawiającego komponent w centrum. Nauczyliśmy się tworzyć aplikacje, a poza tym zobaczyliśmy, jak tworzyć komponenty i rejestrować je w module Angulara.

Uwaga

Samodzielne komponenty są zalecane do szybkiego prototypowania, na potrzeby demo lub podczas nauki Angulara. Gdy będziesz rozwijać swoją aplikację i będziesz dodawać kolejne funkcje do swojej aplikacji, możliwe, że będziesz musiał użyć modułów Angulara w celu lepszej organizacji hierarchii komponentów.

W tym podrozdziale skupiliśmy się na klasie TypeScript komponentów Angulara, ale jak te komponenty współpracują ze swoimi szablonami HTML?

W następnym podrozdziale nauczymy się, jak na stronie wyświetlać szablon HTML komponentu Angulara. Zobaczymy także, jak używać składni szablonu Angulara w celu wprowadzenia interakcji między klasą TypeScript komponentu a jego szablonem HTML.

Interakcja z szablonem

Jak się dowiedzieliśmy, tworzenie komponentu Angulara za pomocą CLI Angulara wymaga wygenerowania zestawu plików towarzyszących. Jeden z tych pików to szablon komponentu z zawartością HTML wyświetlany na stornie. W tym podrozdziale

Kup książkę Poleć książkę

odkryjemy, jak wyświetlać i wchodzić w interakcję z szablonem przez omówienie tych tematów:

- Wczytywanie szablonu komponentu.
- Wyświetlanie danych z klasy komponentu.
- Nadawanie stylu komponentowi.
- Pobieranie danych z szablonu.

Zaczniemy naszą podróż od szablonu komponentu i odkryjemy, jak przygotować komponent do wyświetlania go na stronie internetowej.

Wczytywanie szablonu komponentu

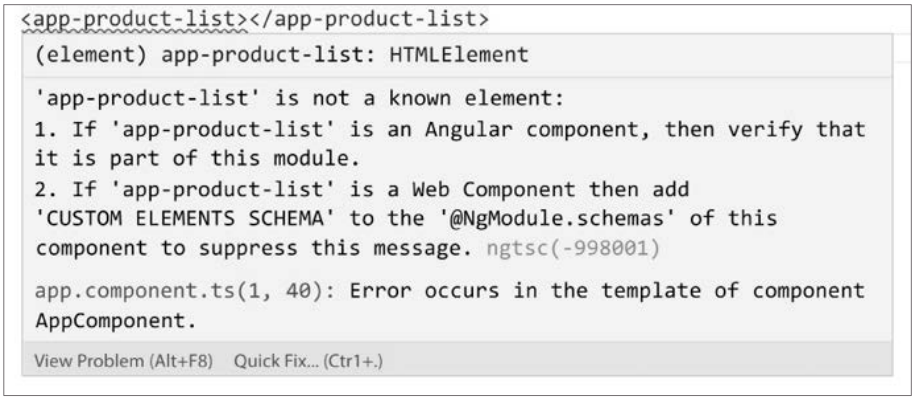
Wiemy już, że Angular używa selektora, by wczytać komponent do szablonu HTML. Typowa aplikacja napisana w Angularze wczytuje szablon z głównego komponenty, przy uruchamianiu aplikacji. Znacznik `<app-root>`, który widzieliśmy w rozdziale 1. „Budowa pierwszej aplikacji opartej na Angularze", to selektor główny komponentu aplikacji. Aby wczytać utworzony przez nas komponent, taki jak komponent listy produktów, musimy dodać jego selektor w szablonie HTML.

Na potrzeby tego scenariusza wczytamy ten komponent do szablonu głównego komponentu aplikacji.

1. Otwórz plik *app.component.html* aplikacji, nad którą aktualnie pracujemy, i zastąp jego zawartość takim kodem:

   ```
   <app-product-list></app-product-list>
   ```

   Po wpisaniu tej linii do pliku pojawi się błąd w edytorze, widoczny na rysunku 4.2.



Rysunek 4.2. Błąd szablonu

Kup książkę Poleć książkę

Ten błąd wynika z tego, że moduł `ProductsModule` nie eksponuje jeszcze listy produktów przez swoje publiczne API.

2. Otwórz plik *products.module.ts* i dodaj klasę `ProductListComponent` do tablicy `exports` dekoratora `@NgModule`.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.component';

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

3. Uruchom następujące polecenie, by uruchomić aplikację:

```
ng serve
```

Po zakończonej pomyślnie kompilacji wpisz w przeglądarce *http://localhost:4200/*, by wyświetlić komponent. Na stronie pokaże się tekst informujący, że lista produktów działa.

```
product-list works!
```

Wyświetlany tekst to zawartość komponentu szablonu znajdującego się w pliku *product-list.component.html*. Podczas tworzenia nowego komponentu CLI Angulara tworzy domyślny szablon HTML z akapitem HTML zawierającym selektor komponentu:

```
<p>product-list works!</p>
```

W kolejnych punktach zobaczymy, jak używać składni szablonu Angulara i wykonywać działania na szablonie poprzez klasę TypeScript. Zaczniemy odkrywać sposób wyświetlania dynamicznych danych zdefiniowanych w klasie TypeScript komponentu.

Wyświetlanie danych z klasy komponentu

Już spotkaliśmy się z interpolacją przy okazji wyświetlania wartości właściwości z klasy komponentu w szablonie.

```
<span>{{title}}</span>
```

Kup książkę Poleć książkę

Angular przekształca właściwość `title` na tekst i wyświetla go na ekranie. Inną metodą przeprowadzenia interpolacji jest **wiązanie właściwości** (ang. *property binding*), która w tym przypadku polega na powiązaniu właściwości `title` z właściwością `innerText` elementu `span` w HTML.

```
<span [innerText]="title"></span>
```

W tej linii przywiązujemy właściwość `title` do właściwości **obiektowego modelu dokumentu** (**DOM**, ang. *Document Object Model*) elementu, a nie atrybutu HTML, jak może się wydawać na pierwszy rzut oka. Właściwość umieszczona w nawiasach kwadratowych to **właściwość docelowa** i jest to właściwość elementu DOM, z którą chcemy powiązać daną właściwość. Zmienna po prawej stronie to **wyrażenie szablonu** i odpowiada właściwości `title` komponentu.

Wskazówka

Po otwarciu strony internetowej w przeglądarce przeglądarka przechodzi przez zawartość HTML strony i przekształca ją na strukturę drzewa, obiektowy model dokumentu (DOM). Każdy element HTML strony jest przekształcany na obiekt nazywany węzłem (ang. *node*), który przedstawia część obiektowego modelu dokumentu. Węzeł definiuje zestaw właściwości i metod reprezentujących obiekt API. Właściwość `innerText` służy do ustawienia tekstu w środku elementu HTML.

Aby lepiej zrozumieć, jak działa mechanizm szablonu, musimy najpierw zrozumieć, jak Angular współpracuje z atrybutami i właściwościami. Ten mechanizm definiuje atrybuty HTML, aby zainicjalizować właściwość DOM, a następnie wiąże dane, by współpracować bezpośrednio z tą właściwością.

Aby ustawić atrybut elementu HTML, używamy składni `attr.` w wiązaniu właściwości. Na przykład w celu ustawienia atrybutu dostępności `aria-label` elementu HTML napisalibyśmy następujący kod:

```
<p [attr.aria-label]="myText"></p>
```

W tej linii kodu właściwość `myText` jest właściwością w powiązanym komponencie Angulara. Pamiętaj, że wiązanie właściwości oddziałuje z właściwościami komponentów Angulara. Zatem jeśli chcielibyśmy ustawić wartość właściwości `innerText` bezpośrednio w kodzie HTML, napisalibyśmy wartość tekstową ujętą w apostrofy.

```
<span [innerText]="'Mój tytuł'"></span>
```

W tym przypadku przekazana do właściwości `innerText` wartość to statyczny tekst, a nie właściwość komponentu.

Wiązanie właściwości w frameworku Angular jest wygodnym sposobem wyświetlania danych i nadawania stylu.

Kup książkę Poleć książkę

Nadawanie stylu komponentowi

W aplikacji webowej style można nadawać albo za pomocą atrybutu `class`, albo `style` elementu HTML.

```
<p class="star"></p>
<p style="color: greenyellow"></p>
```

Framework Angular zapewnia dwa rodzaje wiązania właściwości, by ustawiać dynamicznie atrybut `class` i `style`: **wiązanie klasy** (ang. *class binding*) i **wiązanie stylu** (ang. *style binding*). Możemy dodać jedną klasę do elementu HTML za pomocą tej składni:

```
<p [class.star]="isLiked"></p>
```

W tym kodzie zostanie dodana klasa `star` do elementu paragrafu, gdy wyrażenie `isLiked` jest *prawdziwe*. W przeciwnym razie ta klasa zostanie usunięta z tego elementu. Jeśli chcemy dodać kilka klas jednocześnie, możemy napisać taki kod:

```
<p [class]="currentClasses"></p>
```

Zmienna `currentClasses` jest właściwością komponentu. Wyrażenie używane w wiązaniu klasy może przyjąć jedną z następujących wartości:

- Nazwa klasy w postaci łańcucha znaków, gdzie wyrazy są oddzielone spacją, na przykład `'star active'`.
- Nazwa klasy w postaci obiektu z kluczami i wartościami w postaci warunków zwracających wartość typu `boolean` dla każdego klucza. Klasa jest dodawana do elementu, gdy wartość klucza o tej samej nazwie równa się `true`. W przeciwnym razie klasa jest usuwana z tego elementu.

  ```
  currentClasses = {
     star: true,
     active: false
  };
  ```

Zamiast nadawania stylu naszym elementom za pomocą klas CSS, możemy ustawiać ich style bezpośrednio. Podobnie jak w przypadku wiązania klasy, za pomocą wiązania stylu możemy nadać jednocześnie jeden styl lub wiele stylów. Pojedynczy styl może być nadany elementowi HTML za pomocą takiego kodu:

```
<p [style.color]="'greenyellow'"></p>
```

W tym kodzie element akapitu będzie miał kolor zielono-żółty (ang. *greenyellow*). W wiązaniu niektóre style mogą być jeszcze bardziej rozszerzone, na przykład szerokość paragrafu, którą możemy zdefiniować wraz z jednostkami.

```
<p [style.width.px]="100"></p>
```

Kup książkę Poleć książkę

Akapity będą szerokie na 100 pikseli. Jeśli będziemy musieli nadać kilka stylów naraz, możemy użyć składni obiektu.

```
<p [style]="currentStyles"></p>
```

Zmienna `currentStyles` to właściwość komponentu. Wyrażenie używane w wiązaniu stylów może przyjąć jedną z następujących wartości:

- Styl w postaci łańcucha znaków ze stylami oddzielonymi średnikami, na przykład `'color: greenyellow; width: 100px'`.
- Styl z postaci obiektu, w którym klucze są nazwami stylów, a wartości są ich wartościami.

```
currentStyles = {
   color: 'greenyellow',
   width: '100px'
};
```

Wiązanie klasy i wiązanie stylów to funkcjonalności wbudowane w Angularze. Wraz z konfiguracją stylów CSS, którą możemy zdefiniować w dekoratorze `@Component`, daje to nieskończone możliwości nadawania stylów komponentom Angulara. Zdolność czytania danych z szablonu i zapisywania ich w klasie komponentu jest równie atrakcyjną funkcją.

Odczytywanie danych z szablonu

W poprzednim punkcie nauczyliśmy się wykorzystywać wiązanie właściwości w celu wyświetlania danych z klasy komponentu. W rzeczywistości zwykle wymagany jest dwukierunkowy przepływ danych przez komponenty. Aby pobrać dane z szablonu i zapisać je w klasie komponentu, używamy techniki zwanej **wiązaniem wydarzeń** (ang. *event binding*). Spójrzmy na ten fragment kodu HTML:

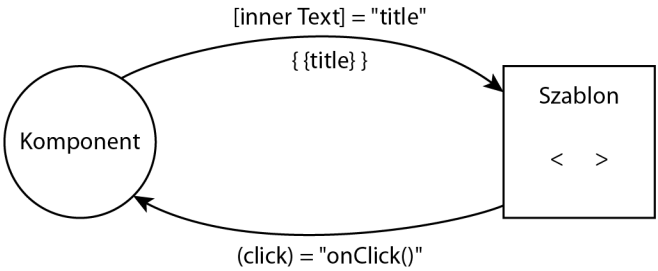
```
<button (click)="onClick()">Kliknij mnie</button>
```

Wiązanie wydarzeń nasłuchuje wydarzeń DOM na docelowym elemencie HTML i reaguje na te wydarzenia przez wywołanie odpowiedniej metody w klasie komponentu. W tym przypadku komponent wywołuję metodę `onClick`, gdy użytkownik naciśnie przycisk. Wydarzenie umieszczone w nawiasach nazywamy **wydarzeniem docelowym** i jest to wydarzenie, którego w danym momencie nasłuchujemy. W Angularze wiązanie wydarzeń obsługuje wszystkie natywne wydarzenia DOM, wymienione na stronie: *https://developer.mozilla.org/en-US/docs/Web/Events*.

Wyrażenie po prawej stronie nazywa się **wyrażeniem szablonu** i w tym przypadku to metoda `onClick` z klasy komponentu.

Współpraca szablonu komponentu z powiązaną klasą TypeScript jest przedstawiona na schemacie na rysunku 4.3.

Kup książkę Poleć książkę



Rysunek 4.3. Interakcja komponent-szablon

Możemy stosować tę samą zasadę co w przypadku współpracy z szablonem komponentu i klasą, kiedy chcemy, by komponenty się ze sobą komunikowały.

Wzajemne porozumiewanie się komponentów

W skrócie komponenty Angulara eksponują publiczne API umożliwiające im komunikację z innymi komponentami. API obejmuje właściwości wejścia, którego używamy, by dostarczyć komponentowi danych. API eksponuje również właściwości, z którymi możemy związać nasłuchiwanie wydarzeń (ang. *event listeners*), dzięki czemu w odpowiednim momencie będziemy otrzymywać informacje o zmianach stanu komponentu.

Zobaczmy, jak Angular rozwiązuje problem wstrzykiwania danych do komponentów i wyciągania danych z komponentów na szybkich i prostych przykładach pokazanych w następnych punktach.

Przekazywanie danych za pomocą wiązania wejścia

Rozwiniemy nasz moduł `products` i utworzymy nowy komponent wyświetlający szczegóły produktu, takie jak nazwa i cena. Dane reprezentujące określone szczegóły produktu będą dynamicznie przekazywane z komponentu listy produktów.

Uwaga

Na tym etapie będziemy przekazywać i wyświetlać tylko nazwę produktu. Jeśli chcesz pracować na przykładowym kodzie, skopiuj style CSS z pliku *style.css*, który znajdziesz w materiałach do pobrania pod adresem: *https://ftp.helion.pl/przyklady/pozan4.zip*, tak jak wspomniano w podrozdziale „Wymagania techniczne".

Kup książkę Poleć książkę

Zaczniemy od utworzenia i konfiguracji komponentu wyświetlającego szczegóły produktu.

1. Uruchom poniżej pokazane polecenie CLI Angulara w folderze *src\app\products*, aby dodać nowy komponent Angulara do projektu.

   ```
   ng generate component product-detail
   ```

2. Otwórz plik *product-detail.component.ts* nowego komponentu i zaimportuj artefakt `Input` z pakietu npm o nazwie *@angular/core*.

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

   Artefakt `Input` to dekorator właściwości Angulara używany wtedy, gdy chcemy przekazać dane z jednego komponentu *w dół* do innego komponentu.

3. Zdefiniuj właściwość `name` w klasie `ProductDetailComponent`, która używa dekoratora `Input`, i zainicjalizuj ją jako pusty łańcuch znaków.

   ```
   @Input() name = '';
   ```

4. Otwórz teraz plik *product-detail.component.html* i dodaj następujący kod, by wyświetlić nazwę produktu:

   ```
   <h2>Szczegóły produktu</h2>
   <h3>{{name}}</h3>
   ```

   W tym szablonie za pomocą składni interpolacji przekształciliśmy właściwość `name` na tekst i wyświetliliśmy ją na stronie.

Wykonaliśmy już większą część pracy, teraz musimy przekazać wartość właściwości wejściowej `name` z komponentu listy produktów, by mogła być odpowiednio wyświetlana w komponencie szczegółów produktu.

1. Otwórz teraz plik *product-list.component.ts* i w klasie `ProductListComponent` utwórz właściwość `selectedProduct`.

   ```
   selectedProduct = '';
   ```

2. Otwórz teraz plik *product-list.component.html* i zastąp jego zawartość takim szablonem:

   ```
   <h2>Lista produktów</h2>
   <ul>
      <li (click)="selectedProduct = 'Kamera internetowa'"
      ↪>Kamera internetowa</li>
      <li (click)="selectedProduct = 'Mikrofon'">Mikrofon</li>
      <li (click)="selectedProduct = 'Klawiatura bezprzewodowa'"
      ↪>Klawiatura bezprzewodowa</li>
   </ul>
   ```

   W tym kodzie przy użyciu nieuporządkowanej listy HTML utworzyliśmy listę produktów. Gdy użytkownik klika produkt, właściwość `selectedProduct` jest

Kup książkę Poleć książkę

ustawiana odpowiednio przez powiązanie wydarzenia `click` z elementem `<li>`.

3. Teraz pod listą produktów dodaj ten kod, by wczytać komponent szczegółów produktu.

```
<app-product-detail [name]="selectedProduct"></app-product-detail>
```

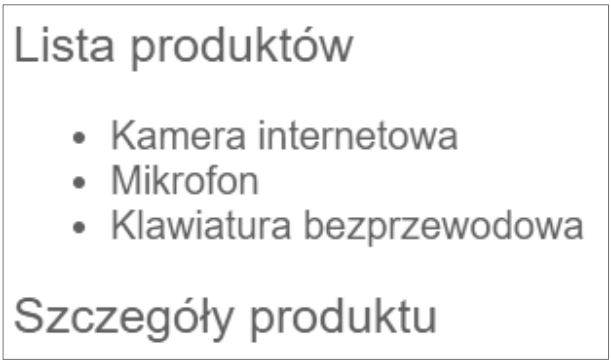
W tym kodzie wykorzystaliśmy wiązanie właściwości, by powiązać wartość właściwości `selectedProduct` z właściwością wejściową `name` komponentu szczegółów produktu. To podejście nazywane jest **wiązaniem wejścia** (ang. *input binding*).

Wskazówka

Są przypadki, gdy chcemy przekazać statyczny tekst lub wartość, która na pewno nigdy się nie zmieni. Wtedy możemy pominąć nawiasy kwadratowe wiązania wejścia, tak jak pokazano poniżej.

```
<app-product-detail name="Kamera internetowa"></app-product-detail>
```

Jeśli uruchomimy teraz aplikację, powinniśmy zobaczyć listę produktów widoczną na rysunku 4.4.

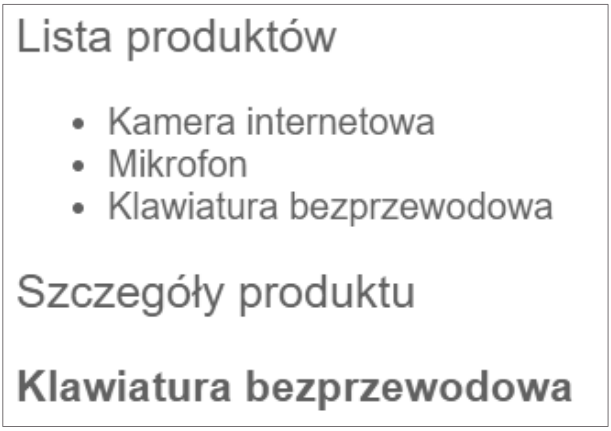


Rysunek 4.4. Lista produktów

Zauważ, że wyświetlona została również część szczegółów produktu, choć nie wybraliśmy jeszcze żadnego z nich. W następnym rozdziale dowiemy się, jak rozwiązać ten problem za pomocą wbudowanych dyrektyw Angulara.

Kliknięcie produktu z listy powinno spowodować wyświetlenie nazwy produktu w części *Szczegóły produktu*, tak jak pokazano na rysunku 4.5.

Kup książkę Poleć książkę



Rysunek 4.5. Szczegóły produktu

To już wszystko! Udało nam się przekazać dane z jednego komponentu do drugiego. W następnym punkcie nauczymy się nasłuchiwać wydarzeń w komponencie i na nie reagować.

Nasłuchiwanie wydarzeń za pomocą wiązania wyjścia

Dowiedzieliśmy się, że wiązanie wejścia jest używane do przekazywania danych między komponentami. Ta metoda ma zastosowanie, gdy mamy dwa komponenty, jeden działający jako nadrzędny, a drugi jako podrzędny. A co, jeśli chcielibyśmy przekazywać dane w drugą stronę, z komponentu podrzędnego do nadrzędnego? Jak poinformujemy komponent nadrzędny o określonych działaniach mających miejsce w komponencie podrzędnym?

Rozważmy scenariusz, w którym komponent szczegółów produktu powinien mieć przycisk dodający ten produkt do koszyka. Koszyk byłby właściwością komponentu listy produktów. W jaki sposób komponent szczegółów produktu zawiadomi komponent listy produktów o naciśnięciu przycisku? Zobaczymy, jak możemy zaimplementować tę funkcjonalność w naszej aplikacji:

1. Otwórz plik *product-detail.component.ts* i zaimportuj artefakty `Output` i `EventEmitter` z pakietu npm o nazwie *@angular/core*.

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

   Artefakt `Output` to właściwość Angulara używana, gdy chcemy utworzyć zdarzenia, które będą wywoływane z jednego komponentu **wyżej** do innego. Klasa `EventEmitter` służy do nadawania tych zdarzeń.

2. Zdefiniuj nową właściwość komponentu, która używa dekoratora `Output` i jest inicjalizowana jako nowy obiekt klasy `EventEmitter`.

   ```
   @Output() bought = new EventEmitter();
   ```

3. W tym samym pliku TypeScript dodaj następującą metodę:

   ```
   buy() {
      this.bought.emit();
   }
   ```

   Metoda `buy` wywołuje metodę `emit` na zdarzeniu `bought` utworzonym w poprzednim kroku. Metoda `emit` nadaje zdarzenie i zawiadamia wszystkie komponenty nasłuchujące w danym momencie tego zdarzenia.

4. Teraz do szablonu dodaj element `<button>` i do jego zdarzenia `click` przywiąż metodę `buy`.

   ```
   <h2>Szczegóły produktu</h2>
   <h3>{{name}}</h3>
   <button (click)="buy()">Kup teraz</button>
   ```

5. Już prawie skończyliśmy! Teraz musimy dokończyć wiązanie w komponencie listy produktów, by ten komponent mógł się komunikować z komponentem szczegółów produktu. Otwórz plik *product-list.component.ts* i dodaj taką metodę:

   ```
   onBuy() {
      window.alert('Właśnie kupiłeś: ${this.selectedProduct}!');
   }
   ```

   W tym fragmencie kodu używamy wbudowanej w przeglądarce metody `alert`, by wyświetlić okno dialogowe.

6. Na koniec zmieniamy selektor komponentu szczegółów produktu w pliku *product-list.component.html*.

   ```
   <app-product-detail [name]="selectedProduct" (bought)="onBuy()">
   ↪</app-product-detail>
   ```

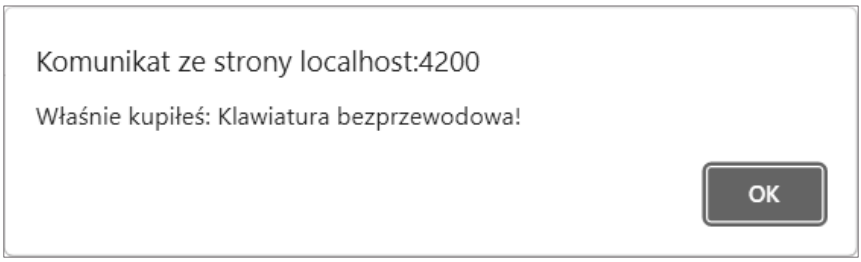
   W tej linii kodu używamy wiązania zdarzenia, by przywiązać metodę `onBuy` komponentu `ProductListComponent` do właściwości wyjściowej `bought` komponentu szczegółów produktu. To podejście nazywamy **wiązaniem wyjścia** (ang. *output binding*).

Wybierz produkt z listy i kliknij przycisk *Kup teraz*. Powinno się otworzyć okno dialogowe widoczne na rysunku 4.6.

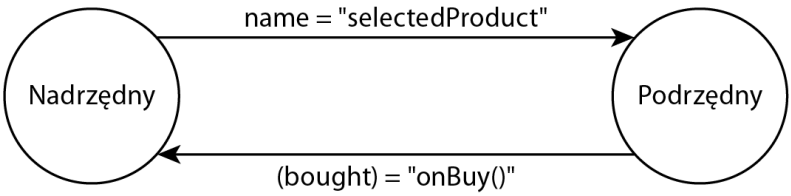
Na rysunku 4.7 przedstawiono schemat działania komunikacji, którą właśnie omówiliśmy.

Kup książkę Poleć książkę

Zdarzenie wyjściowe komponentu szczegółów produktu nie robi nic więcej i nic mniej poza nadawaniem zdarzenia do komponentu nadrzędnego. Jednak możemy użyć tego zdarzenia, by przekazać dane poprzez metodę `emit`, czego nauczymy się w następnym punkcie.



Rysunek 4.6. Okno dialogowe



Rysunek 4.7. Komunikacja dwustronna komponentów

Przekazywanie danych przez niestandardowe zdarzenia

Metoda `emit` właściwości `EventEmitter` może przyjmować dowolne dane do przekazania komponentowi nadrzędnemu. Poprawny sposób polega na początkowym zdefiniowaniu typu danych, które mogą być przekazywane do właściwości `EventEmitter`.

Na tym etapie komponent listy produktów już wie, jaki produkt został wybrany. Wie zatem, jaki produkt został kupiony. Załóżmy, że tak jednak nie jest i komponent listy produktów może poznać wybrany produkt tylko po naciśnięciu przez użytkownika przycisku *Kup teraz*. Użylibyśmy klasy generycznej dla klasy `EventEmitter`, by zadeklarować typ danych, które będą przekazywane do komponentu listy produktu.

```
@Output() bought = new EventEmitter<string>();
```

Metoda `buy` klasy `ProductDetailComponent` wywołałaby potem metodę `emit` i przekazałaby tym samym wartość tekstową.

```
buy() {
   this.bought.emit(this.name);
}
```

Kup książkę Poleć książkę

Dane byłyby dostępne w szablonie komponentu listy produktu przez obiekt `$event`.

```
<app-product-detail [name]="selectedProduct" (bought)="onBuy($event)">
↪</app-product-detail>
```

Obiekt `$event` to zarezerwowane słowo kluczowe w Angularze zawierające dane wsadowe emitera zdarzeń z wiązania wyjścia. Dodatkowo sygnatura metody `onBuy` w klasie `ProductListComponent` powinna się odpowiednio zmienić.

```
onBuy(name: string) {
   window.alert('You just bought ${name}!');
}
```

Wiązania wejścia i wyjścia są doskonałym sposobem komunikacji między komponentami za pomocą publicznego API. Jednak są przypadki, gdy chcemy mieć bezpośredni dostęp do właściwości lub metody przy użyciu lokalnych zmiennych odwołania w szablonie.

Lokalne zmienne odwołania w szablonach

Zobaczyliśmy, jak wiązać dane w naszych szablonach za pomocą interpolacji z podwójnym nawiasem klamrowym. Poza tym często widzimy nazwane identyfikatory poprzedzone symbolem # w elementach należących do naszych komponentów, a nawet w standardowych elementach HTML. Te identyfikatory odwołania, nazywane **zmiennymi odwołania szablonu**, służą do odwoływania się do komponentów nimi oznaczonych w naszym szablonie widoków i mamy do nich dostęp za pomocą skryptu. Komponenty mogą również używać ich do odwoływania się do innych elementów w modelu DOM i uzyskiwania dostępu do ich właściwości.

Wiemy już, jak komponenty się komunikują przez nasłuchiwanie nadawanych zdarzeń przy użyciu wiązania zdarzeń lub przez przekazywanie danych za pomocą wiązania wejścia. Ale gdybyśmy mogli dogłębnie zbadać komponent lub przynajmniej jego wyeksponowane właściwości i metody, a następnie uzyskać do nich dostęp bez konieczności uciekania się do wiązania wejścia i wyjścia? Ustawienie lokalnego odwołania w samym komponencie otwiera drzwi do jego publicznego API.

Wskazówka

Publiczne API komponentu składa się z wszystkich publicznych (`public`) i chronionych (`protected`) członków klasy TypeScript.

W szablonie możemy tak zadeklarować zmienną odwołania dla komponentu szczegółów produktu w pliku *product-list.component.html*.

Kup książkę Poleć książkę

```
<app-product-detail
   #product
   [name]="selectedProduct"
   (bought)="onBuy()"
></app-product-detail>
```

Dzięki temu teraz będziemy mieć bezpośrednio dostęp do członków tego komponentu, a nawet będziemy mogli wiązać ich w innych miejscach szablonu, na przykład w miejscu wyświetlania nazwy produktu.

```
<span>{{product.name}}</span>
```

W ten sposób nie musimy polegać na właściwościach wejścia i wyjścia i możemy zmieniać wartości takich właściwości.

Uwaga

Podejście z użyciem lokalnej zmiennej odwołania stosujemy, gdy nie mamy kontroli nad komponentem podrzędnym i nie możemy dodać wiązania właściwości wyjścia lub wejścia.

Wyjaśniliśmy, jak klasa komponentu współpracuje ze swoim szablonem lub innymi komponentami, ale dotychczas nie przejmowaliśmy się ich stylem.

Enkapsulacja stylów CSS

Styl CSS możemy zdefiniować w obrębie naszych komponentów, by jeszcze lepiej przygotować nasz kod do ponownego użycia. W podrozdziale „Tworzenie naszego pierwszego komponentu" dowiedzieliśmy się, jak definiować style CSS dla komponentu za pomocą zewnętrznego pliku CSS przez właściwość `styleUrls` lub przez zdefiniowanie stylów CSS w pliku komponentu TypeScript przy użyciu właściwości `styles`.

Zwyczajowe reguły specyfiki CSS (ang. *specificity*) dotyczą obu tych sposobów (patrz *https://developer.mozilla.org/docs/Web/CSS/Specificity*).

Stylami CSS i specyfiką selektorów można bardzo łatwo zarządzać z przeglądarkami wspierającymi **Shadow DOM** dzięki zakresom stylów. Style CSS mają zastosowanie do elementów zwartych w komponencie, ale nie wychodzą poza granice tych elementów.

Poza tym Angular osadza arkusze stylów na początku dokumentu, tak by mogły wpływać na pozostałe elementy naszej aplikacji. Aby się tak nie działo, możemy ustawić różne poziomy enkapsulacji widoku.

Enkapsulacja widoku to sposób, w jaki Angular zarządza zakresem CSS w obrębie komponentu, zarówno dla przeglądarek zgodnych z Shadow DOM, jak i dla tych nieobsługujących

Kup książkę Poleć książkę

tego modelu. Może to zostać zmienione przez ustawienie właściwości `encapsulation` dekoratora `@Component` w jednej z następujących wartości `ViewEncapsulation`:

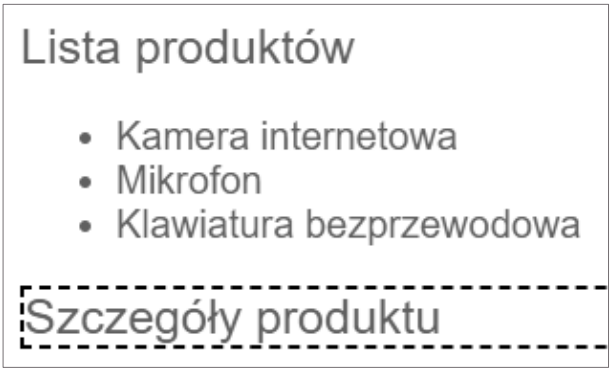
- `Emulated` — to domyślna opcja pociągająca za sobą emulację natywnego zakresu działania modelu Shadow DOM przez umieszczenie w piaskownicy (ang. *sandboxing*) reguł CSS określonego selektora, który wskazuje na komponent. Ta opcja jest zalecana, by upewnić się, że style danego komponentu nie wyciekną poza niego i żadne zewnętrzne style nie będą na niego wpływać.
- `Native` — używa wbudowanego w modelu Shadow DOM mechanizmu enkapsulacji, który działa tylko w przeglądarkach obsługujących Shadow DOM.
- `None` — szablon lub enkapsulacja stylu nie jest zapewniona. Style są wstrzykiwane tak, jakby byłby dodawane do elementu `<head>`.

Opcje `Emulated` i `None` będziemy odkrywać dalej, gdyż oferują szerokie wsparcie.

1. Otwórz plik *product-detail.component.css* i dodaj styl CSS, by zmienić kolor elementu `<h2>`.

```
h2 {
   border: 2px dashed black;
}
```

2. Uruchom aplikację za pomocą polecenia `ng serve` i zwróć uwagę, że napis *Szczegóły produktu* jest otoczony przerywaną linią, tak jak pokazano na rysunku 4.8.



Rysunek 4.8. Enkapsulacja domyślnego widoku

Ten styl nie wpłynął na część *Lista produktów*, która również jest elementem `<h2>`, ponieważ domyślna enkapsulacja zawęża zakres wszystkich zdefiniowanych stylów CSS do określonego komponentu.

Kup książkę Poleć książkę

3. Teraz otwórz plik *product-detail.component.ts* i ustaw enkapsulację komponentu na `None`.

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

@Component({
   selector: 'app-product-detail',
   templateUrl: './product-detail.component.html',
   styleUrls: ['./product-detail.component.css'],
   encapsulation: ViewEncapsulation.None
})
```

4. Przeglądarka odświeży naszą aplikację, która teraz wygląda jak na rysunku 4.9.

Rysunek 4.9. Brak enkapsulacji widoku

Na rysunku 4.9 styl wyciekł w górę drzewa komponentów i wpłynął na element `<h2>` komponentu listy produktów. Teraz część *Lista produktów* również jest otoczona przerywaną linią.

Enkapsulacja komponentu Angulara może rozwiązać wiele problemów podczas nadawania stylów naszym komponentom. Jednak powinna być używana ostrożnie, ponieważ, jak już wiemy, style CSS mogą wyciec do innych części aplikacji i spowodować niezamierzone efekty.

Inną właściwością dekoratora `@Component`, która nie jest zbyt często stosowana, ale ma wiele możliwości, jest strategia wykrywania zmian.

Wybór strategii wykrywania zmian

Wykrywanie zmian to mechanizm używany wewnętrznie przez Angular w celu wykrywania zmian zachodzących we właściwościach komponentu i odzwierciedlający te

Kup książkę Poleć książkę

zmiany w widoku. Wykrywanie zmian jest uruchamiane w odpowiedzi na określone zdarzenia, takie jak naciśnięcie przycisku przez użytkownika, zakończenie asynchronicznego żądania albo zakończenie wykonywania metody `setTimeout` lub `setInterval`. Angular stosuje technikę *monkey patching*, by łatać tego typu zdarzenia przez nadpisanie ich domyślnego działania za pomocą biblioteki o nazwie *Zone.js*.

Każdy komponent ma mechanizm wykrywania zmian, który sprawdza, czy jego właściwości się zmieniły, przez porównanie bieżącej wartości właściwości z poprzednią. Jeśli jest różnica, ta zmiana jest wprowadzana w szablonie komponentu. W komponencie szczegółów produktu, gdy w wyniku wspomnianego wcześniej przez nas zdarzenia zmienia się właściwość wejściowa `name`, uruchamia się mechanizm wykrywania zmian dla tego komponentu i odpowiednio aktualizuje szablon.

Jednak są przypadki, gdy takie działanie nie jest pożądane, na przykład w przypadku komponentów, które renderują bardzo duże ilości danych. W takim scenariuszu domyślny mechanizm wykrywania zmian nie jest wystarczający, gdyż może spowodować problemy z wydajnością aplikacji. Użylibyśmy raczej właściwości `changeDetection` dekoratora `@Component`, który wyznacza komponentowi wybraną strategię wykrywania zmian. Zobaczmy przykład, w którym możemy użyć mechanizmu wykrywania zmian.

1. Otwórz plik *product-detail.component.ts* i utwórz właściwość `getter` zwracającą bieżącą nazwę produktu i pokazującą wiadomość w konsoli przeglądarki.

   ```
   get productName(): string {
      console.log('Pobierz ${this.name}');
      return this.name;
   }
   ```

2. Otwórz plik *product-detail.component.html* i użyj składni interpolacji, by wyświetlić właściwość `productName` w elemencie `<span>`.

   ```
   <span>{{productName}}</span>
   ```

3. Uruchom aplikację za pomocą polecenia `ng serve`, wybierz jeden produkt z listy i sprawdź, co pokazało się w konsoli Twojej przeglądarki. Zauważysz wiadomość z właściwości `getter` *dwa razy* na wybór produktu. Jest to spowodowane przez fakt, że wykrywanie zmian jest wywoływane dwukrotnie, raz podczas inicjalizacji komponentu i drugi raz, gdy właściwość `name` zmienia się w rezultacie wyboru użytkownika.

4. Zmień dekorator `@Component` komponentu szczegółów produktu przez ustawienie właściwości `changeDetection`, by zmienić `ChangeDetectionStrategy.OnPush`.

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

Kup książkę Poleć książkę

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

Dodany kod uruchomi mechanizm wykrywania zmian tylko wtedy, gdy odwołanie właściwości wejścia `name` się zmieni.

5. Gdy przeglądarka się odświeży, spróbuj wybrać kilka produktów z listy, a zauważysz, że teraz widomość w konsoli pojawia się **raz** na wybór.

Strategia wykrywania zmian to mechanizm, który umożliwia nam zmianę sposobu, w jaki nasze komponenty wykrywają zmiany w swoich danych, co znacznie wpływa na poprawę wydajności bardzo dużych aplikacji. Na strategii wyrywania zmian kończymy omawianie konfiguracji komponentu, ale to nie wszystko, jeśli chodzi o framework Angular. Angular pozwala nam podłączyć się do określonych momentów cyklu życia komponentu, o czym dowiemy się następnym podrozdziale.

Wprowadzenie do cyklu życia komponentu

Zdarzenia cyklu życia to haki, które pozwalają nam wkroczyć w określone etapy cyklu życia komponentu i zastosować niestandardową logikę. Nie są obowiązkowe, ale mogą być bardzo cenną pomocą, jeśli zrozumiesz, jak ich używać.

Niektóre z tych haków są uważane za najlepsze praktyki, a inne pomagają w debugowaniu i zrozumieniu, co się dzieje w aplikacji tworzonej w Angularze. Hak przychodzi z interfejsem definiującym metodę, którą musimy zaimplementować. Framework Angular upewnia się, że hak został wywołany, pod warunkiem, że zaimplementowaliśmy tę metodę w komponencie.

Wskazówka

Definiowanie interfejsu w komponencie nie jest obowiązkowe, ale jest zalecane. Angular sprawdza jedynie, czy zaimplementowaliśmy daną metodę, czy nie.

Najbardziej podstawowe haki komponentu Angulara to:

- `OnInit` — wywoływany podczas inicjalizacji komponentu;
- `OnDestroy` — wywoływany podczas niszczenia komponentu;

Kup książkę Poleć książkę

- `OnChanges` — wywoływany, gdy wartości powiązanych właściwości wejścia w komponencie się zmieniają;
- `AfterViewInit` — wywoływany, gdy Angular inicjalizuje widok bieżącego komponentu oraz jego podrzędnych komponentów.

Wszystkie wymienione wyżej haki są dostępne w pakiecie `npm` o nazwie `@angular/core`, który jest częścią frameworku Angular.

Wskazówka

Pełną listę obsługiwanych haków cyklu życia znajdziesz w oficjalnej dokumentacji: *https://angular.io/guide/lifecycle-hooks* (strona w języku angielskim).

W następnych punktach przyjrzymy się bliżej każdemu z haków z listy. Zacznijmy od haka `OnInit`, który jest najbardziej podstawowym hakiem cyklu życia komponentu.

Inicjalizacja komponentu

Hak `OnInit` implementuje metodę `ngOnInit` wywoływaną podczas inicjalizacji komponentu. Na tym etapie wszystkie wiązania wejścia i właściwości zostały ustawione poprawnie i możemy ich bezpiecznie używać. Uzyskanie do nich dostępu za pomocą konstruktora komponentu może być kuszące, ale ich wartości nie będą jeszcze w tym momencie ustawione. Ten przykład pomoże nam to zrozumieć:

1. Otwórz plik *product-detail.component.ts* i dodaj konstruktor, który wpisuje do rejestru właściwość `name` i wyświetla jej wartość w konsoli przeglądarki.

```
constructor() {
   console.log('W konstruktorze właściwość name ma wartość
${this.name}');
}
```

2. Z pakietu npm *@angular/core* zaimportuj artefakt `OnInit`.

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

3. Dodaj artefakt `OnInit` do listy zaimplementowanych interfejsów klasy `ProductDetailComponent`.

```
export class ProductDetailComponent implements OnInit
```

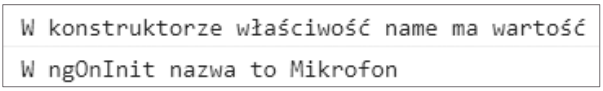
4. Dodaj następującą metodę do klasy `ProductDetailComponent`, by zapisywać w rejestrze te same informacje co w kroku 1.

```
ngOnInit(): void {
   console.log('W ngOnInit nazwa to ${this.name}');
}
```

5. Otwórz plik *product-list.component.ts* i ustaw wartość początkową właściwości `selectedProduct`.

   ```
   selectedProduct = 'Mikrofon';
   ```

6. Uruchom aplikację za pomocą polecenia `ng serve` i sprawdź informacje wyświetlane w konsoli przeglądarki, takie jak te na rysunku 4.10.



Rysunek 4.10. Informacje w konsoli

   Pierwsza informacja z konstruktora zawiera pusty łańcuch znaków jako wartość właściwości `name`. Wynika to z tego, że gdy wartość `undefined` jest przekształcana na tekst za pomocą interpolacji, automatycznie zamieniana jest na pusty łańcuch znaków. W drugiej informacji wartość właściwości `name` jest wyświetlana poprawnie.

Konstruktory powinny być puste i pozbawione logiki, poza ustawieniem początkowej wartości zmiennych. Gdy dodasz logikę biznesową do konstruktora, jej testowanie może stać się trudne.

Hak `OnInit` przydaje się również wtedy, gdy musimy zainicjalizować komponent z danymi z zewnętrznego źródła, na przykład z usługi Angulara, o czym dowiemy się więcej w rozdziale 6. „Obsługa złożonych zadań z wykorzystaniem usług".

Framework Angular zapewnia haki dla wszystkich etapów cyklu życia komponentu, od inicjalizacji do zniszczenia.

Czyszczenie zasobów komponentu

Interfejs, który zaczepiamy do zdarzenia zniszczenia komponentu, to hak `OnDestroy`, który implementuje metodę `ngOnDestroy`.

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

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent implements OnDestroy {
   title = 'my-app';

   ngOnDestroy(): void {
   }
}
```

Kup książkę Poleć książkę

Komponent jest niszczony, gdy jest usuwany z drzewa DOM strony internetowej, z następujących powodów:

- Użycie **dyrektyw strukturalnych**, o których dowiemy się więcej w rozdziale 5. „Wzbogacanie aplikacji za pomocą potoków i dyrektyw”.
- Odejście od komponentu przy użyciu routera Angulara, o którym dowiemy się więcej w rozdziale 9. „Nawigacja po aplikacji za pomocą routingu”.

Zwykle porządkujemy zasoby komponentów w metodzie `ngOnDestroy` i wykonujemy między innymi takie czynności:

- zerujemy liczniki i interwały,
- anulujemy subskrypcję ze strumieni obiektów `Observable`, o których dowiemy się więcej w rozdziale 7. „Uzyskanie reaktywności za pomocą klasy Observable i biblioteki RxJS ”.

Wiemy już, jak przekazywać dane do komponentu za pomocą wiązania wejścia. Framework Angular zapewnia hak cyklu życia o nazwie `OnChanges`, którego możemy użyć do sprawdzania, kiedy określona wartość takiego wiązania się zmieniła.

Wykrywanie zmian wiązania wejścia

Hak cyklu życia `OnChanges` jest wywoływany w momencie, gdy Angular wykrywa zmianę wartości danych wiązania wejścia. Użyjemy tego haka w komponencie szczegółów produktu, by dowiedzieć się, jak ten komponent się zachowuje, gdy wybieramy różne produkty z listy.

1. W pliku *product-detail.component.ts* zaimportuj artefakty `OnChanges` i `SimpleChanges`.

   ```
   import { Component, Input, Output, EventEmitter,
   ↪OnChanges, SimpleChanges } from '@angular/core';
   ```

2. Zmień definicję w klasie `ProductDetailComponent`, by zaimplementować interfejs `OnChanges`.

   ```
   export class ProductDetailComponent implements OnChanges
   ```

3. Zaimplementuj metodę `ngOnChanges` zdefiniowaną w interfejsie `OnChanges`. Ta metoda jako parametr przyjmuje obiekt typu `SimpleChanges` zawierający jeden klucz dla każdej właściwości wejścia, która się zmienia. Każdy klucz wskazuje na inny obiekt z właściwościami `currentValue` i `previousValue`, które przechowują nową i starą wartość właściwości wejścia.

   ```
   ngOnChanges(changes: SimpleChanges): void {
      const product = changes['name'];
      const oldValue = product.previousValue;
   ```

```
    const newValue = product.currentValue;
    console.log('Produkt zmienił się z ${oldValue} na ${newValue}');
  }
```

Ten fragment kodu śledzi zmiany właściwości wejścia o nazwie `name` i zapisuje w konsoli przeglądarki stare i nowe wartości tej właściwości.

4. Aby sprawdzić, co się dzieje w aplikacji, uruchom polecenie `ng serve`, wybierz produkt z listy i zwróć uwagę na informacje w konsoli w przeglądarce, takie jak na rysunku 4.11.

```
Produkt zmienił się z undefined na
Angular is running in development mode. Call enableProdMode() to enable production mode.
Produkt zmienił się z  na Kamera internetowa
```

Rysunek 4.11. Informacje w konsoli

Zwróć uwagę na pierwszą i trzecią linię na rysunku 4.11. Trzecia linia informuje, że z listy produktów została wybrana *Kamera internetowa*. Natomiast pierwsza linia mówi, że produkt został zmieniony z `undefined` na pusty łańcuch znaków. Dlaczego tak jest?

Zdarzenie `OnChanges` jest wywoływane przy pierwszym ustawieniu wartości i przy każdej kolejnej zmianie wartości mającej miejsce w mechanizmie wiązania. Na początku zmienna `oldValue` ma wartość `undefined`, gdyż właściwość nie została jeszcze ustawiona. Zmienna `newValue` to pierwsza wartość właściwości — w naszym przypadku pusty łańcuch znaków, który wynika z początkowej wartości właściwości `selectedProduct` komponentu listy produktów. W celu wyeliminowania niepotrzebnych wpisów w rejestrze możemy sprawdzić, czy jest to pierwsza zmiana za pomocą metody `isFirstChange`.

```
ngOnChanges(changes: SimpleChanges): void {
  const product = changes['name'];
  if (!product.isFirstChange()) {
    const oldValue = product.previousValue;
    const newValue = product.currentValue;
    console.log('Produkt zmienił się z ${oldValue} na ${newValue}');
  }
}
```

Po odświeżeniu przeglądarki w konsoli zobaczymy poprawny komunikat.

W następnym punkcie omówimy ostanie zdarzenie cyklu życia komponentu Angulara — hak `AfterViewInit`.

Kup książkę Poleć książkę

Dostęp do komponentów podrzędnych

Hak `AfterViewInit` cyklu życia komponentu Angulara jest wywoływany po zakończeniu obu tych zdarzeń:

- szablon HTML komponentu został zainicjalizowany,
- szablony HTML komponentów podrzędnych zostały zainicjalizowane.

Możemy zobaczyć, jak działa zdarzenie `AfterViewInit` na przykładzie komponentów listy produktów i szczegółów produktu.

1. Otwórz plik *product-list.component.ts* i zaimportuj artefakty `AfterViewInit` i `ViewChild` z pakietu npm *@angular/core*.

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

2. Dodaj nowe wyrażenie `import`, by zaimportować klasę `ProductDetailComponent`.

   ```
   import { ProductDetailComponent } from '../product-detail/
   ↪productdetail.component';
   ```

3. Utwórz właściwość `productDetail` w klasie `ProductListComponent`.

   ```
   @ViewChild(ProductDetailComponent) productDetail:
   ProductDetailComponent | undefined;
   ```

   W tym kodzie zdefiniowaliśmy właściwość typu `ProductDetailComponent` lub `undefined`. Ten drugi typ jest wymagany, gdyż framework Angular domyślnie działa w **trybie restrykcyjnym** (ang. *strict mode*). W trybie restrykcyjnym Angular upewnia się, że w naszych aplikacjach tworzonych w Angularze stosujemy silne typowanie. Silne typowanie wykrywa problemy na wczesnym etapie, jeszcze przed wdrożeniem, i pomaga unikać błędów w naszych aplikacjach.

   Wiemy już, jak uzyskiwać dane z klasy komponentu z poziomu szablonu HTML przy użyciu lokalnych zmiennych odwołania. Możemy również użyć dekoratora `@ViewChild`, by uzyskiwać dane z komponentu podrzędnego z poziomu komponentu nadrzędnego. Dekorator `@ViewChild` to dekorator właściwości przyjmujący jako parametr typ komponentu, z którego chcemy pobierać dane.

4. Zmień definicję klasy `ProductListComponent`, by implementowała interfejs `AfterViewInit`.

   ```
   export class ProductListComponent implements AfterViewInit
   ```

5. Interfejs `AfterViewInit` implementuje metodę `ngAfterViewInit`, której możemy użyć, by uzyskać dostęp do właściwości `productDetail`.

   ```
   ngAfterViewInit(): void {
   ```

```
    if (this.productDetail) {
      console.log(this.productDetail.name);
    }
}
```

W tej metodzie najpierw sprawdzamy, czy właściwość `productDetail` została ustawiona, ponieważ zadeklarowaliśmy już ją jako `undefined`. Gdy pobieramy wartość właściwości `productDetail`, uzyskujemy instancję klasy `ProductDetailComponent`. Mamy dostęp do wszystkich członków jej publicznego API, między innymi do właściwości `name`.

Zdarzeniem `AfterViewInit` kończymy naszą podróż przez cykl życia komponentów Angulara. Haki cyklu życia komponentu są przydatną funkcjonalnością frameworku i będziemy z nich bardzo często korzystać podczas tworzenia aplikacji Angulara.

Podsumowanie

W tym rozdziale zobaczyliśmy, jak zbudowane są komponenty Angulara i poznaliśmy różne sposoby ich tworzenia. Nauczyliśmy się tworzyć odrębne komponenty lub rejestrować komponenty z modułu Angulara. Omówiliśmy oddzielanie szablonu HTML komponentu przez umieszczenie go w pliku zewnętrznym, gdyż to znacznie ułatwi jego utrzymanie w przyszłości. Ponadto zobaczyliśmy, jak zrobić to samo z arkuszem stylów, który chcemy przypisać do komponentu w przypadku, gdy nie chcemy umieszczać stylów bezpośrednio w kodzie szablonu. Nauczyliśmy się także używać składni szablonu Angulara i wiemy, jak wchodzić w interakcję z szablonem komponentu. Co więcej, zobaczyliśmy, jak komponenty wymieniają ze sobą informacje w obie strony za pomocą właściwości i wiązania zdarzeń.

Przeszliśmy przez dostępne opcje Angulara służące do tworzenia API o wielu możliwościach dla naszych komponentów, dzięki czemu nasze komponenty mogą ze sobą współpracować, możemy konfigurować ich właściwości przez przypisanie statycznych wartości lub zarządzanych wiązań. Zobaczyliśmy również, jak komponenty mogą pełnić rolę komponentu goszczącego dla innego komponentu podrzędnego przez utworzenie instancji niestandardowego elementu w szablonie komponentu podrzędnego, tworząc w ten sposób podwalimy większych drzew komponentów w naszej aplikacji. Parametry wyjściowe zapewnią wymaganą warstwę interaktywności, gdy sprawimy, że komponent będzie nadawać zdarzenia. W ten sposób nasze komponenty mogą się komunikować z dowolnym komponentem nadrzędnym, który koniec końców może być dla nich hostem.

Odwołania w szablonach utorowały nam drogę do tworzenia odniesień w naszych elementach niestandardowych, które możemy wykorzystać jako punkt dostępu do ich

Kup książkę Poleć książkę

właściwości i metod z poziomu szablonu w sposób deklaratywny. Przegląd wbudowanych funkcji obsługi enkapsulacji widoku CSS w Angularze rzucił dodatkowe światło na to, jak możemy korzystać zakresu CSS modelu Shadow DOM w oparciu o komponent. W końcu dowiedzieliśmy się, jak ważne jest wykrywanie zmian w aplikacji napisanej w Angularze i jak możemy dostosować ten mechanizm do naszych potrzeb, by jeszcze bardziej poprawić wydajność naszej aplikacji.

Przeszliśmy przez cykl życia komponentu i nauczyliśmy się wykonywać niestandardową logikę za pomocą wbudowanych haków cyklu życia. Wciąż jeszcze mamy wiele do odkrycia, jeśli chodzi o zarządzanie szablonem w Angularze, szczególnie dwie najczęściej wykorzystywane w Angularze koncepcje, dyrektywy i potoki, co będzie tematem następnego rozdziału.

Kup książkę Poleć książkę


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
GRUPA
Helion

Poznaj najlepsze strony Angulara!

Angular od lat jest uważany za wiodącą platformę programistyczną JavaScript. Profesjonalistom ułatwia tworzenie poprawnego, czystego kodu, umożliwia bezproblemowe testowanie, a sama praca z nim jest przyjemna, efektywna i satysfakcjonująca. Aby w pełni skorzystać z możliwości Angulara, koniecznie trzeba się zapoznać z architekturą jego aplikacji, a także z modułami i komponentami.

Ta książka szczególnie przyda się osobom, które rozpoczynają pracę z Angularem. Dzięki niej szybko zaczniesz tworzyć aplikacje z wykorzystaniem wiersza poleceń (CLI), pisać testy jednostkowe i używać stylów zgodnych ze standardem Material Design. Dowiesz się również, jak wdrażać aplikacje w środowisku produkcyjnym. W tym wydaniu zaprezentowano wiele nowych funkcjonalności i praktyk ułatwiających pracę twórcom frontendów. Dodano nowy rozdział poświęcony klasie Observable i bibliotece RxJS, a także rozszerzono zakres informacji o obsłudze błędów i debugowaniu w Angularze. Poszczególne zagadnienia zostały zilustrowane przykładami rzeczywistych rozwiązań, a prezentowany kod powstał zgodnie z najlepszymi praktykami programistycznymi.

W książce:

- wdrażanie nowej aplikacji w Angularze od podstaw
- korzystanie ze standardowych komponentów i tworzenie własnych
- szablony obsługiwane przez Angular
- usługi danych HTTP i uzyskiwanie dostępu do API
- budowa aplikacji z osobnymi API
- debugowanie aplikacji i obsługa błędów

Aristeidis Bampakos jest greckim programistą z ponad dwudziestoletnim doświadczeniem. Specjalizuje się w tworzeniu aplikacji w Angularze. W 2020 roku otrzymał tytuł GDE (Google Developer Expert) dla platformy Angular.

Pablo Deeleman jest autorem kilku książek o Angularze. Od 1998 roku programuje w języku JavaScript dla takich firm jak Gameloft, Red Hat czy Dynatrace. Pracuje jako główny inżynier oprogramowania w firmie Twilio.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0385-2
9 788328 903852
Cena: 89,00 zł
packt