Technologia i rozwiązania

Angular 2

Tworzenie interaktywnych aplikacji internetowych

Gion Kunz

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Mastering Angular 2 Components

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-3196-9

First published in the English language under the title 'Mastering Angular 2 Components – (978178588-4641)'.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Spis treści

ROZDZIAŁ 2

Trzy, dwa, jeden, start!

W tym rozdziale rozpoczniemy tworzenie aplikacji do zarządzania zadaniami. Od razu przejdziemy do rzeczy i zaczniemy tworzyć komponenty niezbędne do obsługi prostej listy zadań. W tym rozdziale poznasz następujące zagadnienia:

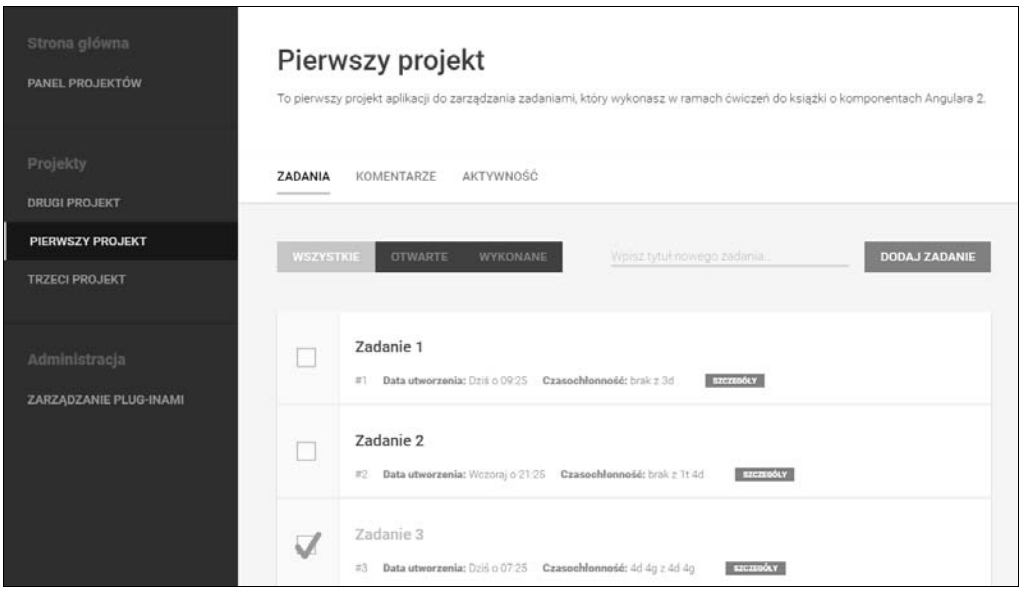
- uruchamianie aplikacji napisanej w Angularze za pomocą głównego komponentu;
- wejścia i wyjścia komponentu;
- dowiązywanie właściwości do elementu hostującego;
- enkapsulacja stylów i widoku;
- import szablonów HTML za pomocą mechanizmu wczytywania tekstu narzędzia SystemJS;
- wykorzystanie `EventEmitter` do emitowania własnych zdarzeń;
- dwukierunkowe dowiązania danych;
- cykl życia komponentu.

Zarządzanie zadaniami

Po poprzednim rozdziale, w którym opisałem podstawowe koncepcje, od razu przystąpimy do wykonania komponentów przykładowej aplikacji do zarządzania zadaniami. Najpierw zaprezentuję pewne podstawowe pojęcia, a następnie sprawdzimy je w praktycznym przykładzie. W trakcie całego rozdziału będziesz zdobywać wiedzę, w jaki sposób zapewnić odpowiednią strukturę aplikacji dzięki wykorzystaniu komponentów. Dotyczy ona zarówno struktury folderów, jak i interakcji pomiędzy poszczególnymi komponentami.

Wizja

Aplikacja do zarządzania zadaniami powinna umożliwiać użytkownikom łatwe zarządzanie niewielkimi projektami i ich organizowanie. Użyteczność to jeden z najważniejszych elementów każdej aplikacji — warto zaprojektować aplikację w sposób nowoczesny i przyjazny dla użytkownika.



Wygląd aplikacji do zarządzania zadaniami, którą będziemy tworzyli w tej książce

Aplikacja powstanie z wielu różnych komponentów umożliwiających utworzenie platformy, która zapewni elastyczne zarządzanie zadaniami. Zdefiniujmy podstawowe funkcjonalności aplikacji:

- zarządzanie zadaniami w wielu projektach i prezentowanie podsumowania projektu;
- proste harmonogramowanie i mechanizm śledzenia czasu pracy;
- widok podsumowania z wykresami graficznymi;
- śledzenie aktywności i wizualny log wszystkich zmian;
- prosty system komentarzy, który działałby między różnymi komponentami.

Aplikacja do zarządzania zadaniami posłuży jako główny przykład do zobrazowania współpracy komponentów. Oznacza to jednak, że klocki, z których będziemy budować, będą poddane głównemu tematowi książki. Oczywiście poza komponentami aplikacja potrzebuje innych funkcjonalności, na przykład projektu wizualnego, zarządzania sesją i innych istotnych elementów. Choć kod dotyczący między innymi warstwy wizualnej można pobrać ze strony WWW, w książce skupimy się na kodzie samych komponentów.

Zaczynamy od zera

Zacznijmy od utworzenia nowego folderu o nazwie *angular-2-komponenty*, w którym znajdzie się budowana aplikacja:

1. Otwórz okno konsoli wewnątrz nowo utworzonego folderu i wykonaj poniższe polecenie, aby zainicjalizować nowy projekt Node.js:

   ```
   npm init
   ```

2. Zakończ kreator, odpowiadając na wszystkie pytania przy użyciu wartości domyślnych (klawisz *Enter*).

3. Ponieważ używamy JSPM do zarządzania zależnościami, musimy zainstalować to narzędzie jako zależność projektu Node.js:

   ```
   npm install jspm --save-dev
   ```

4. Zainicjalizuj projekt JSPM wewnątrz folderu aplikacji. Zastosuj domyślne ustawienia dla wszystkich wartości (klawisz *Enter*) poza ustawieniami dotyczącym transpilatora — wybierz wartość TypeScript. Rozpocznij kreator poleceniem:

   ```
   jspm init
   ```

5. Użyj JSPM do instalacji istotnych pakietów Angulara 2 jako zależności projektu. Dodatkowo zainstalujemy inne zależności, między innymi plugin, który wczytuje pliki tekstowe jako moduły. Więcej informacji na ten temat pojawi się w dalszej części rozdziału.

   ```
   jspm install npm:@angular/core npm:@angular/common npm:@angular/compiler
   npm:@angular/platform-browser-dynamic npm:@angular/platform-browser npm:rxjs
   text reflect-metadata zone.js
   ```

Sprawdźmy, co udało nam się wykonać za pomocą narzędzi wiersza poleceń NPM i JSPM.

Plik *package.json* to główny plik konfiguracyjny Node.js, który posłuży również jako baza dla JSPM (menedżer pakietów) i SystemJS (wczytywanie modułów i transpilacja). Jeśli zajrzysz do pliku *package.json*, zobaczysz fragment dotyczący zależności JSPM:

```
"jspm": {
    "dependencies": {
      "@angular/common": "npm:@angular/common@^2.0.1",
      "@angular/compiler": "npm:@angular/compiler@^2.0.1",
      "@angular/core": "npm:@angular/core@^2.0.1",
      "@angular/platform-browser": "npm:@angular/platform-browser@^2.0.1",
      "@angular/platform-browser-dynamic": "npm:@angular/
      ↪platform-browser-dynamic@^2.0.1",
      "reflect-metadata": "npm:reflect-metadata@^0.1.8",
      "rxjs": "npm:rxjs@^5.0.0-beta.12",
      "text": "github:systemjs/plugin-text@^0.0.9",
      "zone.js": "npm:zone.js@^0.6.25"
    },
```

```
    "devDependencies": {
      "typescript": "npm:typescript@1.8.10"
    }
  }
```

Przyjrzyjmy się najważniejszym zależnościom zainstalowanym przy użyciu JSPM i ich zastosowaniu.

Pakiet	Opis
`@angular/core`	To główny pakiet dotyczący Angulara 2, pobieranym z repozytorium NPM. Jeśli pamiętasz z rozdziału 1., JSPM to tylko menedżer, który korzysta z innych repozytoriów. Pakiet `core` zawiera wszystkie najistotniejsze moduły, na przykład dekorator `@Component`, wykrywanie zmian, wstrzykiwanie zależności itp.
`@angular/common`	Zawiera najbardziej podstawowe dyrektywy, takie jak `NgIf` i `NgFor`. Dodatkowo zawiera wszystkie podstawowe przekształcenia i dyrektywy związane z obsługą formularzy.
`@angular/compiler`	Zawiera wszystkie elementy niezbędne do kompilacji szablonów widoków. Angular nie tylko umożliwia wcześniejszą kompilację szablonów w celu uzyskania lepszego czasu wczytywania, ale dodatkowo pozwala na dynamiczną kompilację szablonu tekstowego na wersję skompilowaną. Ten pakiet jest niezbędny do kompilacji szablonów „w locie".
`@angular/platform-browser-dynamic`	Zawiera funkcjonalność pozwalającą uruchomić aplikację z poziomu przeglądarki. Ten rodzaj inicjalizacji wykorzystuje rozwiązanie dynamiczne, czyli dokonuje kompilacji szablonów „w locie".
`typescript`	Ta zależność deweloperska dotyczy transpilacji kodu w TypeScripcie przez SystemJS. W ten sposób uzyskujemy automatyczną zamianę kodu w ECMAScripcie 6 i TypeScripcie na kod w ECMAScripcie 5 bezpośrednio w przeglądarce.
`text`	Domyślnie SystemJS obsługuje wczytywanie tylko plików JavaScriptu, ale ten dodatek dodaje obsługę plików tekstowych. To szczególnie przydatne rozwiązanie, jeśli chce się wczytać szablony HTML i uniknąć asynchronicznych żądań.

Pozostałe zależności są niezbędne do poprawnego działania Angulara 2 i stanowią biblioteki uzupełniające lub wdrożenie funkcjonalności, które pojawią dopiero w projektowanych wersjach języka ECMAScript.

Głównym punktem wejścia do aplikacji z poziomu przeglądarki internetowej będzie strona indeksu. Plik *index.html* zapewni realizację następujących zadań.

- Wczyta z serwerów CDN uzupełnienie przeglądarki o elementy ECMAScriptu 6, jeśli jeszcze ich nie posiada. W ten sposób upewnimy się, że przeglądarka rozumie najnowsze API ECMAScript 6.
- Wczytanie SystemJS i pliku *config.js*, który zawiera informacje o odwzorowaniu bibliotek wygenerowane przez JSPM.
- Użycie funkcji `System.import()` do wczytania i wykonania głównego punktu wejścia do aplikacji, czyli pliku *bootstrap.js*.

Utwórz plik *index.html* wewnątrz głównego folderu projektu:

```
<!doctype html>
<html>
<head lang="pl">
   <meta charset="UTF-8">
   <title>Angular 2. Komponenty</title>
</head>
<body>

<script src="https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.35.1/es6-shim.
↪min.js"></script>
<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script>
   System.import('lib/bootstrap.js');
</script>
</body>
</html>
```

Przejdźmy do komponentu aplikacji. Potraktuj go jako najbardziej zewnętrzny komponent zapewniający działanie całej aplikacji. To właśnie on reprezentuje całą aplikację. Każda aplikacja wymaga jednego i tylko jednego takiego komponentu. Komponent ten staje się korzeniem całego drzewa komponentów.

Nadamy głównemu komponentowi nazwę `App`, ponieważ reprezentuje całą aplikację. Utwórz komponent wewnątrz folderu *lib*, który posłuży jako główny folder elementów projektu. Utwórz plik *app.js* o następującej zawartości:

```
// Potrzebujemy dekoratora Component oraz wyliczenia ViewEncapsulation.
import {Component, ViewEncapsulation} from '@angular/core';

// Za pomocą wczytywania tekstowego możemy zaimportować szablon.
import template from './app.html!text';

// W ten sposób tworzymy główną treść aplikacji.
@Component({
   // Informuje Angular, aby szukał elementu <ngc-app> w celu utworzenia tego komponentu.
   selector: 'ngc-app',
   // Wykorzystajmy treść wczytanego szablonu HTML.
   template,
   // Poinformuj Angular, aby ignorował enkapsulację widoku.
   encapsulation: ViewEncapsulation.None
})
export class App {}
```

Nie ma tu w zasadzie nic, czego o strukturze komponentu nie dowiedzielibyśmy się już w poprzednim rozdziale. Pojawiły się jednak dwa nowe elementy w porównaniu z wcześniejszym

komponentem. Właściwość `template` nie zawiera już przypisanego bezpośrednio kodu HTML szablonu zapisanego jako szablon ECMAScriptu 6. Szablon zostaje wczytany z zewnętrznego pliku jako część mechanizmu SystemJS. Aby wczytać z systemu dowolny plik, wystarczy dodać `!text` na końcu standardowej ścieżki importu ECMAScriptu 6:

```
import template from './app.html!text';
```

Powyższy kod wczytuje plik *app.html* z aktualnego folderu i umieszcza w zmiennej `template` jego zawartość jako zwykły tekst.

Druga różnica polega na zastosowaniu `ViewEncapsulation` do określenia, jak Angular 2 obsłuży enkapsulację widoku. Angular obsługuje trzy różne sposoby enkapsulacji, które zapewniają różne poziomy szczegółowości. Każdy z tych sposobów ma swoje wady i zalety. Oto krótki opis sposobów enkapsulacji:

Sposób enkapsulacji	Opis
`ViewEncapsulation.Emulated`	Jeśli komponent działa w tym trybie, Angular będzie emulował enkapsulację stylów poprzez generowanie atrybutów w komponencie, które pozwolą na zastosowanie stylów CSS tylko do tego elementu. W ten sposób uzyskuje się częściową hermetyzację, choć style zewnętrzne nadal mogą przenikać do komponentu, jeśli są stylami globalnymi. Jeśli komponent nie zmieni sposobu enkapsulacji, tryb ten będzie stosowany domyślnie.
`ViewEncapsulation.Native`	Ten sposób enkapsulacji powinien być celem hermetyzacji widoków w Angularze 2. Wykorzystuje shadow DOM (zobacz poprzedni rozdział), aby utworzyć odizolowany DOM dla całego komponentu. Ten tryb wymaga obsługi shadow DOM przez przeglądarkę internetową, więc nie zawsze może być zastosowany. Pamiętaj, że w tym trybie nie są respektowane globalne style, a style lokalne trzeba umieszczać lokalnie w znacznikach `style` lub używać właściwości `styles` adnotacji komponentu.
`ViewEncapsulation.None`	Ten tryb informuje Angular, aby nie zapewniał żadnej enkapsulacji szablonów i stylów. Oznacza to, że aplikacja będzie korzystać głównie ze stylów pochodzących z globalnego pliku CSS. Z tego trybu skorzystamy w większości komponentów prezentowanych w niniejszej książce. Ponieważ ani shadow DOM, ani style lokalne nie będą używane, użyjemy klas CSS określonych w globalnym pliku CSS.

Ponieważ komponent chce wczytać szablon z systemu plików, utwórz w folderze *lib* plik *app.html* o początkowej treści:

```
<div>Witaj, świecie!</div>
```

Na tę chwilę to wszystko, co umieścimy w szablonie. Struktura folderów po tej operacji powinna mieć następującą postać:

```
angular-2-komponenty
├── node_modules/
├── jspm_packages/
├── config.js
├── index.html
├── lib
│   ├── app.html
│   └── app.js
└── package.json
```

Po utworzeniu komponentu możemy dodać go do głównego pliku *index.html* w miejscu, w którym ma się pojawić na stronie WWW:

```
<!doctype html>
<html>
<head lang="pl">
   <meta charset="UTF-8">
   <title>Angular 2. Komponenty</title>
</head>
<body>
<ngc-app></ngc-app>
...
```

Moduł aplikacji

Angular 2 wprowadził pojęcie modułów jako elementów, które informują framework Angulara, w jaki sposób kompilować i uruchamiać kod znajdujący się wewnątrz wybranej grupy komponentów. Dodatkowo moduł to pewna bardzo wygodna jednostka modularyzacji aplikacji i łączenia grup komponentów w jedną, spójną całość. W odróżnieniu od wcześniejszej wersji Angulara obecnie wszystkie dyrektywy, komponenty i potoki muszą znajdować się w modułach. Moduł może niektóre z nich udostępnić do użycia w innych komponentach aplikacji lub innych modułów. Ponadto w module mogą pojawić się dostawcy usług wykorzystywani później do wstrzykiwania zależności (przykład pojawi się w dalszej części rozdziału).

Ponieważ wszystkie komponenty muszą znajdować się w modułach, dotyczy to również utworzonego wcześniej komponentu aplikacji. Utwórzmy główny moduł aplikacji, w którym początkowo umieścimy tylko komponent aplikacji.

W folderze *lib* utwórz plik o nazwie *app.module.js* i umieść w nim następującą treść:

```
// Zaimportuj dekorator modułu i pozostałe zależności.
import { NgModule }       from '@angular/core';
import { BrowserModule  } from '@angular/platform-browser';
import { App }   from './app';

// Utwórz moduł główny wykorzystujący komponent główny aplikacji i moduł przeglądarki.
@NgModule({
```

```
    declarations: [App],
    imports:      [BrowserModule],
    bootstrap:    [App],
})
export class AppModule {}
```

Zauważ, że moduł deklaruje, że korzysta z komponentu `App` i że jest to główny komponent aplikacji (właściwość `bootstrap`). Do prawidłowego działania modułu w środowisku aplikacji niezbędne jest jednak zaimportowanie do modułu aplikacji modułu zawierającego elementy współpracy z przeglądarką.

Kod startowy aplikacji

Plik *index.html* wczytuje plik *bootstrap.js*, który pojawia się w znaczniku `script` wykorzystującym import SystemJS. Plik *bootstrap.js* służy do wczytania wszystkich niezbędnych zależności początkowych i rozpoczęcia uruchamiania frameworku Angular 2.

Uruchamiając aplikację Angulara 2, musimy wskazać główny moduł, który odpowiada za rdzeń całej aplikacji. W tej sytuacji będzie to `AppModule`. Ponieważ działamy w przeglądarce, musimy zaimportować funkcję `platformBrowserDynamic` z jednej z bibliotek Angulara. Następnie przekazujemy moduł aplikacji jako parametr funkcji `bootstrapModule()`:

```
// Najpierw zaimportuj wymagane zależności zapewniające zgodność wsteczną.
import 'zone.js';
import 'reflect-metadata';

// Zaimportuj funkcję uruchamiania frameworku Angular.
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

// Zaimportuj główny moduł.
import {AppModule} from './app.module';

// Uruchamiamy Angular, używając głównego modułu aplikacyjnego.
platformBrowserDynamic().bootstrapModule(AppModule);
```

Pierwsze dwa importy, które pojawiły się w pliku, umożliwiają wczytanie uzupełnień API, które zapewniają zgodność wsteczną wymaganą przez framework. Bez tych poprawek aplikacja Angulara 2 nie zadziała prawidłowo. Pamiętaj, aby wczytać te biblioteki przed rozpoczęciem wczytywania jakiegokolwiek kodu związanego z aplikacją Angulara.

Uruchomienie aplikacji

Kod, który wykonaliśmy, znajduje się w stanie pozwalającym na pierwsze uruchomienie aplikacji. Zanim jednak uruchomimy `live-server`, upewnijmy się, że są gotowe wszystkie niezbędne pliki. Struktura folderów powinna wyglądać następująco:

```
angular-2-komponenty
├── jspm_packages/
├── node_modules/
├── config.js
├── index.html
├── lib
│   ├── app.html
│   ├── app.js
│   ├── app.module.js
│   └── bootstrap.js
└── package.json
```

Uruchommy serwer, który zapewnia automatyczną aktualizację kodu w przeglądarce. Do startu serwera i wyświetlenia strony w przeglądarce wystarczy jedno proste polecenie:

```
live-server
```

Jeśli wszystko zadziała prawidłowo, w przeglądarce internetowej pojawi się strona WWW z tekstem *Witaj, świecie!*.

Powtórka

Powtórzmy to, czego dowiedzieliśmy się do tej pory.

1. Zainicjalizowaliśmy nowy projekt za pomocą NPM i JSPM, a następnie użyliśmy JSPM do pobrania zależności Angulara 2.
2. Utworzyliśmy główny komponent aplikacji w *app.js* oraz towarzyszący mu główny moduł *app.module.js*.
3. Ponadto utworzyliśmy skrypt *bootstrap.js*, który odpowiada za rozpoczęcie uruchamiania frameworku Angular 2.
4. Dodaliśmy komponent do pliku *index.html*, umieszczając znacznik odpowiadający właściwości `selector` komponentu.
5. Uruchomiliśmy serwer i zobaczyliśmy pierwszy efekt swoich prac w przeglądarce internetowej.

Tworzenie listy zadań

Gdy udało się przeprowadzić konfigurację aplikacji i napisać główny komponent, możemy przystąpić do prac nad komponentami listy zadań. Drugi komponent, który utworzymy, będzie skupiał się na wyświetleniu listy zadań. Zgodnie z zasadami kompozycji komponent `task-list` wykonamy jako subkomponent głównego komponentu.

Utwórz wewnątrz folderu *lib* nowy folder o nazwie *task-list*, a w nim nowy plik *task-list.js*. Umieść w pliku następujący kod:

```
import {Component, ViewEncapsulation} from '@angular/core';
import template from './task-list.html!text';

@Component({
   selector: 'ngc-task-list',
   // Właściwość host umożliwia ustawienie niektórych właściwości
   // na elemencie HTML, który powoduje użycie komponentu.
   host: {
      class: 'task-list'
   },
   template,
   encapsulation: ViewEncapsulation.None
})
export class TaskList {
   constructor() {
      this.tasks = [
         {title: 'Zadanie 1', done: false},
         {title: 'Zadanie 2', done: true}
      ];
   }
}
```

Ten bardzo prosty komponent listy przechowuje w sobie listę zadań. Komponent zostanie dołączony do elementów HTML pasujących do selektora `ngc-task-list`.

Utwórzmy widok dla tego komponentu, aby wyświetlić zadania. Jak łatwo zauważyć, musimy utworzyć plik *task-list.html*, bo taka nazwa pojawiła się w imporcie komponentu:

```
<div *ngFor="let task of tasks" class="task">
   <input type="checkbox" [checked]="task.done">
   <div class="task__title">{{task.title}}</div>
</div>
```

Wykorzystujemy dyrektywę `NgFor` do powtórzenia elementu `<div>` o klasie `task` tyle razy, ile jest zadań na liście znajdującej się w komponencie. Dyrektywa `NgFor` powoduje utworzenie przez Angular dodatkowego szablonu, który powtórzy swoją zawartość dla tylu elementów, ile jest wpisów w pobranej tablicy. Ponieważ obecnie tablica zawiera dwa elementy, komponent `task-list` wyświetli dwa egzemplarze szablonu.

Jak wcześniej wspomniałem, komponenty Angulara muszą znajdować się w modułach. Choć moglibyśmy umieścić nowy komponent w głównym module aplikacji, warto od samego początku zastosować właściwą modularyzację i potraktować listę zadań jako osobny moduł.

Utwórz w folderze *task-list* plik *task-list.module.js* o następującej treści:

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

// Wczytaj moduł zawierający podstawowe dyrektywy.
import {CommonModule} from '@angular/common';
```

```
// Wczytaj listę zadań jako zależności modułu.
import {TaskList} from './task-list';

// Utwórz moduł listy zadań.
@NgModule({
   declarations: [TaskList],
   imports: [CommonModule],
   exports: [TaskList]
})
export class TaskListModule {}
```

Zauważ, że moduł listy zadań importuje moduł o nazwie `CommonModule`. Jest to niezbędne, aby komponenty zawarte w module mogły korzystać z dyrektyw `NgFor` i `NgIf`. Dodatkowo moduł eksportuje komponent `TaskList`. Jest to niezbędne, aby z listy zadań mógł skorzystać główny komponent aplikacji, ponieważ tylko komponenty eksportowane przez moduł są widoczne dla pozostałej części kodu aplikacji (enkapsulacja na poziomie modułów).
Struktura folderu *lib* po dodaniu nowych elementów powinna być następująca:

```
angular-2-komponenty
└── lib
    ├── app.html
    ├── app.js
    ├── app.module.js
    ├── bootstrap.js
    └── task-list
        ├── task-list.html
        ├── task-list.module.js
        └── task-list.js
```

Pozostaje jeszcze dodanie nowego modułu listy zadań do głównego modułu aplikacji. Zmodyfikuj plik *app.module.js* i dodaj na jego początku następujący wpis:

```
import {TaskListModule} from './task-list/task-list.module';
```

W tym samym pliku uzupełnij wiersz importujący modułu o następujący wpis:

```
imports: [BrowserModule, TaskListModule],
```

W ten sposób komponenty eksportowane przez moduł `TaskListModule` stają się dostępne dla komponentów obejmowanych swym zasięgiem przez główny moduł aplikacji. Oznacza to, że będzie mógł z komponentu `TaskList` skorzystać komponent `App`.

Pozostał do wykonania jeszcze jeden krok, czyli wstawienie w odpowiednim miejscu w pliku *app.html* elementu hostującego komponent `task-list`. Zamień wcześniejszą treść szablonu na poniższą:

```
<ngc-task-list></ngc-task-list>
```

To już wszystkie zmiany niezbędne do uruchomienia komponentu `task-list`. Aby sprawdzić, czy wszystko działa prawidłowo, uruchom serwer poleceniem `live-server` i sprawdź, czy przeglądarka wyświetliła listę zadań.

Powtórka

Powtórzmy, co udało nam się zrealizować w poprzednim bloku zmian. Wykonaliśmy prosty komponent wyświetlający listę zadań i przygotowaliśmy moduł hermetyzujący elementy listy.

1. Utworzyliśmy plik JavaScriptu komponentu, który zawiera główną logikę komponentu.
2. Utworzyliśmy szablon komponentu jako osobny plik HTML.
3. Utworzyliśmy moduł, który będzie zawierał wszystkie komponenty dotyczące listy zadań.
4. Dodaliśmy nowy moduł do elementów importowanych przez główny moduł aplikacji.
5. Dodaliśmy element HTML komponentu do głównego szablonu aplikacji.

Właściwy poziom enkapsulacji

Lista zadań wyświetla się poprawnie, a kod, który pozwolił zrealizować to zadanie, wygląda dobrze. Jeśli jednak poważnie traktujemy kompozycję, powinniśmy przemyśleć projekt komponentu `task-list`. Jeżeli wypiszemy na kartce obowiązki komponentu listy zadań, to będą one następujące: wyświetlenie listy, dodanie nowego zadania do listy oraz sortowanie i filtrowanie listy. Nie ma tu jednak mowy o zagadnieniach realizowanych przez pojedyncze zadanie. Oznacza to, że rendering pojedynczego zadania wykracza poza obowiązki komponentu listy zadań. Komponent powinien stanowić dla zadań jedynie kontener.

Jeśli przyjrzymy się kodowi ponownie, okaże się, że łamie zasadę pojedynczej odpowiedzialności i renderuje całe zadania jako część komponentu `task-list`. Zastanówmy się, jak możemy to naprawić poprzez zwiększenie szczegółowości enkapsulacji.

Przeprowadzimy w tym podrozdziale proste ćwiczenie z refaktoryzacji kodu nazywane ekstrakcją. Wyniesiemy szablon odpowiedzialny za pojedyncze zadanie z listy zadań do całkowicie nowego komponentu.

Utwórz w folderze *task-list* nowy podfolder o nazwie *task*, a w nim plik szablonu o nazwie *task.html*:

```
<input type="checkbox" [checked]="task.done">
<div class="task__title">{{task.title}}</div>
```

Zawartość pliku *task.html* w dużej mierze odpowiada zawartości szablonu *task-list.html*. Odnosimy się jednak w nowym kodzie do nowego modelu o nazwie `task`.

W folderze *task* utwórz nowy plik JavaScriptu o nazwie *task.js*, który będzie zawierał klasę sterującą komponentem:

```
import {Component, Input, ViewEncapsulation} from '@angular/core';
import template from './task.html!text';

@Component({
   selector: 'ngc-task',
   host: {
      class: 'task'
   },
   template,
   encapsulation: ViewEncapsulation.None
})
export class Task {
   // Model task może zostać dołączony w elemencie nadrzędnym w widoku.
   @Input() task;
}
```

W poprzednim rozdziale mówiliśmy o enkapsulacji i warunkach tworzenia czystej hermetyzacji komponentów interfejsu użytkownika. Jednym z tych warunków było określenie właściwych interfejsów wejścia i wyjścia z komponentu. Takie punkty są niezbędne, aby komponent działał poprawnie jako część kompozycji. To właśnie w ten sposób komponent otrzyma dane lub je opublikuje.

Przedstawiona implementacja komponentu zadania wykorzystuje adnotację `@Input`, aby wskazać pole klasy jako właściwość wejściową. Aby móc zastosować adnotację, musimy ją wcześniej zaimportować z głównego modułu Angulara.

Właściwości wejściowe w Angularze umożliwiają dowiązanie wyrażeń w szablonach do pól egzemplarza komponentu. Uzyskujemy w ten sposób mechanizm przekazania danych z zewnątrz komponentu do jego wnętrza przy użyciu szablonu komponentu. Z punktu widzenia komponentu jest to przykład dowiązania jednokierunkowego.

Jeśli zastosujemy dowiązanie właściwości dla zwykłego atrybutu DOM, Angular dowiąże wyrażenie bezpośrednio do właściwości DOM elementu. Właśnie ten rodzaj dowiązania wykorzystujemy dla oznaczenia zakończenia zadania — właściwość `checked` elementu `input`.

Użycie	Opis
`@Input() inputProp;`	Umożliwia dowiązanie atrybutu `inputProp` w komponencie nadrzędnym do właściwości używanego elementu. Angular zakłada, że atrybut elementu ma taką samą nazwę jak właściwość.
`@Input('inp') inputProp;`	Można zmienić nazwę atrybutu widoczną na zewnątrz komponentu. W przedstawionym przykładzie atrybut `inp` zostanie przekazany do właściwości wejściowej komponentu o nazwie `inputProp`.

Ostatni brakujący element to modyfikacja istniejącego szablonu listy zadań, aby zaczął używać nowego komponentu.

Odniesiemy się do komponentu zadania, dodając w szablonie listy znacznik `<ngc-task>`, czyli nazwę podaną w selektorze komponentu zadania. Dodamy również dowiązanie właściwości. W tym celu przekażemy obiekt `task` z aktualnej iteracji `NgFor` do wejścia `task` komponentu `task`. Zastąp aktualną zawartość pliku *task-list.html* poniższą:

```
<ngc-task *ngFor="let task of tasks"
          [task]="task"></ngc-task>
```

Aby komponent `task-list` wiedział o komponencie `task`, musimy dodać go do deklaracji dostępnych w module listy zadań. W tym celu zmodyfikuj plik *task-list.module.js* w sposób przedstawiony poniżej:

```
...
import {Task} from './task/task';

@NgModule({
   declarations: [TaskList, Task],
   ...
})
...
```

Gratulacje! Udało nam się zrefaktoryzować listę zadań i przenieść pojedyncze zadanie do własnego komponentu w celu uzyskania czystej enkapsulacji. Możemy powiedzieć, że lista zadań to kompozycja pojedynczych zadań.

Z punktu widzenia konserwacji i przyszłego wielokrotnego wykorzystania elementów tego rodzaju refaktoryzacja stanowi bardzo ważny krok w procesie budowania aplikacji. Warto cały czas poszukiwać możliwości rozbicia elementu na mniejsze komponenty, jeśli tylko taka zmiana może okazać się pomocna w przyszłości. Oczywiście nie należy też przesadzać — nie istnieje żadna złota zasada szczegółowości komponentów, więc warto kierować się zdrowym rozsądkiem.

> Odpowiednia szczegółowość enkapsulacji w architekturze komponentowej zawsze zależy od kontekstu. Moja osobista wskazówka polega na trzymaniu się zasad projektowania obiektowego, na przykład zasady pojedynczej odpowiedzialności, co pozwala właściwie zaprojektować drzewo komponentów. Zawsze upewnij się, że komponent robi tylko to, co sugeruje jego nazwa. Lista zadań odpowiada za prezentację listy i ewentualną jej filtrację. Odpowiedzialność za prezentację pojedynczego zadania i związanych z nim operacji z pewnością nie należy do komponentu listy zadań, ale do komponentu zadania.

Powtórka

W tej części rozdziału przeprowadziliśmy rozbicie jednego komponentu na dwa mniejsze, co zapewniło lepszą enkapsulację. Dodatkowo zapoznaliśmy się z dowiązaniem właściwości komponentu do argumentów wejściowych. Wykonaliśmy następujące zadania.

1. Utworzyliśmy subkomponent zadania.
2. Wykorzystaliśmy nowy subkomponent wewnątrz komponentu `task-list`.
3. Wykorzystaliśmy dowiązanie jednokierunkowe, aby przekazać dane z komponentu listy do konkretnego komponentu zadania.

Wejście generuje wyjście

Lista zadań zaczyna wyglądać coraz lepiej, ale byłaby bezużyteczna, gdyby użytkownik nie mógł dodawać do niej nowych zadań. Utwórzmy komponent zapewniający wpisywanie nowych zadań. Ponieważ komponent ten będzie stanowił część głównego komponentu listy zadań, utwórz podfolder *enter-task* w folderze *task-list*. Zadaniem komponentu będzie obsługa całej logiki interfejsu użytkownika związanej z wprowadzeniem nowego zadania.

Wykorzystajmy tę samą konwencję co w pozostałych komponentach i utwórzmy plik o nazwie *enter-task.html* do przechowywania szablonu komponentu:

```
<input type="text" class="enter-task__title-input"
       placeholder="Wpisz tytuł nowego zadania..."
       #titleInput>
<button class="button" (click)="enterTask(titleInput)">Dodaj zadanie</button>
```

Szablon składa się z pola wejściowego oraz przycisku do wpisywania nowego zadania. W szablonie korzystamy z tak zwanych zmiennych lokalnych widoku, aby nadać polu wejściowemu nazwę `#titleInput`. Po nadaniu elementowi nazwy można odnieść się do niego jako `titleInput` w innym elemencie aktualnego widoku komponentu.

W przedstawionym przykładzie wykorzystujemy ten mechanizm do przekazania pola wejściowego jako elementu DOM do funkcji `enterTask`, która zostanie uruchomiona po kliknięciu przycisku *Dodaj zadanie*.

Przyjrzyjmy się implementacji klasy komponentu odpowiedzialnej za dodanie nowego zadania, która znajdzie się w pliku *enter-task.js*:

```
import {Component, Output, ViewEncapsulation, EventEmitter} from
↪'@angular/core';
import template from './enter-task.html!text';

@Component({
   selector: 'ngc-enter-task',
   host: {
      class: 'enter-task'
   },
   template,
   encapsulation: ViewEncapsulation.None
})
```

```
export class EnterTask {
   // Emiter zdarzeń, który zostaje wywołany, gdy wpiszemy nowe zadanie.
   @Output() taskEntered = new EventEmitter();

   // Funkcja emituje zdarzenie taskEntered i czyści pole tytułu zadania.
   enterTask(titleInput) {
      this.taskEntered.next(titleInput.value);
      titleInput.value = '';
      titleInput.focus();
   }
}
```

W tym komponencie zastosowaliśmy podejście projektowe, które stara się zapewnić możliwie luźne powiązanie listy zadań z kodem odpowiadającym za zbieranie nowych zadań. Choć sam komponent jest mocno powiązany z listą zadań, sam nie zawiera żadnego jawnego odniesienia do listy.

Jedną z najprostszych form odwrócenia sterowania jest funkcja wywołania zwrotnego lub nasłuchiwanie zdarzeń. W komponencie stosujemy adnotację `@Output`, aby utworzyć emiter zdarzeń. Właściwości wyjściowe muszą być polami egzemplarza komponentu, które przechowują emiter zdarzeń. Kod dodający komponent wykorzysta reprezentujący go element HTML do dowiązania swojej logiki i rozpoczęcia nasłuchiwania zdarzeń. W ten sposób uzyskujemy ogromną elastyczność i czysty projekt aplikacji, ponieważ złączenie poszczególnych klocków następuje w widoku.

Użycie	Opis
`@Output() outputProp = ↪new EventEmitter();`	Po wykonaniu metody `outputProp.next()` zostanie wyemitowane zdarzenie powiązane z nazwą `outputProp`. Angular będzie oczekiwał dowiązania nasłuchiwania tego zdarzenia w kodzie szablonu używającym tego komponentu o postaci: <my-comp (outputProp)="doSomething()"> W wyrażeniu dotyczącym zdarzenia zawsze pojawi się dostęp do sztucznie wygenerowanej zmiennej `$event`. Zmienna ta odnosi się do danych przekazanych od obiektu `EventEmitter`.
`@Output('out') outputProp = ↪new EventEmitter();`	Zastosuj ten format, jeśli właściwości wyjściowe dostępne w komponentach nadrzędnych mają stosować inną nazwę niż wewnątrz komponentu. W przedstawionym przykładzie powiązanie z wywołaniem funkcji `outputProp.next()` wymaga zastosowania poniższego kodu w komponencie nadrzędnym: <my-comp (out)="doSomething()">

Wykorzystajmy nowy komponent do dodawania nowych zadań do komponentu `task-list`. Zmodyfikujmy istniejący szablon komponentu listy zadań. Otwórz plik *task-list.html* znajdujący się w folderze *task-list*. Musimy dodać komponent `enter-task`, a także obsłużyć zdarzenie, które będzie emitowane po wpisaniu nowego zadania w komponencie:

```
<ngc-enter-task (taskEntered)="addTask($event)">
</ngc-enter-task>
<ngc-task *ngFor="let task of tasks"
          [task]="task"></ngc-task>
```

Ponieważ właściwość wyjściowa komponentu `enter-task` nosi nazwę `taskEntered`, możemy ją dowiązać do atrybutu `(taskEntered)=""` elementu HTML w komponencie nadrzędnym.

W wyrażeniu wpisujemy nazwę funkcji komponentu `task-list` o nazwie `addTask`. Potrzebujemy również skorzystać ze zmiennej `$event`, która zawiera tytuł zadania wyemitowany z komponentu `enter-task`. Gdy klikniemy przycisk w komponencie `enter-task`, zostanie zgłoszone zdarzenie, które przechwyci komponent nadrzędny i skieruje je do funkcji `addTask`.

Musimy również dokonać pewnych drobnych zmian w napisanym w JavaScripcie kodzie komponentu `task-list`. Otwórz plik *task-list.js* i dodaj pogrubiony fragment:

```
export class TaskList {
   ...
   // Funkcja dodająca zadanie z widoku.
   addTask(title) {
      this.tasks.push({
         title, done: false
      });
   }
}
```

Dodaliśmy w kodzie komponentu funkcję `addTask()`, która dodaje nowe zadanie do listy na podstawie tytułu otrzymanego jako parametr. Zamknęliśmy w ten sposób cały krąg — zdarzenie z komponentu `enter-task` trafia do tej funkcji i powoduje dodanie zadania w komponencie `task-list`.

Ostatnia zmiana dotyczy dodania nowego komponentu do modułu listy zadań, aby komponent stał się widoczny dla innych komponentów wewnątrz modułu. Otwórz plik *task-list.module.js* i dodaj pogrubione fragmenty:

```
...
// Komponent do wpisywania nowych zadań.
import {EnterTask} from './enter-task/enter-task';

@NgModule({
   declarations: [TaskList, Task, EnterTask],
   ...
})
```

Uruchom serwer projektu i sprawdź, czy możesz dodawać nowe zadania. Zauważ, że dodane zadanie automatycznie pojawi się na liście.

Powtórka

Dodaliśmy nowy subkomponent listy zadań, który odpowiada za zapewnienie logiki interfejsu użytkownika związanej z dodawaniem nowych zadań. Wzbogaciliśmy projekt o następujące elementy.

1. Utworzyliśmy subkomponent, który jest luźno powiązany z resztą kodu właściwościami wyjściowymi i emiterami zdarzeń.
2. Poznaliśmy adnotację `@Output` i jej wykorzystanie do tworzenia właściwości wyjściowych.
3. Wykorzystaliśmy dowiązanie zdarzeń, aby powiązać zachowanie, czyli przekazać dane od widoku do komponentu.

Własne elementy interfejsu użytkownika

Standardowe elementy interfejsu użytkownika dostępne w każdej przeglądarce internetowej sprawdzają się, ale często nowoczesne aplikacje internetowe wymagają bardziej złożonych i inteligentnych elementów wejściowych niż te dostępne domyślnie.

Wykonamy dwa własne elementy interfejsu użytkownika, które wykorzystamy w dalszej części aplikacji, aby uczynić jej obsługę znacznie wygodniejszą:

- **Pole wyboru** — choć istnieje pole wyboru wbudowane w przeglądarkę, czasem trudno dostosować je do wizualnego wyglądu pozostałej części aplikacji. Wbudowane pola wyboru mają ograniczoną zdolność dostosowywania swojego wyglądu, więc niełatwo dostosować je do wymagań. Nie od dziś wiadomo, że nieraz to małe elementy powodują, iż z aplikacji korzysta się wygodnie.
- **Przyciski przełączania** — to lista przycisków przełączania, w której w danej chwili tylko jeden z przycisków może być włączony. W zwykłej przeglądarce użylibyśmy w ich miejscu przycisków opcji (ang. *radio button*). Podobnie jak z polami wyboru, ich styl wizualny nie zawsze udaje się dostosować do wymagań wyglądu strony. Lista przycisków w wielu przypadkach w dużo bardziej nowoczesny sposób reprezentuje wybór typu „jeden z listy". A tak poza tym to kto nie lubi wciskać przycisków?

Zacznijmy od wykonania pola wyboru. Ponieważ w przyszłości będziemy najprawdopodobniej wykonywać znacznie więcej elementów interfejsu użytkownika, utwórzmy w folderze *lib* folder *ui*.

W folderze *ui* utwórz folder *checkbox* dla komponentu pola wyboru. Następnie utwórz w nowym folderze plik *checkbox.html* i wstaw w nim poniższy kod:

```
<input type="checkbox"
       [checked]="checked"
       (change)="onChecke dChange($event.target.checked)">
   {{label}}
```

W szablonie zastosowaliśmy dwa dowiązana. Pierwsze z nich to dowiązanie do właściwości `checked` elementu DOM. Pole `checked` obsługujące to dowiązanie po stronie komponentu wykonamy w dalszej części rozdziału.

Dodatkowo dodaliśmy dowiązanie nasłuchiwania zdarzenia zmiany stanu pola opcji (`change`). Każde wywołanie tego zdarzenia spowoduje wywołanie funkcji `onCheckedChange` w klasie komponentu. Używamy syntetycznej zmiennej `$event`, aby przekazać nową wartość właściwości `checked` elementu, który był źródłem zdarzenia.

Wykonajmy również implementację klasy komponentu. Utwórz w folderze *checkbox* nowy plik *checkbox.js* o następującej treści:

```
import {Component, Input, Output, ViewEncapsulation, EventEmitter} from
'@angular/core';
import template from './checkbox.html!text';

@Component({
   selector: 'ngc-checkbox',
   host: {
      class: 'checkbox'
   },
   template,
   encapsulation: ViewEncapsulation.None
})
export class Checkbox {
   // Opcjonalna etykieta, którą można nadać opcji.
   @Input() label;
   // Informacja, czy opcja jest włączona czy wyłączona.
   @Input() checked;
   // Emiter zdarzeń, gdy dochodzi do zmiany stanu zaznaczenia.
   // Korzysta z konwencji odnoszącej się do dowiązania dwukierunkowego [(checked)].
   @Output() checkedChange = new EventEmitter();

   // Ta funkcja spowoduje zgłoszenie zdarzenia.
   onCheckedChange(checked) {
      this.checkedChange.next(checked);
   }
}
```

W zasadzie klasa komponentu nie zawiera żadnych specjalnych elementów, jeśli przyjrzymy się jej tylko pobieżnie. Używa właściwości wejściowej, aby umożliwić ustawienie stanu zaznaczenia z zewnątrz, a dodatkowo posiada właściwość wyjściową w postaci emitera zdarzeń, aby poinformować zewnętrzne komponenty o dokonanej zmianie stanu. Istnieje jednak pewna konwencja nazewnicza, która czyni komponent dość szczególnym. Konwencja, w której nazwa właściwości wejściowej pojawia się również w nazwie właściwości wyjściowej, ale z dodanym słowem `Change`, powoduje, że programista stosujący komponent może skorzystać ze skrótu zapewniającego dowiązanie dwukierunkowe.

Angular jako taki nie zapewnia dowiązania dwukierunkowego, ale jego wykonanie jest dziecinnie proste. W zasadzie dwukierunkowe dowiązanie danych nie różni się znacząco od połączenia dowiązania właściwości z dowiązaniem zdarzenia.

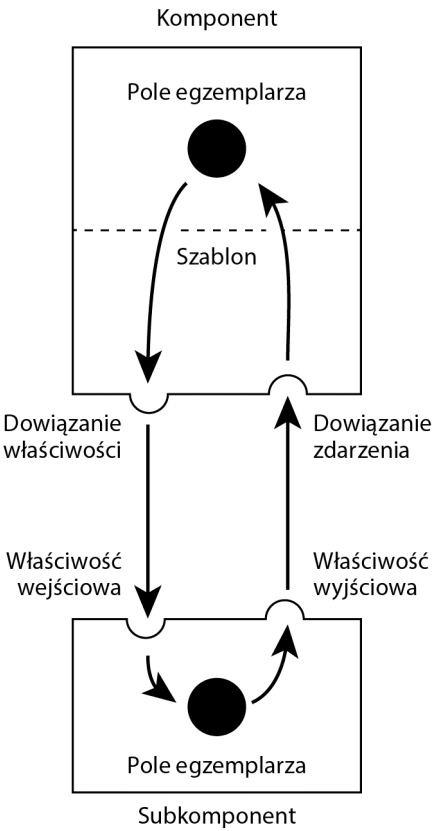
Poniższy przykład tworzy bardzo proste dowiązanie dwukierunkowe dla pola tekstowego:

```
<input type="text" (input)="value = $event.target.value"
                   [value]="value">
```

Prostota Angulara i ogólne podejście do rozszerzania wbudowanej funkcjonalności przeglądarek czyni implementację tego mechanizmu wyjątkowo prostą.

Implementacja dowiązania dwukierunkowego między komponentem i subkomponentem nie jest wcale trudniejsza. W zasadzie musimy się tylko skupić na uczestniczących w zadaniu właściwościach wejściowych i wyjściowych subkomponentu.

Przyjrzyj się poniższemu diagramowi.



Dowiązanie dwukierunkowe między polami składowymi komponentu i subkomponentem

Ponieważ dowiązanie dwukierunkowe było funkcjonalnością, której dodania do Angulara domagało się wiele osób, istnieje bardzo przydatny skrót pozwalający je uzyskać. Przyjrzyjmy się kilku przykładom, w jaki sposób zaimplementować dowiązanie danych między szablonem komponentu a subkomponentem.

Właściwości subkomponentu	Dowiązania w szablonie komponentu

`@Input() text;`
`@Output() textOut =`
`new EventEmitter();`
	<sc [text]="myText"
	(textOut)="myText = $event">
	Dowiążemy właściwość `myText` komponentu do właściwości wejściowej `text` subkomponentu. Dodatkowo wyłapujemy zdarzenie `textOut` z subkomponentu i aktualizujemy właściwość `myText`.

`@Input() text;`
`@Output() textChange =`
`new EventEmitter();`
	<sc [(text)]="myText">
	Możemy uprościć dowiązanie dwukierunkowe poprzez użycie konwencji nazewniczej, w której emiter zdarzeń ma dołączone na końcu słowo `Change`, ale poza tym nazwy są identyczne. Taki zapis pozwala zastosować w szablonie skróconą notację w postaci `[(`*właściwość*`)]`.

Jeśli przyjrzymy się implementacji komponentu `checkbox`, zauważymy dostosowanie się do konwencji pozwalającej na użycie skróconego zapisu dowiązania dwukierunkowego. W ten sposób, pisząc szablony używające komponentu `checkbox`, będziemy mogli zaoszczędzić sobie nieco pisania.

Ponieważ tworzymy nowy zestaw komponentów, będziemy potrzebowali nowego modułu Angulara. W folderze *ui* utwórz plik *ui.module.js* i umieść w nim poniższy kod:

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

// Wczytaj moduł zawierający podstawowe dyrektywy.
import {CommonModule} from '@angular/common';

// Wczytaj komponenty.
import {Checkbox} from './checkbox/checkbox';

// Utwórz moduł listy zadań.
@NgModule({
   declarations: [Checkbox],
   imports: [CommonModule],
   exports: [Checkbox]
})
export class UIModule {}
```

W przedstawionym kodzie modułu nie ma żadnych nowych elementów, ale warto pamiętać, że użycie komponentów na zewnątrz modułu wymaga podania ich również w sekcji `exports`.

Ponieważ moduł listy zadań będzie korzystał z modułu interfejsu użytkownika, nanieś w pliku *task-list/task-list.module.js* pogrubione zmiany:

```
...
// Wczytaj moduł zawierający komponenty interfejsu użytkownika.
import {UIModule} from '../ui/ui.module';
...
@NgModule({
   imports: [CommonModule, UIModule],
   ...
})
...
```

Zintegrujmy nowy komponent z komponentem zadania, zastępując obecnie stosowaną implementację zwykłego pola wyboru. Zmodyfikuj plik *task.html* znajdujący się w folderze *task-list/task*. Zastąp pole wyboru wbudowane w przeglądarkę poniższym kodem:

```
<ngc-checkbox [(checked)]="task.done"></ngc-checkbox>
```

Ponieważ importujemy moduł interfejsu użytkownika w module listy zadań, komponent `checkbox` będzie dostępny we wszystkich komponentach modułu listy. Musimy jednak dodać w pliku *task.js* dodatkową obsługę stanu zaznaczenia. Wprowadź w pliku następujące zmiany.

```
...
import {..., HostBinding} from '@angular/core';
...
export class Task {
  // Model task może zostać dołączony na elemencie nadrzędnym wewnątrz widoku.
  @Input() task;

  @HostBinding('class.task--done')
  get done() {
     return this.task && this.task.done;
  }
}
```

Już wcześniej wykorzystywaliśmy właściwość host w komponentach. Umożliwia ona zdefiniowanie dodatkowych zdarzeń lub właściwości w elemencie hostującym komponent. Element hostujący to element DOM, w którym znajduje się komponent wewnątrz komponentu nadrzędnego.

Istnieje jednak inny sposób ustawiania właściwości na elemencie hostującym komponent, który przydaje się, jeśli musimy ustawiać właściwość na podstawie danych znajdujących się w komponencie.

Adnotacja `@HostBinding` umożliwia tworzenie dowiązań właściwości w elemencie hostującym komponent na podstawie składowych komponentu. Przedstawiony kod wykorzystuje adnotację do warunkowego ustawienia klasy CSS `task--done` w elemencie HTML komponentu. W ten sposób odróżnimy wizualnie zadania zakończone od trwających.

Był to ostatni krok integrujący własny komponent `checkbox` z listą zadań. Uruchom serwer poleceniem `live-server` i pobaw się dużymi polami wyboru na liście zadań. Czy nie są one dużo przyjemniejsze do zaznaczania niż standardowe pola wyboru? Nie zaniedbujmy interfejsów użytkownika. Może to mieć bardzo pozytywny wpływ na popularność produktu.

Lista zadań po dodaniu własnego komponentu checkbox

Po zakończeniu prac nad komponentem `checkbox` możemy rozpocząć prace nad komponentem przełączania przycisków, z którego skorzystamy w następnym podrozdziale. Utwórz w folderze *ui* folder o nazwie *toggle*, a w nim plik *toggle.html* o następującej treści:

```
<button class="button button--toggle"
        *ngFor="let button of buttonList"
        [class.button--active]="button === selectedButton"
        (click)="onButtonActivate(button)">{{button}}</button>
```

Tak naprawdę nie pojawi się tu nic nowego! Powielamy przycisk kilkukrotnie za pomocą dyrektywy `NgFor` na podstawie zawartości listy `buttonList`. Lista zawiera etykiety przycisków. Warunkowo ustawiamy klasę CSS o nazwie `button--active` za pomocą dowiązania właściwości, porównując aktualny przycisk z nazwą znajdującą się w polu `selectedButton`. Kliknięcie przycisku powoduje wywołanie funkcji `onButtonActivate`, która otrzyma treść aktualnie kliknięty przycisku.

W folderze *toggle* utwórz plik *toggle.js* i umieść w nim poniższą klasę komponentu:

```
import {Component, Input, Output, ViewEncapsulation, EventEmitter}
↪from '@angular/core';
import template from './toggle.html!text';
```

```
@Component({
   selector: 'ngc-toggle',
   host: {
      class: 'toggle'
   },
   template,
   encapsulation: ViewEncapsulation.None
})
export class Toggle {
  // Lista obiektów, która zostanie wykorzystana jako wartości przycisku.
  @Input() buttonList;
  // Wejście i stan informujący, który przycisk jest wybrany, muszą odnosić się do obiektu
  // w buttonList.
  @Input() selectedButton;
  // Emiter zdarzeń po zmianie selectedButton wykorzystuje dowiązanie dwukierunkowe
  // o składni [(selected-button)].
  @Output() selectedButtonChange = new EventEmitter();

  // Funkcja zwrotna cyklu życia komponentu wywoływana po tym, jak konstruktor i dane
  // wejściowe zostały ustawione.
  ngOnInit() {
    if (this.selectedButton === undefined) {
      this.selectedButton = this.buttonList[0];
    }
  }

  // Funkcja wybierająca wybrany przycisk i emitująca zdarzenie.
  onButtonActivate(button) {
    this.selectedButton = button;
    this.selectedButtonChange.next(button);
  }
}
```

W komponencie wykorzystujemy składową `buttonList` jako tablicę obiektów, po której iterujemy w szablonie dyrektywą `NgFor`. Składowa `buttonList` jest właściwością wejściową, więc trafi to komponentu jako część danych z komponentu nadrzędnego.

Składowa `selectedButton` przechowuje aktualnie wybrany obiekt tablicy `buttonList`. Wykorzystujemy dla niej dowiązanie dwukierunkowe. W ten sposób możemy nie tylko zmodyfikować aktualne zaznaczenie z komponentu, ale również poinformować świat zewnętrzny o wybraniu innego przycisku przez użytkownika.

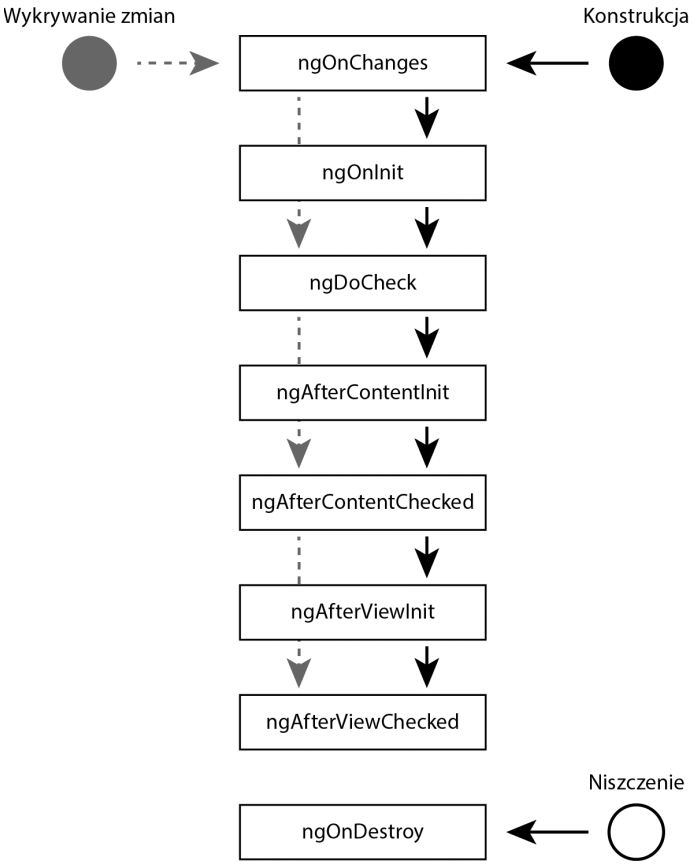
Wewnątrz funkcji `onButtonActivate` ustawiamy składową `selectedButton` i zgłaszamy zdarzenie zmiany.

Metodę `ngOnInit` wywoła Angular, bo stanowi ona jedną z metod cyklu życia dyrektyw i komponentów. W sytuacji, gdy zmienna wejściowa `selectedButton` nie została ustawiona, ustawiamy domyślne zaznaczenie na pierwszy przycisk z listy. Ponieważ zarówno `selectedButton`, jak i `button`

↪`List` są właściwościami wejściowymi, musimy zaczekać na ich inicjalizację, zanim uruchomimy wspomnianą logikę. Jest niezwykle ważne, aby nie uruchamiać inicjalizacji parametrów wejściowych w konstruktorze komponentu. Metoda cyklu życia `ngOnInit()` zostanie uruchomiona dopiero po podpięciu wszystkich właściwości wejściowych i wyjściowych, a ta sytuacja ma miejsce dopiero po skonstruowaniu dyrektywy.

> Jeśli zdefiniujemy w komponencie dowolną z metod cyklu życia komponentu, Angular wywoła ją automatycznie.

Przedstawiony poniżej diagram ilustruje cykl życia komponentu Angulara. Przy tworzeniu komponentu wszystkie metody zostaną wywołane w kolejności wskazanej w diagramie poza metodą `ngOnDestroy`, która zostanie wywołana w momencie niszczenia komponentu.



Ilustracja cyklu życia komponentu Angulara

Wykrywanie zmian również powoduje uruchomienie części metod cyklu życia. Zostaną uruchomione przynajmniej dwie z wymienionych poniżej metod, ale zawsze w przedstawionej tu kolejności:

- `ngDoCheck`,
- `ngAfterContentChecked`,
- `ngAfterViewChecked`,
- `ngOnChanges` (jeśli zostaną wykryte jakiekolwiek zmiany).

Szczegółowy opis wszystkich metod cyklu życia komponentu znajduje się w dokumentacji frameworku Angular dostępnej na stronie *https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html*.

Pamiętaj o dodaniu komponentu `toggle` do modułu interfejsu użytkownika znajdującego się w pliku *ui.module.js* w sekcjach `declarations` i `exports`.

Powtórka

W tej części wyjaśniłem, jak budować własne komponenty interfejsu użytkownika, które są ogólne i luźno powiązane z pozostałą częścią aplikacji, więc mogą być stosowane w innych komponentach jako subkomponenty. Wykonaliśmy w tym podrozdziale następujące zadania:

1. Utworzyliśmy subkomponent, który jest luźno powiązany z resztą aplikacji dzięki właściwościom zewnętrznym i emiterom zdarzeń.
2. Dowiedzieliśmy się, czym jest adnotacja `@Output` i jak wykorzystać ją do tworzenia właściwości zewnętrznych.
3. Wykorzystaliśmy adnotację `@HostBinding` do utworzenia w sposób deklaratywny właściwości w elemencie HTML hostującym komponent.
4. Wykorzystaliśmy dowiązanie zdarzeń, aby przekazać dane z widoku do komponentu.
5. Wykonaliśmy dowiązanie dwukierunkowe, wykorzystując notację skróconą.
6. Poznaliśmy cykl życia komponentu Angulara i wykorzystaliśmy metodę cyklu życia `ngOnInit` do inicjalizacji komponentu po otrzymaniu przez komponent właściwości wejściowych i wyjściowych.

Filtrowanie zadań

To ostatnie zadanie w tym rozdziale. Poznaliśmy wiele elementów Angulara 2, tworząc proste komponenty i łącząc je w większe elementy. W poprzednim podrozdziale wykonaliśmy ogólny komponent interfejsu użytkownika, który można zastosować w wielu różnych projektach. Wykorzystamy okazję i nie tylko zastosujemy komponent przycisków przełączania do filtrowania zadań, ale również wprowadzimy pojęcie usługi danych.

Rozpocznijmy od dodatkowego ćwiczenia z refaktoryzacji. Do tej pory przechowywaliśmy dane listy zadań bezpośrednio w komponencie `task-list`. Zmieńmy tę sytuację, bo rzadko będzie odpowiadała rzeczywistości, i skorzystajmy z usługi danych.

Zastosowana usługa nie będzie korzystała z bazy danych — po prostu będzie niezależnym źródłem danych o zadaniach. Dodatkowo w trakcie realizacji zadania po raz pierwszy skorzystamy z mechanizmu wstrzykiwania zależności Angulara 2.

Utwórz nowy plik o nazwie *task-list-service.js* w folderze *lib/task-list* i umieść w nim następującą treść:

```
// Klasy, które mają zostać udostępnione dla wstrzykiwania zależności, muszą być
// oznaczone tym dekoratorem.
import {Injectable} from '@angular/core';

@Injectable()
export class TaskListService {
   constructor() {
      this.tasks = [
         {title: 'Zadanie 1', done: false},
         {title: 'Zadanie 2', done: false},
         {title: 'Zadanie 3', done: true},
         {title: 'Zadanie 4', done: false}
      ];
   }
}
```

Wszystkie zadania zostały przeniesione do nowej usługi. Aby możliwe było wstrzykiwanie klasy usługi, musimy udekorować ją adnotacją `@Injectable`.

Wprowadźmy pewne zmiany do komponentu `task-list`, modyfikując plik *task-list.js* znajdujący się w folderze *task-list*. Zmodyfikowane fragmenty kodu zostały pogrubione:

```
import {..., Inject} from '@angular/core';

// Tymczasowy serwis, z którego będziemy pobierali zadania.
import {TaskListService} from './task-list-service';

@Component({
  ...
  // Ustaw TaskListService jako dostawcę.
  providers: [TaskListService]
})
export class TaskList {
   // Wstrzyknij TaskListService i określ dane do filtrowania.
   constructor(@Inject(TaskListService) taskListService) {
      this.taskListService = taskListService;
      this.taskFilterList = ['wszystkie', 'otwarte', 'wykonane'];
      this.selectedTaskFilter = 'wszystkie';
   }
```

```
    // Metoda zwraca przefiltrowaną listę zadań na podstawie wybranego rodzaju filtrowania.
    getFilteredTasks() {
      return this.taskListService.tasks ?
this.taskListService.tasks.filter((task) => {
        if (this.selectedTaskFilter === 'wszystkie') {
          return true;
        } else if (this.selectedTaskFilter === 'otwarte') {
          return !task.done;
        } else {
          return task.done;
        }
      }) : [];
    }

    // Funkcja dodająca zadanie z widoku.
    addTask(title) {
      this.taskListService.tasks.push({
        title,
        done: false
      });
    }
}
```

W części dotyczącej importu pobieramy usługę listy zadań. Wstrzykiwanie zależności posłuży do faktycznego pobrania egzemplarza klasy `TaskListService` i przekazania go do konstruktora komponentu. Wykorzystujemy w tym celu nową adnotację, która pozwala określić typ wstrzykiwanej zależności. Dekorator `Inject` importujemy z głównego modułu Angulara, aby móc go użyć w konstrukcji `@Inject`. Zauważ, że w przedstawionym powyżej kodzie przekazujemy do `@Inject` typ obiektu, który chcemy wstrzyknąć.

Poza użyciem adnotacji `@Inject` w konstruktorze klasy musimy wykonać jeszcze jedno zadanie, aby zapewnić prawidłowe działanie wstrzykiwania — niezbędna jest rejestracja `TaskListService` jako dostawcy usług za pomocą właściwości `providers` adnotacji `@Component`.

Utworzyliśmy kod, który otrzyma wstrzyknięty `TaskListService` w momencie konstruowania dyrektywy. Referencję do niego możemy przechować we właściwości egzemplarza klasy.

W konstruktorze klasy chcemy również określić listę stanów możliwych do użycia jako filtr. Lista ta posłuży też jako parametr wejściowy listy przełączników. Przypomnę, że właściwość wejściowa komponentu listy przycisków akceptowała element `buttonList` przyjmujący etykiet. Aby zapamiętać aktualnie wybrany filtr, użyjemy pola egzemplarza o nazwie `selectedTaskFilter`.

Ostatnim elementem, który dodamy do komponentu `task-list`, jest metoda `getFilteredTasks()`. Zamiast przechowywać pełną listę w polu egzemplarza, możemy pobierać ją na żądanie przy użyciu metody komponentu. Logika wewnątrz metody sprawdza właściwość `selectedTaskFilter` i zwraca listę przefiltrowaną według wybranego kryterium.

To wszystko, co chcieliśmy zmienić w implementacji komponentu. Musimy jeszcze dodać listę filtrów, odpowiednio na nią reagować i pobierać przefiltrowaną listę zadań. W tym celu otwórz plik *task-list.html* w folderze *task-list* i zmodyfikuj go zgodnie z poniższą wersją:

```
<ngc-toggle [buttonList]="taskFilterList"
            [(selectedButton)]="selectedTaskFilter">
</ngc-toggle>
<ngc-enter-task (taskEntered)="addTask($event)">
</ngc-enter-task>
<ngc-task *ngFor="let task of getFilteredTasks()"
          [task]="task"></ngc-task>
```

Ponieważ dodaliśmy wcześniej komponent `toggle` do listy komponentów eksportowanych przez moduł komponentów interfejsu użytkownika, możemy go bez przeszkód użyć w komponencie. Dowiązujemy właściwość wejściową `buttonList` do listy `taskFilterList` przechowywanej w komponencie. Wykorzystujemy również dowiązanie dwukierunkowe, aby powiązać właściwość wejściową `selectedButton` z polem `selectedTaskFilter` listy zadań. W ten sposób możemy nie tylko ustawić wartość początkową listy z poziomu komponentu listy zadań, ale również otrzymać informację o zmianie wartości dokonanej na liście przycisków.

Pozostaje jeszcze drobna zmiana w zawartości dyrektywy `NgFor`, aby pobierać przefiltrowaną listę zadań. Ponieważ listę przefiltrowanych zadań udostępnia metoda `getFilteredTasks()`, to właśnie jej używamy do pobrania aktualnej listy zadań komponentu `task-list`.

I to wszystko! Pomyślnie dodaliśmy filtrowanie listy zadań i wykorzystaliśmy komponent przełącznika wykonany w poprzednim podrozdziale. Uruchom serwer poleceniem `live-server` i zobacz w pełni działającą listę zadań umożliwiającą dodawanie nowych zadań i filtrowanie istniejących.

Zrzut ekranu aplikacji z listą zadań zapewniającą dodawanie nowych zadań i ich filtrowanie

Podsumowanie

W tym rozdziale omówiłem wiele podstawowych koncepcji związanych z budowaniem komponentów interfejsu użytkownika w aplikacji Angulara. Dodatkowo utworzyliśmy główny komponent aplikacji do zarządzania zadaniami, czyli listę zadań. Przedstawiłem koncepcję właściwości wejściowych i wyjściowych oraz wykorzystanie ich do budowania dowiązań dwukierunkowych.

Przedstawiłem również podstawy cyklu życia komponentu Angulara, a w szczególności możliwość podpięcia się pod krok występujący tuż po wstępnej inicjalizacji komponentu.

Jako ostatni krok zintegrowaliśmy komponent listy przycisków z główną listą zadań, aby uzyskać mechanizm filtrowania. Ponadto przerobiliśmy komponent `task-list` w taki sposób, aby używał dodatkowej usługi do pobierania listy zadań. Wymagało to zastosowania mechanizmu wstrzykiwania zależności.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

Z

Angular 2

Tworzenie interaktywnych aplikacji internetowych

Wykorzystywanie komponentów do budowy aplikacji internetowych jest uważane za wyjątkowo ważny krok naprzód w tej dziedzinie. Szczególnie ciekawym pomysłem jest tworzenie interfejsów użytkownika bazujących na komponentach. Framework Angular 2 zmienia technologię budowania aplikacji: ułatwia pisanie współdzielonych bloków kodu HTML, które można bez problemu wielokrotnie wykorzystywać dzięki zastosowaniu mechanizmu shadow DOM. Jest to bardzo obiecująca perspektywa pod warunkiem, że programista potrafi efektywnie użyć architektury komponentowej.

Niniejsza książka ma na celu nauczenie programistów tego nowego podejścia do programowania interaktywnych stron internetowych i równocześnie pokazanie najlepszych możliwości Angulara. Poza praktycznymi aspektami korzystania z tego frameworka największy nacisk położono na tworzenie interfejsów użytkownika bazujących na komponentach — wyjaśniono podstawowe koncepcje tego sposobu programowania i opisano, jak użyć frameworka do budowania nowoczesnych, wydajnych i łatwych w utrzymaniu interfejsów użytkownika.

Architektura oparta na komponentach — najlepsza recepta na świetny interfejs!

W książce znajdziesz:

- podstawy tworzenia interfejsów z wykorzystaniem komponentów
- tworzenie komponentów wielokrotnego użytku w Angularze
- wykorzystanie komponentów do routingu, logowania i śledzenia czasu
- korzystanie z zewnętrznych bibliotek w Angularze
- zapewnienie rozszerzalności komponentów

Gion Kunz od wielu lat pisze interaktywne interfejsy użytkownika w języku JavaScript. Bardzo chętnie korzysta z frameworka Angular 2. Często zabiera głos na konferencjach. Jest również głównym instruktorem w SAE Institute w Zurychu. Angażuje się na rzecz *open source* — napisał responsywną bibliotekę do rysowania wykresów Chartist. W wolnych chwilach zajmuje się muzyką, wędkowaniem lub po prostu spędza czas ze swoją ukochaną i małym pieskiem.

[PACKT] open source PUBLISHING community experience distilled

Helion

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-3196-9

cena: 59,00 zł