

Nathan Rozentals

Język TypeScript

Tajniki kodu

Wydanie II

Helion 

Packt 

Tytuł oryginału: Mastering TypeScript, Second Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-3641-4

Copyright © Packt Publishing 2017.

First published in the English language under the title 'Mastering TypeScript - Second Edition - (9781786468710)'

Polish edition copyright © 2017 by Helion SA
All rights reserved.

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 jakiegokolwiek 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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/jeztyp.zip>

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorze	15
O recenzentach	17
Wstęp	19
Rozdział 1. TypeScript — dostępne narzędzia i frameworki	25
Przedstawienie języka TypeScript	27
Standard EcmaScript	27
Zalety języka TypeScript	28
Zintegrowane środowiska programistyczne obsługujące język TypeScript	36
Kompilacja w środowisku Node	36
Visual Studio 2017	38
WebStorm	44
Visual Studio Code	48
Inne edytory	55
Podsumowanie	57
Rozdział 2. Typy, zmienne oraz funkcje	59
Typy podstawowe	60
Typy w języku JavaScript	60
Określanie typów w języku TypeScript	61
Składnia określania typów	62
Wnioskowanie typów	64
Kacze typowanie	65
Łańcuchy szablonów	66
Tablice	67
Pętle for...in oraz for...of	68
Typ any	69
Jawne rzutowanie	70
Typy wyliczeniowe	71
Ustalone typy wyliczeniowe	73

Stałe	74
Słowo kluczowe let	75
Funkcje	77
Typy wartości zwracanych przez funkcje	77
Funkcje anonimowe	78
Parametry opcjonalne	78
Parametry domyślne	80
Parametr reszty	80
Funkcje zwrotne	82
Sygnatury funkcji	84
Przeciążanie funkcji	86
Zaawansowane typy	87
Typ unii	88
Strażniki typów	88
Nazwy zastępcze typów	90
Wartość null i undefined	91
Reszta obiektu i rozproszenie	93
Podsumowanie	94
Rozdział 3. Interfejsy, klasy i dziedziczenie	95
Interfejsy	96
Właściwości opcjonalne	97
Kompilacja interfejsów	98
Klasy	98
Właściwości klas	99
Implementacja interfejsów	100
Konstruktory klas	101
Funkcje klas	102
Definicje funkcji interfejsów	105
Modyfikatory klas	106
Modyfikatory dostępu w konstruktorach	108
Właściwości tylko do odczytu	109
Akcesory właściwości klas	109
Funkcje statyczne	111
Właściwości statyczne	111
Przestrzenie nazw	112
Dziedziczenie	113
Dziedziczenie interfejsów	114
Dziedziczenie klas	114
Słowo kluczowe super	115
Przeciążanie funkcji	116
Składowe chronione	117
Klasy abstrakcyjne	118
Domknięcia JavaScript	121
Stosowanie interfejsów, klas i dziedziczenia — wzorzec projektowy Fabryka	123
Wymagania biznesowe	123
Co robi wzorzec Fabryka?	123
Stosowanie klasy fabrykującej	127
Podsumowanie	128

Rozdział 4. Dekoratory, typy ogólne i asynchroniczność	129
Dekoratory	130
Składnia dekoratorów	131
Stosowanie wielu dekoratorów	132
Fabryki dekoratorów	132
Parametry dekoratorów klas	133
Dekoratory właściwości	135
Dekoratory właściwości statycznych	136
Dekoratory metod	137
Stosowanie dekoratorów metod	138
Dekoratory parametrów	139
Metadane dekoratorów	140
Stosowanie metadanych dekoratorów	142
Typy ogólne	143
Składnia typów ogólnych	144
Tworzenie instancji klas ogólnych	145
Stosowanie typu T	146
Ograniczenia typu T	148
Interfejsy ogólne	150
Tworzenie nowych obiektów w klasach ogólnych	151
Mechanizmy programowania asynchronicznego	153
Obietnice	153
Składnia obietnic	155
Stosowanie obietnic	156
Składnia funkcji zwrotnych a składnia obietnic	158
Zwracanie wartości z obietnic	158
Słowa kluczowe async i await	160
Słowo kluczowe await a obsługa błędów	161
Obietnice a składnia słowa kluczowego await	162
Komunikaty a słowo kluczowe await	163
Podsumowanie	165
Rozdział 5. Pisanie i stosowanie plików deklaracji	167
Zmienne globalne	168
Stosowanie bloków kodu JavaScript w kodzie HTML	170
Dane strukturalne	171
Pisanie własnych plików deklaracji	173
Słowo kluczowe module	175
Interfejsy	177
Typy unii	179
Scalanie modułów	180
Składnia plików deklaracji	180
Przestanianie funkcji	181
Zagnieżdżone przestrzenie nazw	181
Klasy	182
Przestrzenie nazw klas	182
Przeciążanie konstruktora klas	182

Właściwości klas	183
Funkcje klas	183
Właściwości i funkcje statyczne	183
Funkcje globalne	184
Sygnatury funkcji	184
Właściwości opcjonalne	184
Scalanie funkcji i modułów	185
Podsumowanie	185
Rozdział 6. Biblioteki innych twórców	187
Pobieranie plików definicji	188
Stosowanie NuGet	190
Stosowanie menedżera pakietów	190
Instalowanie plików deklaracji	191
Stosowanie konsoli menedżera pakietów	191
Stosowanie narzędzia Typings	192
Poszukiwanie pakietów	193
Inicjalizacja Typings	194
Instalowanie plików definicji	194
Instalowanie konkretnej wersji pliku	195
Ponowna instalacja plików definicji	195
Stosowanie programu Bower	196
Stosowanie npm i @types	196
Stosowanie bibliotek innych twórców	197
Wybór frameworka JavaScript	197
Backbone	198
Stosowanie dziedziczenia we frameworku Backbone	199
Stosowanie interfejsów	201
Stosowanie składni typów ogólnych	201
Stosowanie języka ECMAScript 5	202
Zgodność frameworka Backbone z językiem TypeScript	203
Angular	203
Klasy Angular i zmienna \$scope	205
Zgodność frameworka AngularJS z językiem TypeScript	207
Dziedziczenie — Angular kontra Backbone	207
ExtJS	208
Tworzenie klas w ExtJS	208
Stosowanie rzutowania typów	210
Kompilator TypeScript dla ExtJS	211
Podsumowanie	211
Rozdział 7. Frameworki zgodne z językiem TypeScript	213
Czym jest MVC?	214
Model	215
Widok	215
Kontroler	216
Podsumowanie wzorca MVC	217

Zalety stosowania wzorca MVC	218
Szkielet przykładowej aplikacji	219
Stosowanie frameworka Backbone	220
Wydajność wyświetlania	221
Konfiguracja frameworka Backbone	222
Modele Backbone	223
Widok itemView	224
Widok collectionView	225
Aplikacja Backbone	227
Stosowanie frameworka Aurelia	228
Konfiguracja frameworka Aurelia	228
Zagadnienia do rozważenia	229
Wydajność frameworka Aurelia	230
Modele frameworka Aurelia	230
Widoki frameworka Aurelia	231
Wczytywanie aplikacji Aurelia	232
Zdarzenia frameworka Aurelia	233
Framework Angular 2	234
Konfiguracja frameworka Angular 2	234
Modele frameworka Angular 2	235
Widoki Angular 2	235
Wydajność frameworka Angular 2	237
Zdarzenia Angular 2	237
Stosowanie frameworka React	238
Konfiguracja frameworka React	238
Widoki React	240
Wczytywanie aplikacji React	243
Zdarzenia React	245
Podsumowanie	246
Rozdział 8. Programowanie w oparciu o testy	247
Programowanie w oparciu o testy	248
Testy jednostkowe, integracyjne oraz akceptacyjne	249
Testy jednostkowe	250
Testy integracyjne	250
Testy akceptacyjne	250
Frameworki testów jednostkowych	251
Jasmine	251
Prosty test Jasmine	252
Plik SpecRunner	252
Obiekty dopasowujące	254
Uruchamianie i kończenie testów	256
Testy bazujące na danych	256
Stosowanie szpiegów	258
Szpiegowanie funkcji zwrotnych	259
Stosowanie szpiegów jako imitacji	261
Testy asynchroniczne	261

Stosowanie funkcji done()	263
Modyfikacje DOM w Jasmine	264
Zdarzenia DOM	265
Mechanizmy wykonawcze Jasmine	266
Testem	267
Karma	268
Protractor	270
Stosowanie ciągłej integracji	272
Zalety ciągłej integracji	273
Wybór serwera budowy	274
Raportowanie testów integracyjnych	275
Podsumowanie	276
Rozdział 9. Testowanie frameworków zgodnych z językiem TypeScript	277
Testowanie naszej aplikacji przykładowej	278
Modyfikacja aplikacji przykładowej w celu umożliwienia testowania	279
Testowanie frameworka Backbone	280
Złożone modele	280
Aktualizacje widoków	283
Modyfikacje obsługi zdarzeń DOM	283
Testy modeli	285
Testy modelu złożonego	287
Testy wyświetlania	288
Testy zdarzeń DOM	289
Podsumowanie testów Backbone	291
Testowanie frameworka Aurelia	291
Komponenty frameworka Aurelia	291
Model widoku komponentów Aurelia	292
Komponent widoku frameworka Aurelia	293
Wyświetlanie komponentu	293
Konwencje nazewnictwa frameworka Aurelia	294
Konfiguracja testów we frameworku Aurelia	295
Testy jednostkowe we frameworku Aurelia	296
Testy wyświetlania	297
Testy przekrojowe we frameworku Aurelia	299
Podsumowanie testów frameworka Aurelia	303
Testowanie frameworka Angular 2	303
Aktualizacja aplikacji	303
Konfiguracja testów we frameworku Angular 2	305
Testy modelu w Angular 2	305
Testy wyświetlania w Angular 2	306
Testowanie DOM w Angular 2	307
Podsumowanie testów Angular 2	308
Testowanie frameworka React	308
Wiele punktów wejścia	309
Modyfikacje w aplikacji React	309
Testy jednostkowe komponentów React	312

Testy modelu i widoków React	313
Testy zdarzeń we frameworku React	315
Podsumowanie	316
Rozdział 10. Modularyzacja	317
Podstawowe informacje o modułach	318
Eksportowanie modułów	320
Importowanie modułów	320
Zmiana nazwy modułu	321
Eksporty domyślne	322
Eksportowanie zmiennych	323
Wczytywanie modułów AMD	324
Kompilacja modułów AMD	324
Konfiguracja modułów AMD	326
Konfiguracja Require	326
Konfiguracja przeglądarki dla modułów AMD	327
Zależności w modułach AMD	328
Wczytywanie frameworka Require	331
Poprawianie błędów konfiguracji Require	333
Wczytywanie modułów przy użyciu SystemJS	334
Instalacja SystemJS	335
Konfiguracja SystemJS do użycia w przeglądarce	335
SystemJS i zależności modułów	337
Wczytywanie Jasmine	340
Stosowanie Express i Node	340
Konfiguracja Express	341
Stosowanie modułów w aplikacjach Express	342
Określanie i obsługa tras w aplikacjach Express	344
Stosowanie szablonów w Express	346
Stosowanie Handlebars	347
Zdarzenia POST we frameworku Express	350
Przekierowywanie żądań HTTP	353
Podsumowanie informacji o Node i Express	355
Podsumowanie	355
Rozdział 11. Programowanie obiektowe	357
Zasady programowania obiektowego	358
Programowanie w oparciu o interfejsy	358
Zasady SOLID	359
Projektowanie interfejsu użytkownika	360
Projekt koncepcyjny	361
Konfiguracja aplikacji Angular 2	363
Stosowanie frameworka Bootstrap	364
Tworzenie panelu bocznego	365
Tworzenie nakładki	369
Koordynacja efektów przejść	371

Wzorzec Stan	372
Interfejs stanu	372
Konkretne stany	373
Wzorzec Mediator	374
Modularny kod	375
Komponent NavBar	376
Komponent SideNav	377
Komponent RightScreen	378
Komponenty podrzędne	380
Implementacja interfejsu mediatora	381
Klasa Mediator	382
Stosowanie klasy Mediator	385
Reagowanie na zdarzenia DOM	386
Podsumowanie	387
Rozdział 12. Wstrzykiwanie zależności	389
Wysyłanie poczty elektronicznej	390
Wykorzystanie pakietu nodemailer	390
Ustawienia konfiguracyjne	393
Stosowanie lokalnego serwera SMTP	395
Wstrzykiwanie zależności	396
Wzorzec Lokalizacja usługi	396
Lokalizacja usługi — antywzorzec	398
Wstrzykiwanie zależności	399
Implementacja wstrzykiwania zależności	399
Wyznaczanie interfejsów	399
Wyznaczanie z użyciem typów wyliczeniowych	400
Wyznaczanie nazwy klasy	401
Wstrzykiwanie przy użyciu konstruktora	402
Wstrzykiwanie przy użyciu dekoratora	403
Stosowanie definicji klasy	404
Analiza parametrów konstruktora	405
Określanie typów parametrów	406
Wstrzykiwanie właściwości	407
Stosowanie wstrzykiwania zależności	408
Wstrzykiwanie rekurencyjne	409
Podsumowanie	410
Rozdział 13. Tworzenie aplikacji	411
Interfejs użytkownika	412
Stosowanie edytora Brackets	413
Stosowanie rozszerzenia Emmet	415
Tworzenie panelu logowania	417
Witryna korzystająca z frameworka Aurelia	419
Kompilacja Node i frameworka Aurelia	420
Udostępnianie aplikacji Aurelia	421
Strony frameworka Aurelia w aplikacji Express	422

Komponenty aplikacji Aurelia	424
Przetwarzanie danych JSON	425
Formularze we frameworku Aurelia	427
Przesyłanie danych	430
Stosowanie komunikatów we frameworku Aurelia	431
Witryna Angular 2	434
Konfiguracja Angular 2	434
Udostępnianie stron aplikacji Angular 2	434
Komponenty aplikacji Angular 2	437
Przetwarzanie danych JSON	440
Przesyłanie danych do aplikacji	442
Witryna Express i React	443
Express i React	443
Udostępnianie aplikacji React	445
Większa liczba plików package.json	447
Komponenty React	448
Korzystanie z punktów końcowych REST	451
Komponent panelu logowania	452
Wiązanie danych w aplikacji React	454
Przesyłanie danych JSON w żądaniach POST	456
Podsumowanie	457
Rozdział 14. Czas zakasać rękawy!	459
Aplikacja SurfDechy	460
Wyjściowa aplikacja Angular 2	461
Testy jednostkowe	463
Testy implementacji wzorców projektowych	464
Stan panelu logowania	468
Integracja paneli	471
Struktura danych JSON	474
Komponent BoardList	476
Testy jednostkowe żądań HTTP	477
Wyświetlanie listy desek	483
Testowanie zdarzeń interfejsu użytkownika	484
Widok szczegółów deski	487
Stosowanie filtra	489
Panel logowania	492
Architektura aplikacji	496
Podsumowanie	497
Skorowidz	499

Programowanie obiektowe

W 1995 roku tak zwany *Gang Czwórka* (ang. *Gang of Four*, w skrócie *GoF*) opublikował książkę pt. *Design Patterns: Elements of Reusable Object-Oriented Software*¹. Autorzy tej książki — Erich Gamma, Richard Helm, Ralph Johnson oraz John Vlissides — opisali w niej szereg klasycznych programowych wzorców projektowych, stanowiących proste i eleganckie rozwiązania powszechnie występujących problemów programistycznych. Jeśli nigdy nie słyszałeś o wzorcach projektowych, takich jak Fabryka (ang. *Factory*), Kompozyt (ang. *Composite*), Obserwator (ang. *Observer*) czy też Singleton, gorąco zachęcam do przeczytania tej książki.

Wzorce projektowe zaprezentowane przez Gang Czwórka zostały zaimplementowane w wielu różnych językach programowania, także w Javie i C#. Vilić Vane napisał książkę pt. *TypeScript Design Patterns*, w której każdy z wzorców przedstawionych przez Gang Czwórka został zaimplementowany i opisany z perspektywy języka TypeScript. W trzecim rozdziale tej książki, „Interfejsy, klasy i dziedziczenie”, poświęciłem nieco czasu na zaimplementowanie klasycznego wzorca projektowego Fabryka, będącego jednym z bardziej popularnych wzorców opisanych w książce autorstwa Gangu Czwórka. TypeScript, wraz ze swoimi konstrukcjami zgodnymi ze standardami ES6 i ES7, jest doskonałym przykładem obiektowego języka programowania. Ze swoimi możliwościami tworzenia klas, klas abstrakcyjnych, interfejsów, z dziedziczeniem oraz typami ogólnymi aplikacje pisane w języku TypeScript mogą już w pełni korzystać ze wzorców projektowych opisanych w książce Gangu Czwórka.

Przedstawienie implementacji każdego z wzorców projektowych opisanych przez Gang Czwórka napisanej w języku TypeScript jest zagadnieniem daleko wykraczającym poza ramy jednego

¹ Polskie tłumaczenie tej książki zostało wydane nakładem wydawnictwa Helion, pt. *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku — przyp. tłum.*

rozdziału, poza tym byłoby to niesprawiedliwe względem wspaniałej prezentacji tych wzorców napisanej przez Vilica Vane'a. Dlatego też w tym rozdziale skoncentrujemy się na procesie pisania kodu obiektowego, przedstawionym na przykładzie dwóch wzorców projektowych, które doskonale współpracują i ułatwiają obsługę złożonych interfejsów użytkownika. Wzorcami tymi są **Stan** (ang. *State*) oraz **Mediator** (ang. *Mediator*), które koncentrują się na stanie aplikacji oraz sposobach wzajemnej interakcji pomiędzy obiektami. W tym rozdziale utworzymy aplikację Angular 2 dysponującą raczej złożonym interfejsem użytkownika i korzystającą z wyszukanych efektów przejść CSS. Następnie rozpoczniemy proces przepisywania tej początkowej wersji aplikacji tak, by korzystała z zasad programowania obiektowego, a przy okazji poznamy wzajemne interakcje pomiędzy tworzącymi ją obiektami. W końcu, zaimplementujemy wzorce projektowe Stan i Mediator, aby hermetyzować logikę odpowiedzialną za określanie, które elementy interfejsu użytkownika mają być w danej chwili widoczne, a które nie.

W tym rozdziale zostaną opisane następujące zagadnienia.

- Zasady programowania obiektowego.
- Stosowanie interfejsów.
- Zasady **SOLID**.
- Projektowanie interfejsów użytkownika.
- Wzorzec Stan.
- Wzorzec Mediator.
- Kod modularny.

Zasady programowania obiektowego

Każdą pisaną aplikację należy oceniać pod kątem najlepszych praktyk programowania obiektowego. Robert Martin opublikował tak zwane zasady projektowe SOLID, przy czym nazwa ta jest akronimem pięciu praktyk programowania obiektowego. Postępując zgodnie z tymi zasadami, łatwiej sprawimy, że nasz kod będzie: prosty do utrzymania, łatwy do zrozumienia i rozszerzania, jak również odporny na zmiany. W naszym błyskawicznie rozwijającym się świecie zazwyczaj nie możemy pozwolić sobie na luksus poświęcania dużych ilości czasu na modyfikowanie aplikacji i zaspokajanie nieustannie zmieniających się wymagań. Im szybciej będziemy w stanie dostarczać aktualizacje spełniające wymagania biznesowe, tym większa szansa, że uda się wyprzedzić konkurencję. Postępowanie zgodnie z zasadami SOLID będzie stanowić doskonałą podstawę, która zapewni możliwość łatwego wprowadzania zmian w istniejącym kodzie, niezbędną do zaspokojenia błyskawicznie zmieniających się wymagań stawianych przed naszym kodem.

Programowanie w oparciu o interfejsy

Jednym z podstawowych założeń wykorzystywanych przez Gang Czworoga jest to, że programiści *powinni programować w oparciu o interfejsy, a nie implementacje*. Oznacza to, że programy

powinny być tworzone w taki sposób, by wszystkie interakcje pomiędzy obiektami były realizowane przy wykorzystaniu interfejsów. Dzięki temu obiekty pełniące rolę klientów nie muszą nic wiedzieć o wewnętrznej logice działania obiektów, od których są zależne, gdyż wtedy stają się znacznie bardziej odporne na zmiany. Definiując interfejs, zaczynamy tworzyć API opisujące możliwości funkcjonalne udostępniane przez dany obiekt, sposoby korzystania z danego obiektu oraz to, jak mają wyglądać interakcje pomiędzy obiektami.

Zasady SOLID

Rozszerzeniem zasady programowania w oparciu o interfejsy są tak zwane zasady projektowe SOLID, bazujące na pomysłach Roberta Martina. SOLID to akronim nazw pięciu różnych zasad projektowych, o którym warto wspominać w każdej dyskusji poświęconej zasadom programowania obiektowego.

Zasada jednej odpowiedzialności

Idea leżąca u podstaw wzorca jednej odpowiedzialności głosi, że obiekt powinien mieć tylko jedną odpowiedzialność. Ma robić tylko jedną rzecz i robić to naprawdę dobrze. Przykłady wykorzystania tej zasady mogliśmy zobaczyć w wielu frameworkach TypeScript, których używaliśmy w tej książce. Przykładowo klasa `Model` jest używana do reprezentowania jednego modelu. Z kolei klasa `Collection` służy do reprezentacji kolekcji takich modeli, a klasa `View` pozwala na wyświetlanie modeli i ich kolekcji.

Jeśli zauważymy, że któraś z klas zaczyna stawać się superklasą, czyli zaczyna wykonywać operacje różnych typów, stanowi to sygnał, że narusza ona zasadę jednej odpowiedzialności. Oto prosty przykład: jeśli plik źródłowy jakiejś klasy zaczyna być bardzo długi, najprawdopodobniej klasa ta wykonuje zbyt wiele czynności. W takim przypadku należy zastanowić się, jaka jest jej podstawowa odpowiedzialność, a następnie podzielić możliwości funkcjonalne takiej klasy na mniejsze fragmenty i umieścić je w nowych klasach.

Zasada otwarte-zamknięte

Zasada otwarte-zamknięte głosi, że obiekt powinien być otwarty na rozszerzenia i zamknięty na modyfikacje. Innymi słowy, po zaprojektowaniu interfejsu klasy jego zmiany pojawiające się stopniowo wraz z upływem czasu powinny być wprowadzane przy wykorzystaniu zasad dziedziczenia, a nie poprzez bezpośrednią modyfikację samego interfejsu.

Koniecznym jest zwrócić uwagę, że podczas pisania bibliotek, które za pośrednictwem udostępnianego API są używane przez innych programistów, zasada ta nabiera kluczowego znaczenia. Zmiany takiego publicznego API powinny być wprowadzane wyłącznie poprzez udostępnianie jego nowej wersji opatrzonej nowym numerem i w celu zapewnienia zgodności wstecznej nigdy nie powinny powodować problemów w działaniu istniejących, wcześniejszych wersji API lub interfejsu.

Zasada podstawienia Liskova

Zasada podstawienia Liskova (ang. *Liskov Substitution Principle*, w skrócie LSP) głosi, że jeśli jeden obiekt dziedziczy po innym obiekcie, obu tych obiektów można używać zamiennie bez wywoływania problemów z funkcjonowaniem kodu. Choć można by sądzić, że zaimplementowanie tej zasady jest stosunkowo łatwe, jednak okazuje się, że może ono być złożonym problemem, zwłaszcza w kontekście reguł dziedziczenia związanych z bardziej złożonymi typami, takimi jak listy obiektów lub operacje na obiektach, które są powszechnie stosowane w kodzie korzystającym z typów ogólnych. W takich przypadkach wprowadza się pojęcie wariacji, a obiekty mogą być kowariantne, kontrawariantne bądź też inwariantne. W tej książce nie będziemy zagłębiać się w tajniki wariacji, trzeba jednak pamiętać o tej zasadzie w razie pisania bibliotek lub kodu korzystającego z typów ogólnych.

Zasada segregacji interfejsów

Ideą tej zasady jest założenie, że utworzenie wielu interfejsów jest lepszym rozwiązaniem niż utworzenie jednego dużego interfejsu ogólnego przeznaczenia. Jeśli połączymy tę zasadę z zasadą jednej odpowiedzialności, zaczniemy postrzegać nasze interfejsy jako współpracujące ze sobą niewielkie elementy jednej dużej układanki, a nie jako duże interfejsy obejmujące znaczne obszary możliwości funkcjonalnych tworzonego rozwiązania.

Zasada wstrzykiwania zależności

Chodzi w niej o to, że tworzone rozwiązania powinny zależeć od abstrakcji (czy też od interfejsów), a nie od instancji konkretnych obiektów. Jest ona zbliżona do zasady programowania w oparciu o interfejsy, a nie implementacje.

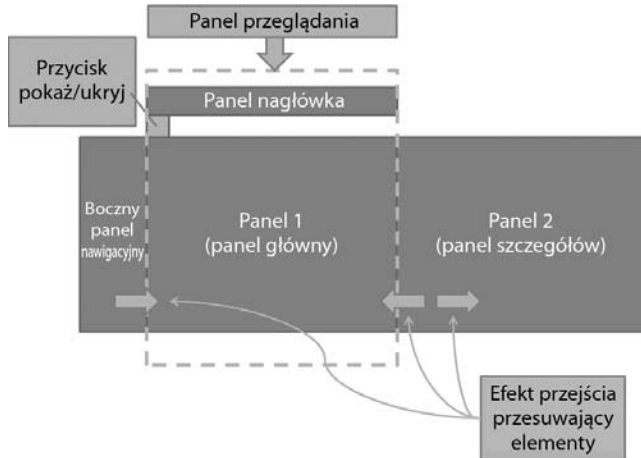
Projektowanie interfejsu użytkownika

W ramach przykładu wykorzystania zasad projektowania SOLID napiszemy teraz aplikację posiadającą stosunkowo złożony interfejs użytkownika i przekonamy się, jak można wykorzystać te zasady, by podzielić jej kod na mniejsze moduły, które będą używane przy wykorzystaniu interfejsów i które znacznie ułatwią zarządzanie bazą kodu aplikacji.

W tym podrozdziale napiszemy aplikację Angular 2, której układ będzie podzielony na panele przesuwane w poziomie. Do określania wyglądu tej aplikacji wykorzystamy framework Bootstrap, który uzupełnimy o efekty przejść CSS, dające możliwość przesuwania paneli w prawo i lewo. W ten sposób postaramy się zapewnić użytkownikom nieco inne doznania niż te, jakie daje większość witryn, których zawartość jest przewijana w pionie.

Projekt koncepcyjny

Zobaczmy, jaka jest koncepcja działania naszej witryny wraz z jej przesuwanymi panelami. Została ona zilustrowana na rysunku 11.1.



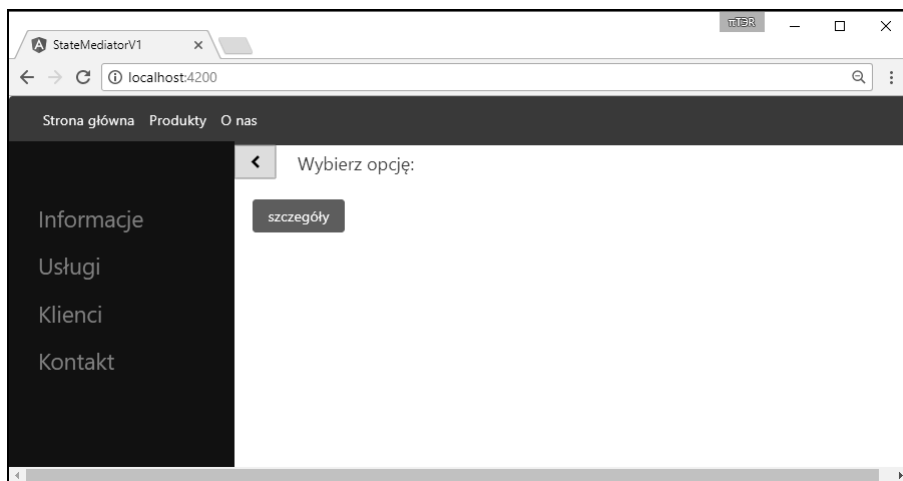
Rysunek 11.1. Projekt koncepcyjny interfejsu użytkownika aplikacji z przesuwanymi panelami

Główną stroną naszej aplikacji będzie stanowił **panel przeglądania** wraz z **panelem nagłówka** oraz **przyciskiem** kontrolującym wyświetlanie i ukrywanie bocznego panelu nawigacyjnego umieszczonego z lewej strony. W momencie otwierania lewego panelu będzie on wyświetlany z wykorzystaniem animacji CSS, które utworzą efekt przypominający wsuwanie panelu do widocznego obszaru strony. Analogicznie taki sam efekt będzie używany podczas zamykania tego panelu — będzie on wysuwany na lewo. I podobnie, po kliknięciu przycisku w celu wyświetlenia drugiego panelu (Panel 2) będzie on wsuwany z prawej strony i zajmie obszar **panelu przeglądania**; także w tym przypadku efekt wizualny zostanie uzyskany dzięki zastosowaniu animacji CSS.

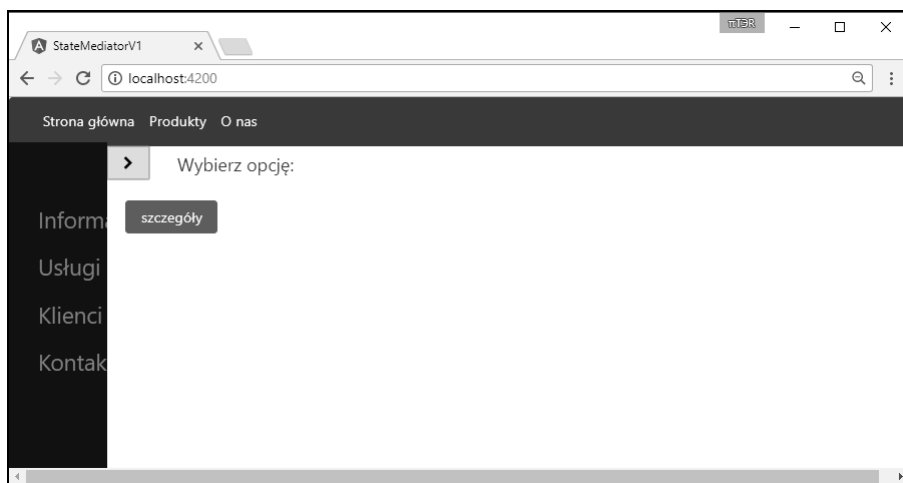
Rysunek 11.2 przedstawia panel przeglądania wraz z panelem nagłówka oraz panelem nawigacyjnym.

Na rysunku wyraźnie widać panel nawigacyjny umieszczony u góry strony, panel nawigacyjny umieszczony z lewej strony oraz dwa przyciski. Pierwszy z przycisków jest widoczny po lewej stronie tekstu *Wybierz opcję*: i widzimy na nim jedynie znak mniejszości (<) informujący, że kliknięcie przycisku ukryje panel. Kliknięcie tego przycisku uruchomi animację CSS, która wysunie panel w lewo, tak że przestanie być widoczny. Postać strony w trakcie tej animacji została przedstawiona na rysunku 11.3.

Na tym rysunku animacja została zatrzymana, tak by było widać, że lewy panel jest w trakcie ukrywania, a panel główny poszerza się, by zająć cały obszar strony panelu przeglądania. Przy tej okazji warto zwrócić uwagę, że postać przycisku do obsługi panelu nawigacyjnego została zmieniona z < na >. Ta drobna zmiana informuje, że panel boczny można ponownie wyświetlić, klikając przycisk >.



Rysunek 11.2. Główny panel przeglądarki wraz z widocznym lewym panelem nawigacyjnym



Rysunek 11.3. Animacja ukrywająca lewy panel nawigacyjny

Z kolejnego kliknięcia przycisku *Szczegóły* spowoduje, że zarówno lewy panel nawigacyjny, jak i panel główny zostaną wysunięte w lewo, dzięki czemu pojawi się drugi panel — panel szczegółów. Także i ta operacja będzie wykonywana przy wykorzystaniu animacji CSS, której postać została przedstawiona na rysunku 11.4.

Na tym rysunku widać, jak panel szczegółów jest wysuwany z prawej ku lewej, natomiast oba panele widoczne wcześniej — nawigacyjny oraz główny — są wysuwane w lewo.



Rysunek 11.4. Animacja wyświetlająca panel szczegółów

Konfiguracja aplikacji Angular 2

Skoro już wiemy, jaka jest koncepcja wyglądu naszej aplikacji, możemy zająć się implementacją tego układu, zaczynając od utworzenia i skonfigurowania aplikacji Angular 2. Jak mogliśmy się przekonać podczas prac nad poprzednimi projektami korzystającymi z tego frameworka, nową aplikację można utworzyć, wykonując z poziomu wiersza poleceń komendę `ng new`. Ten proces spowoduje zainstalowanie wszelkich niezbędnych zależności wymaganych przez framework oraz utworzy początkowe pliki aplikacji, w tym `app.component.ts` oraz `app.component.html`.

Plik `app/app.component.ts` nie może chyba być prostszy:

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

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

Na początku pliku importujemy moduł `Component`, tak jak robiliśmy już wcześniej, a następnie definiujemy trzy właściwości, które przekazujemy do dekoratora `@Component`. Należy zwrócić uwagę, że zamiast określać wartość właściwości `template`, która normalnie zawiera kod HTML, określiliśmy wartość właściwości `templateUrl`. Informuje ona framework o tym, że powinien wczytać z dysku plik o podanej nazwie, a następnie użyć jego zawartości jako szablonu. I analogicznie, używając właściwości `styleUrls`, podaliśmy adres arkusza stylów CSS, który ma być używany przez nasz komponent. Umieszczona poniżej definicja klasy `AppComponent` zawiera już tylko jedną właściwość o nazwie `title`.

Zastosowanie właściwości `templateUrl` do wczytania oddzielnego pliku zawierającego kod HTML szablonu jest przykładem zastosowania zasady wstrzykiwania zależności. Nasza klasa `AppComponent` jest zależna od szablonu HTML, który jest niezbędny do wyświetlenia komponentu w przeglądarce. W przypadku stosowania właściwości `template` pojawia się silne powiązanie pomiędzy szablonem HTML i klasą. Pojawienie się takiego ścisłego powiązania oznacza, że wszelkie zmiany wprowadzane w kodzie szablonu będą wymuszać ponowną kompilację modułu. Po przeniesieniu kodu szablonu do osobnego pliku możemy wyeliminować to ścisłe powiązanie, a klasa modułu będzie mogła być modyfikowana zupełnie niezależnie od kodu HTML szablonu.

Na obecnym etapie prac nad aplikacją plik `app.component.html` także jest bardzo prosty; jego kod został przedstawiony poniżej:

```
<div>
  {{title}}
</div>

<div>
  <button>szczegóły</button>
</div>
```

Jak widać, całą zawartość tego pliku stanowią dwa elementy `<div>`. Pierwszy z nich zawiera tytuł, a drugi — pojedynczy przycisk.

Stosowanie frameworka Bootstrap

Skoro przygotowaliśmy już podstawową strukturę naszej aplikacji, możemy zająć się uatrakcyjnieniem jej wyglądu. Do tego celu użyjemy frameworka o nazwie Bootstrap. Jest to framework HTML i CSS służący do tworzenia komponentów stron WWW, zapewniający przeważającą większość możliwości funkcjonalnych oraz stylów potrzebnych do tworzenia nowoczesnych witryn i aplikacji internetowych. Zaczynając od przycisków i ikon, poprzez karty, różnego rodzaju komunikaty i niemal wszystkie inne powszechnie używane elementy stron, Bootstrap udostępnia prostą składnię pozwalającą na łatwe dodawanie do naszych stron profesjonalnie wyglądających elementów interfejsu użytkownika. Bootstrap powstał jako framework responsywny, co oznacza, że automatycznie dostosowuje się i dobiera optymalny sposób prezentacji dla urządzeń mobilnych, tabletów i komputerów biurowych. Aby zastosować Bootstrap w naszym projekcie, trzeba go najpierw zainstalować przy użyciu narzędzia `npm`:

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

Aby dołączyć do generowanych stron plik CSS Bootstrapa, wystarczy otworzyć w edytorze plik `angular-cli.json` zapisany w głównym katalogu projektu i dodać do niego właściwość `styles`, tak jak pokazano na poniższym przykładzie:

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

Teraz możemy już przystąpić do upiększania naszej strony *app.component.html*; zaczniemy od dodania paska nawigacyjnego, który będzie umieszczony u góry. Oto jego kod HTML:

```
<nav class="navbar navbar-inverse bg-inverse navbar-toggleable-sm">
  <a class="navbar-brand">&nbsp;</a>
  <div class="nav navbar-nav">
    <a class="nav-item nav-link active">Strona główna</a>
    <a class="nav-item nav-link">Produkty</a>
    <a class="nav-item nav-link">0 nas</a>
  </div>
</nav>
```

Powyższy fragment kodu odpowiada za wyświetlenie paska nawigacyjnego. Składa się on ze znacznika `<nav>` używającego kilku klas CSS frameworka Bootstrap, dzięki którym zostanie wyświetlony jako pasek zajmujący górny fragment strony. Wewnątrz znacznika `<nav>` jest umieszczony pusty znacznik `<a>`, a następnie zagnieżdżony element `<div>` zawierający trzy kolejne zagnieżdżone znaczniki `<a>`. Te trzy odnośniki mają — odpowiednio — tytuły: *Strona główna*, *Produkty* oraz *O nas* i będą prezentowane jako elementy nawigacyjne.

Należy zwrócić uwagę, że w tym rozdziale używamy frameworka Bootstrap w wersji 4.0.0-alpha.4, który pod wieloma względami różni się od wcześniejszych wersji. Jeśli powyższy pasek nawigacyjny nie będzie wyświetlany prawidłowo, należy sprawdzić plik *package.json* i upewnić się, że jest zainstalowana odpowiednia wersja frameworka Bootstrap:

```
"dependencies": {
  ... inne biblioteki npm ...
  "bootstrap": "^4.0.0-alpha.4",
  ... inne biblioteki npm ...
}
```

Tworzenie panelu bocznego

Teraz zajmiemy się przygotowaniem lewego panelu strony. Doskonałym miejscem do poznawania elementów HTML, stylów CSS oraz animacji jest witryna <http://w3schools.com>. Jej dział dokumentacji zawiera bardzo wiele przykładów prezentujących, jak tworzyć pokazy slajdów, modalne okna dialogowe, paski postępów, responsywne tabele i wiele innych elementów stron. W naszej aplikacji wykorzystamy przykład bocznej sekcji nawigacyjnej o nazwie *Sidenav Push Content*. Pokazuje on, jak utworzyć boczny ekran nawigacyjny, który podczas otwierania wypycha na bok główną część strony, a nie przesłania jej. Zaczniemy od dodania do pliku *app.component.html* poniższego fragmentu kodu HTML:

```
<div id="mySidenav" class="sidenav">
  <a href="#">Informacje</a>
  <a href="#">Usługi</a>
  <a href="#">Klienci</a>
  <a href="#">Kontakt</a>
</div>
```

Ten fragment kodu przedstawia element `<div>` o identyfikatorze `mySidenav`, do którego dodaliśmy klasę CSS `sidenav`. Wewnątrz niego umieszczone zostały cztery odnośniki. Aby przekształcić

ten fragment kodu HTML w atrakcyjny pasek nawigacyjny, musimy otworzyć w edytorze plik *app.component.css* i dodać do niego poniższą definicję stylu:

```

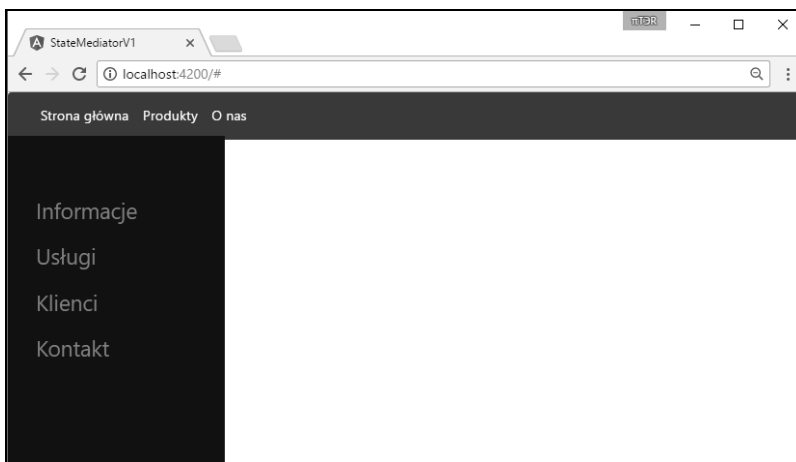
/* Boczne menu nawigacyjne */
.sidenav {
  height: 100%; /* 100% pełna wysokość */
  width: 250px; /* lub 0 - zmieniane przy użyciu JavaScriptu */
  position: fixed; /* pozostaje na miejscu */
  z-index: 1; /* pozostaje nad innymi elementami */
  top: 50px;
  left: 0;
  background-color: #111; /* czarne tło */
  overflow-x: hidden; /* bez poziomego paska przewijania */
  padding-top: 60px; /* treść przesunięta w dół o 60px */
  transition: 0.3s;
}

/* odnośniki nawigacyjne */
.sidenav a {
  padding: 8px 8px 8px 32px;
  text-decoration: none;
  font-size: 25px;
  color: #818181;
  display: block;
  transition: 0.3s
}

/* zmiana koloru po wskazaniu odnośnika myszą */
.sidenav a:hover, .offcanvas a:focus{
  color: #f1f1f1;
}

```

Dzięki tym paru stylom CSS nasz boczny panel nawigacyjny zaczyna wyglądać ciekawiej, co widać na poniższym rysunku 11.5.



Rysunek 11.5. Prosty boczny panel nawigacyjny po zastosowaniu stylów CSS

Jak widać, po zastosowaniu stylów uzyskaliśmy atrakcyjny boczny panel nawigacyjny. Niestety główna zawartość strony została pod nim ukryta, a to z kolei oznacza, że będziemy musieli przesunąć główny panel treści w prawo, wykorzystując do tego celu zewnętrzny element `<div>` i odpowiednie style CSS. Poniżej przedstawiona została zmodyfikowana wersja kodu HTML panelu głównego:

```

<div id="main" class="main-content-panel">
  <div class="row">
    <div class="col-sm-1">
      <button class="btn button-no-borders"
        (click)="showHideSideClicked()" >
        <span id="show-hide-side-button" class="fa "><</span>
      </button>
    </div>
    <div class="col-sm-11 ">
      <div class="row-content-header">{{title}}</div>
    </div>
  </div>
  <div class="main-content">
    <button class="btn btn-primary"
      (click)="buttonClickedDetail()">szczegóły</button>
  </div>
</div>

```

Jak widać, główną treść strony umieściliśmy wewnątrz elementu `<div>` o identyfikatorze "main", do którego dodaliśmy także klasę "main-content-panel". Obszar tego elementu podzieliliśmy następnie na dwie kolumny, których szerokości wynoszą — odpowiednio — 1 i 11. Ten wiersz zawiera przycisk do ukrywania i wyświetlania panelu bocznego oraz element `{{title}}`. Poniżej tego wiersza nagłówek znajduje się główna zawartość strony, która obecnie składa się z jednego przycisku z napisem *szczegóły*. Poniżej przedstawione zostały style CSS określające postać tej części strony:

```

#main {
  margin-left: 250px;
  transition: .3s;
}

#main-body {
  transition: .3s;
}

.main-content {
  padding: 20px;
}

.row-content-header {
  padding: 5px;
  font-size: 20px;
}

```

Powyższy arkusz zawiera dwie kluczowe reguły, które mają wpływ na postać treści prezentowanych na stronie. Pierwszą z nich jest reguła `margin-left: 250px` stylu `#main`. Użycie właściwości `margin-left` CSS spowoduje przesunięcie głównej treści strony w prawo, kiedy będzie widoczny lewy panel nawigacyjny. Odpowiada ona szerokości bocznego panelu nawigacyjnego, określonej przez styl `.sidenav { width: 250px; }`. Innymi słowy, panel boczny ma szerokość 250px i taką samą szerokość ma lewy margines głównego panelu strony. Połączone zastosowanie tych dwóch stylów pozwala wyświetlić boczny panel nawigacyjny i jednocześnie przesunąć w prawo panel główny. Później będziemy zmieniać te wartości z 250px na 0px i na odwrót, by ukrywać i ponownie wyświetlać panel boczny.

Drugim kluczowym stylem jest styl zawierający regułę `CSS transition: .3s`. Określa ona, ile ma trwać animacja, która będzie ukrywać lub wyświetlać panel boczny, jak również odpowiednio przesunąć główny panel strony. Po dodaniu tych stylów do arkusza możemy dopisać do strony prosty kod JavaScript, który będzie uruchamiał efekt przejścia. W tym celu w kodzie HTML strony musimy zarejestrować funkcję obsługującą zdarzenia kliknięcia, a następnie zdefiniować tę funkcję w pliku `app.component.ts`. Zaczniemy od przedstawienia kodu definiującego obsługę zdarzeń kliknięcia, umieszczonego w pliku `app.component.html`:

```
<button class="btn button-no-borders"
  (click)="showHideSideClicked()" >
  <span id="show-hide-side-button" class="fa "><</span>
</button>
```

Kolejny fragment kodu przedstawia funkcję `showHideSideClicked`, która będzie wywoływana po każdym kliknięciu przycisku do ukrywania i wyświetlania bocznego panelu nawigacyjnego. Oto zmiany, jakie należy wprowadzić w kodzie pliku `app.component.ts`:

```
export class AppComponent {
  title = 'Wybierz opcję:';
  isSideNavVisible = true;
  showHideSideClicked() {
    if (this.isSideNavVisible) {
      document.getElementById('main').style.marginLeft = "0px";
      document.getElementById('mySidenav').style.width = "0px";
      this.isSideNavVisible = false;
    } else {
      document.getElementById('main').style.marginLeft = "250px";
      document.getElementById('mySidenav').style.width = "250px";
      this.isSideNavVisible = true;
    }
  }
}
```

Jak widać, w powyższym fragmencie kodu dodaliśmy do klasy `AppComponent` nową właściwość o nazwie `isSideNavVisible`, której początkowo przypisujemy wartość `true`. Właściwość ta będzie przechowywać informację o tym, czy boczny panel nawigacyjny jest widoczny, czy nie. Poniżej umieszczona została implementacja funkcji `showHideSideClicked`. Jeśli panel boczny jest widoczny, właściwości `marginLeft` panelu głównego przypisujemy wartość `0px`, podobnie jak właściwości

with elementu mySideNav. Operacje te odpowiadają zwinięciu panelu bocznego i przesunięciu panelu głównego w lewo, tak by zajął całą szerokość strony. Jeśli natomiast boczny panel nawigacyjny jest zwinięty, postępujemy odwrotnie, czyli właściwościom marginLeft oraz width odpowiednich elementów przypisujemy wartość 250px, a dodatkowo właściwości isSideNavVisible przypisujemy wartość true. Po uruchomieniu aplikacji na obecnym etapie prac możemy już ukrywać i wyświetlać panel nawigacyjny, który dzięki zastosowaniu właściwości CSS transition: .3s jest animowany w atrakcyjny sposób.

Tworzenie nakładki

Teraz możemy skoncentrować się na drugiej stronie, która po kliknięciu przycisku *szczegóły* będzie wsuwać się na stronę od prawej krawędzi okna do lewej. Tę drugą stronę definiuje poniższy fragment kodu HTML:

```
<div id="mySidenav" class="sidenav">
  ... kod panelu bocznego ...
</div>

<div id="myRightScreen" class="overlay">
  <button class="btn button-no-borders" (click)="closeClicked()">
    <span class="fa fa-chevron-left"></span>
  </button>
  <div class="overlay-content">
    <h1>strona 2.</h1>
  </div>
</div>

... istniejący panel główny ...
<div id="main" class="main-content-panel">
```

Jak widać, do kodu pliku *app.component.html* wstawiliśmy element `<div>` o identyfikatorze `myRightScreen` i dodaliśmy do niego klasę CSS `overlay`. Ten prosty element `<div>` zawiera umieszczony u góry przycisk, w którym zdefiniowaliśmy funkcję obsługującą zdarzenia `click` (`closeClicked`) oraz element `<h1>` wyświetlający nagłówek *strona 2.* Podobnie jak w przypadku poprzednich elementów strony, także teraz do określenia postaci tego fragmentu strony będziemy potrzebowali arkusza stylów. Arkusz ten będzie musiał spełniać dwa podstawowe cele.

Przede wszystkim ta druga strona musi być początkowo przesunięta w prawo, a my musimy dysponować sposobem, by przesunąć ją w lewo po kliknięciu przycisku *szczegóły*. Poniżej przedstawiony został kod CSS określający postać elementu:

```
/* nakładka - strona 2. (tło) */
.overlay {
  height: 100%;
  width: 100%;
  position: fixed; /* ma pozostawać w miejscu */
  z-index: 1; /* ma być widoczna nad innymi elementami */
  left: 0;
```

```

top: 54px;
overflow-x: hidden; /* wyłączamy poziomy pasek przewijania */
transition: 0.3s;
transform: translateX(100%);
border-left: 1px solid;
}

```

Także w tym arkuszu używane są dwie reguły stylów kontrolujące sposób wyświetlania tej strony. Pierwszą z nich jest `transform: translateX(100%)`, a drugą `transition: 0.3s`. Pierwsza w tym przypadku zmienia początkową współrzędną X elementu `<div>` na 100%. Domyślnie oznacza to, że przesunięcie elementu na osi X ma wynosić 100% szerokości strony, czyli że element będzie całkowicie wysunięty poza nią. Z kolei druga reguła — `transition: 0.3s` — określa jedynie animację pokazywania i ukrywania elementu.

Aby pokazać działanie tych efektów, zaimplementujemy obsługę kliknięć na stronie. Najpierw zajmiemy się przyciskiem *szczegóły*; oto kod używanej przez niego funkcji obsługi zdarzeń `click`:

```

buttonClickedDetail() {
  document.getElementById('myRightScreen')
    .style.transform = "translateX(0%)";
  document.getElementById('main')
    .style.transform = "translateX(-100%)";
}

```

Powyższa funkcja wykonuje dwie operacje. Pierwszą jest przypisanie właściwości `transform` drugiej strony wartości `translateX(0%)`. Jej efekt jest przeciwieństwem użycia wartości `translateX(100%)`: przypisuje początkowej wartości współrzędnej X elementu `<div>` wartość 0%. Dzięki zastosowaniu właściwości `translate` uzyskujemy interesujący nas efekt przesuwania elementu od prawej do lewej.

Natomiast drugą wykonywaną operacją jest przypisanie właściwości `transform` elementu `<div>` strony głównej wartości `translate(-100%)`. Zapewni to efekt wysunięcia panelu głównego w lewo. Zanim przetestujemy działanie tych efektów przejść, zaimplementujemy jeszcze funkcję `closeClicked`, która będzie zamykać prawy panel strony. Jej kod został przedstawiony poniżej:

```

closeClicked() {
  document.getElementById('myRightScreen')
    .style.transform = "translateX(100%)";
  document.getElementById('main')
    .style.transform = "translateX(0%)";
}

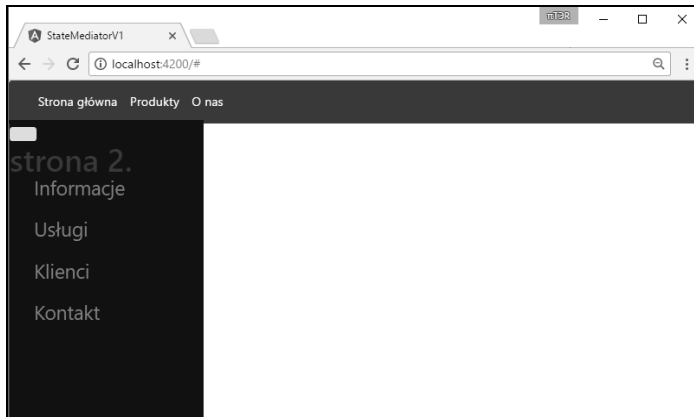
```

Ta funkcja działa odwrotnie niż przedstawiona wcześniej funkcja `buttonClickedDetail`, a jej przeznaczeniem jest wysunięcie panelu szczegółów na prawo, tak by główny panel strony ponownie stał się widoczny. Obie te funkcje współpracują, przypisując odpowiednie wartości `transform` właściwościom `transform` obu elementów `<div>`, czyli głównemu panelowi strony oraz elementowi o identyfikatorze `myRightScreen`.

Jeśli teraz wyświetlimy stronę w przeglądarce, będziemy już mogli kliknąć przycisk *szczegóły*, aby zobaczyć, jak drugi panel wysuwa się zza prawej krawędzi okna przeglądarki.

Koordinacja efektów przejść

Dotychczas udało się utworzyć prostą aplikację internetową składającą się z panelu głównego, lewego panelu nawigacyjnego oraz dodatkowego panelu szczegółów; elementy te upiększyliśmy przy użyciu stylów wzbogaconych o efekty przejść, które pozwoliły uzyskać atrakcyjną wizualnie strukturę strony. Niestety w bieżącej implementacji naszej aplikacji występują pewne problemy. Kiedy jednocześnie będzie widoczna strona główna oraz lewy panel nawigacyjny, kliknięcie przycisku *szczegóły* nie spowoduje zamknięcia panelu nawigacyjnego przed wsunięciem panelu szczegółów. W efekcie ta strona szczegółów zostanie nasunięta na panel i będzie go częściowo przesłaniać, tak jak pokazano na rysunku 11.6.



Rysunek 11.6. Prawy panel szczegółów nakładający się na panel nawigacyjny

Problem ten mogliśmy rozwiązać, wywołując zaimplementowaną już funkcję `showHideSideClicked`, która pozwoliłaby ukryć panel nawigacyjny. Choć w taki sposób mogliśmy wyeliminować problem z nakładaniem się prawego i lewego panelu, jednak takie rozwiązanie przyczyniłoby się do wystąpienia innego błędu: gdybyśmy przy początkowo widocznym panelu nawigacyjnym wyświetlili, a następnie ukryli panel szczegółów, panel nawigacyjny pozostałby niewidoczny. I choć mogliśmy próbować poradzić sobie z tym problemem, ponownie wywołując funkcję `showHideSideClicked`, także i to rozwiązanie przyczyniłoby się do występowania nowych, dziwacznych błędów.

Chociaż mogliśmy dopracować logikę aplikacji tak, by wyeliminować wszystkie te błędy, jednak szybko wpadlibyśmy we frustrujący krąg, w którym próby poprawiania jednego błędu prowadziłyby do występowania kolejnych problemów. W naszej aplikacji przydałby się jakiś mechanizm pozwalający na śledzenie wszystkich elementów wizualnych oraz kontrolowanie reakcji aplikacji na interakcje z użytkownikiem. I to właśnie w takich przypadkach bardzo przydają się wzorce projektowe Stan oraz Mediator.

Wzorzec Stan

Jednymi z wielu wzorców projektowych opisanych w książce Gangu Czworka są wzorce Stan (ang. *State*) oraz Mediator (ang. *Mediator*). Pierwszy z nich do opisanie konkretnego stanu używa grupy konkretnych klas mających wspólną klasę bazową. W ramach przykładu wyobraźmy sobie typ wyliczeniowy opisujący stany, w jakich mogą znajdować się drzwi. W pierwszym odruchu zapewne uznamy, że drzwi mogą być Otwarte lub Zamknięte.

Zastanówmy się jednak, co się stanie z naszym przepływem sterowania oraz logiką działania aplikacji, jeśli będziemy także potrzebowali stanów ZamknięteNaKlucz oraz Otworzone bądź też jeśli będziemy chcieli opisać stany drzwi przesuwanych, takie jak LekkoUchylone, PółOtwarte, PrawieOtwarte oraz CałkowicieOtwarte. Wzorzec projektowy Stan pozwala na bardzo proste definiowanie wszystkich tych stanów oraz dostosowywanie logiki na podstawie bieżącego stanu obiektu.

Jeśli zastanowimy się nad naszą aplikacją, dojdziemy do wniosku, że jej ekran w każdej chwili będzie się znajdował w konkretnym stanie. Może w niej być widoczny bądź to panel główny, bądź też panel szczegółów. Także lewy panel nawigacyjny może być widoczny lub ukryty. Potencjalne kombinacje pozwalają wskazać trzy stany aplikacji:

- widoczny tylko panel główny,
- widoczne panele główny i nawigacyjny,
- widoczny panel szczegółów.

Interfejs stanu

Wzorzec projektowy Stan ułatwia definiowanie takich stanów w kodzie programu. Podstawową zasadą, na jakiej opiera się ten wzorzec projektowy, jest utworzenie interfejsu lub abstrakcyjnej klasy bazowej oraz konkretnych klas dla każdej z możliwych specjalizacji. W naszej aplikacji Stan musi zapewniać odpowiedzi na dwa pytania.

- Czy boczny panel nawigacyjny jest widoczny?
- Czy jest widoczny panel główny, czy panel szczegółów?

Jeśli dodatkowo aktualnie jest widoczny panel główny, musimy wiedzieć, czy w jego lewym, górnym rogu należy wyświetlać przycisk ze strzałką <, czy też ze strzałką >. Nasz interfejs stanu będzie zatem miał następującą postać:

```
export enum StateType {
    MainPanelOnly,
    MainPanelWithSideNav,
    DetailPanel
}
export enum PanelType {
    Primary,
    Detail
}
```

```

export interface IState {
  getPanelType() : PanelType;
  getStateType() : StateType;
  isSideNavVisible() : boolean;
  getPanelButtonClass() : string;
}

```

Powyższy fragment kodu zaczyna się od definicji typu wyliczeniowego `StateType`, który będzie przekazywał informację o tym, w którym z trzech stanów aplikacja się aktualnie znajduje. Następnie zdefiniowaliśmy typ wyliczeniowy `PanelType`, który będzie służył do określania, czy aktualnie jest widoczny panel główny (`Primary`), czy też panel szczegółów (`Detail`). Nasz interfejs stanu `IState` określa cztery funkcje. Pierwsza z nich, `getPanelType`, zwraca wartość typu `PanelType`; a druga, `getStatePanel`, wartość typu `StateType`. Kolejna funkcja, `isSideNavVisible`, zwraca wartość logiczną określającą, czy boczny panel nawigacyjny jest widoczny, czy nie. I w końcu ostatnia z funkcji, `getPanelButtonClass`, będzie zwracać nazwę klasy CSS pozwalającą na zmianę przycisku `< na >`, zależnie od stanu bocznego panelu nawigacyjnego.

Tworząc ten interfejs, zdefiniowaliśmy jednocześnie pytania, które możemy zadawać każdej z konkretnych klas reprezentujących stany naszej aplikacji. Odpowiedzi na te pytania będą nieco inne w zależności od tego, czy widoczny będzie panel główny, czy panel szczegółów. Tak wyglądają podstawowe założenia wzorca projektowego Stan. Należy zdefiniować interfejs zwracający odpowiedzi na wszelkie pytania dotyczące stanu, a następnie używać go w kodzie aplikacji. Takie rozwiązanie pozwala odseparować implementowaną logikę korzystającą ze stanów od samej definicji tych stanów. Innymi słowy, dodawanie nowych czy też usuwanie istniejących stanów nie będzie miało żadnego wpływu na kod korzystający z interfejsu `IState`.

Konkretne stany

A teraz przyjrzymy się trzem konkretnym klasom, reprezentującym stany, w jakich może się znajdować aplikacja. Oto pierwsza z nich:

```

export class MainPanelOnly
  implements IState {
  getPanelType() : PanelType { return PanelType.Primary; }
  getStateType() : StateType { return StateType.MainPanelOnly; }
  getPanelButtonClass() : string { return 'fa-chevron-right'; }
  isSideNavVisible() : boolean { return false; }
}

```

Powyższy fragment kodu przedstawia definicję klasy `MainPageOnly` opisującej stan w sytuacji, gdy boczny panel nawigacyjny nie jest widoczny, a wyświetlony jest panel główny. To naprawdę bardzo prosta klasa implementująca interfejs `IState`, a działanie każdej z czterech funkcji interfejsu sprowadza się jedynie do zwracania odpowiednich wartości. Na podstawie tych wartości widać, że wyświetlony jest panel główny (`PanelType.Primary`), funkcja `IsSideNavVisible` zwraca `false`, a do wyświetlenia przycisku sterującego widocznością panelu nawigacyjnego używana będzie klasa `'fa-chevron-right'`. Jak pokazuje poniższy fragment kodu, pozostałe dwie konkretne klasy są bardzo podobne:

```

export class MainPanelWithSideNav
  implements IState {
  getPanelType() : PanelType { return PanelType.Primary; }
  getStateType() : StateType { return StateType.MainPanelWithSideNav; }
  getPanelButtonClass() : string { return 'fa-chevron-left'; }
  isSideNavVisible() : boolean { return true; }
}

export class DetailPanel
  implements IState {
  getPanelType() : PanelType { return PanelType.Detail; }
  getStateType() : StateType { return StateType.DetailPanel; }
  getPanelButtonClass() : string { return ''; }
  isSideNavVisible() : boolean { return false; }
}

```

Jak widać, klasa `MainPanelWithSideNav` jest niemal taka sama jak klasa `MainPanel`, z tą różnicą, że jej funkcja `isSideNavVisible` zwraca wartość `true`, a funkcja `getPanelButtonClass` zwraca łańcuch znaków `'fa-chevron-left'`. Z kolei funkcje klasy `DetailPanel` zwracają następujące wartości: funkcja `getPanelType` wartość `PanelType.Detail`, funkcja `isSideNavVisible` wartość `false`, a funkcja `getPanelButtonClass` pusty łańcuch znaków.

Wszystkie te trzy klasy są bardzo proste i opisują stan, w jakim ma być interfejs użytkownika aplikacji zależnie od stanu aplikacji. Klasy te pozwolą nam hermetyzować logikę niezbędną do zarządzania różnymi elementami widocznymi na ekranie.

Wzorzec Mediator

Skoro udało się już opisać różne stany, w jakich może się znajdować interfejs użytkownika naszej aplikacji, możemy zacząć implementować logikę wymaganą do przechodzenia pomiędzy tymi stanami. Wzorzec Mediator służy do definiowania wzajemnych interakcji w zbiorze obiektów. Wzorzec ten wstrzykuje obiekt Mediator pomiędzy obiekty, które wzajemnie na siebie wpływają, dzięki czemu obiekty nie przestają komunikować się ze sobą bezpośrednio. Takie rozwiązanie ułatwia wprowadzanie luźnych powiązań pomiędzy obiektami, które mają współpracować.

Wzorzec Mediator składa się z dwóch głównych części. Pierwszą z nich jest interfejs, którego Mediator może używać w celu wprowadzania niezbędnych zmian. W naszej aplikacji Mediator będzie musiał dysponować możliwością sygnalizowania interfejsowi użytkownika, że należy wyświetlić lub ukryć panel główny bądź też wyświetlić lub ukryć panel szczegółów. Oprócz tego Mediator będzie potrzebował możliwości zmieniania postaci przycisku — z `<` na `>` i na odwrót — w zależności od bieżącego stanu aplikacji. Mediator musi także dysponować rejestrem wszystkich dopuszczalnych stanów, a w odpowiedzi na zmianę stanu aplikacji będzie wywoływać funkcje interfejsu użytkownika.

Definiując interfejs dla tych interakcji, postępujemy zgodnie z najlepszymi praktykami programowania obiektowego i oddzielamy kod wzorca Mediator od faktycznej implementacji logiki odpowiedzialnej za zmiany w interfejsie użytkownika. Dzięki temu będziemy także mogli przetestować logikę działania Mediatora bez konieczności wykorzystania w tym celu faktycznego interfejsu użytkownika.

Nasz interfejs Mediatora będzie mieć następującą postać:

```
export interface IMediatorImpl {
  showNavPanel();
  hideNavPanel();
  showDetailPanel();
  hideDetailPanel();
  changeShowHideSideButton(fromClass: string, toClass: string);
}
```

W pięciu powyższych funkcjach zawarliśmy wszelkie zmiany interfejsu użytkownika, jakich może wymagać nasza aplikacja. Możemy zatem wyświetlić lub ukryć panel nawigacyjny, wyświetlić lub ukryć panel szczegółów oraz zmienić klasę określającą postać przycisku.

Analizując wprowadzone modyfikacje, można zauważyć, że pod dwoma względami udało się uprościć logikę biznesową naszej aplikacji. Przede wszystkim zdefiniowaliśmy stany, w jakich może się znajdować jej interfejs użytkownika, a po drugie, zdefiniowaliśmy funkcje niezbędne do wprowadzania zmian w jej wyglądzie. W ten sposób dotarliśmy już całkiem daleko na drodze do utworzenia modularnej, obiektowej i łatwej do zrozumienia aplikacji, której kodem będzie można wygodnie zarządzać.

Modularny kod

Na obecnym etapie prac nasza aplikacja dysponuje już całym kodem HTML i CSS oraz fragmentem logiki biznesowej zaimplementowanej w klasie `AppComponent`. Choć rozdzieliliśmy już tę klasę na pliki `app.component.html` i `app.component.css`, to i tak zawierają one kilka odrębnych komponentów. Skorzystajmy zatem z okazji, by nieco poprawić modularność naszego kodu. W tym celu wprowadzimy trzy nowe klasy.

- `NavbarComponent`: ta klasa będzie wyświetlać i obsługiwać pasek nawigacyjny umieszczony na samej górze strony.
- `SideNavComponent`: ta klasa będzie wyświetlać lewy panel nawigacyjny.
- `RightScreenComponent`: z kolei ta klasa będzie obsługiwać panel szczegółów wsuwany i wysuwany z prawej strony.

Wprowadzenie tych trzech nowych komponentów oznacza, że klasa `AppComponent` stanie się centralną klasą aplikacji i będzie odpowiadać za koordynację funkcjonowania wszystkich trzech pozostałych komponentów.

Komponent Navbar

Najpierw zajmiemy się klasą `NavbarComponent`, której jedynym zadaniem będzie wyświetlanie paska nawigacyjnego umieszczonego na samej górze strony. W tym celu utworzymy w katalogu `app` dwa nowe pliki — `navbar.component.ts` oraz `navbar.component.html`.

Zawartość pliku HTML wystarczy skopiować z istniejącego już pliku `app.component.html`:

```
<nav class="navbar navbar-inverse bg-inverse navbar-toggleable-sm">
  <a class="navbar-brand">&nbsp;</a>
  <div class="nav navbar-nav">
    <a class="nav-item nav-link active">Strona główna</a>
    <a class="nav-item nav-link">Produkty</a>
    <a class="nav-item nav-link">0 nas</a>
  </div>
</nav>
```

Kolejny listing przedstawia kod klasy `NavbarComponent`:

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

@Component( {
  selector: 'navbar-component',
  templateUrl: './app/navbar.component.html'
})

export class NavbarComponent {

}
```

To bardzo prosta klasa Angular 2, która w dekoratorze `@Component` odwołuje się do określonego pliku HTML i określa właściwość `selector`. Aby użyć tej nowej klasy w aplikacji, musimy ją zarejestrować we frameworku Angular, a następnie umieścić znacznik `<navbar-component>` w pliku `app.component.html`. Rejestracja komponentu sprowadza się do zaimportowania go do pliku `app.module.ts`, tak jak pokazano na poniższym przykładzie:

```
import { AppComponent } from './app.component';

// Dołączamy, aby można było odszukać dany modul.
import { NavbarComponent } from './navbar.component';

@NgModule({
  declarations: [
    AppComponent,
    NavbarComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
```



```

    providers: [],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

Jak widać, w tym pliku dodaliśmy instrukcję `import` odwołującą się do naszego nowego modułu `navbar.component`, a następnie zaktualizowaliśmy zawartość tablicy `declarations` dekoratora `@NgModule`. To właśnie dodanie wpisu do tablicy `declarations` powoduje zarejestrowanie znacznika `<navbar-component>` i umożliwia stosowanie go w kodzie HTML szablonów. Ostatnią zmianą, jaką musimy wprowadzić, jest oddanie znacznika `<navbar-component>` do pliku `app.component.html`. Po wprowadzeniu tych zmian nasza aplikacja będzie już używać modułu `navbar.component` podczas wyświetlania strony.

Komponent SideNav

Teraz w bardzo podobny sposób zaimplementujemy komponent `SideNavController`, który będzie odpowiadał za wyświetlanie i obsługę lewego panelu nawigacyjnego. W tym przypadku będziemy musieli utworzyć trzy nowe pliki: `sidenav.component.ts`, `sidenav.component.html` oraz `sidenav.component.css`. Podobnie jak we wcześniejszym komponencie, także i w tym plik HTML będzie zawierał kopię fragmentu kodu HTML z istniejącego pliku `app.component.html`:

```

<div id="mySidenav" class="sidenav">
  <a href="#">Informacje</a>
  <a href="#">Usługi</a>
  <a href="#">Klienci</a>
  <a href="#">Kontakt</a>
</div>

```

Poniżej przedstawiony został kod klasy `SideNavController`:

```

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

@Component({
  selector: 'sidenav-component',
  templateUrl: './sidenav.component.html',
  styleUrls: ['./sidenav.component.css']
})

export class SideNavController {
  closeNav() {
    document.getElementById('mySidenav').style.width = "0px";
  }

  showNav() {
    document.getElementById('mySidenav').style.width = "250px";
  }
}

```

Także w tym przypadku mamy do czynienia ze stosunkowo prostą klasą Angular 2, która w dekoratorze `@Component` rejestruje właściwość `selector` i określa używane pliki HTML i CSS. Należy jednak zwrócić uwagę, że klasa ta definiuje dwie funkcje — `closeNav` oraz `showNav`. Funkcje te ustawiają wartość właściwości CSS `style.width` na — odpowiednio — `0px` oraz `250px`. W ten sposób udało się nam w praktyce hermetyzować w jednej klasie wszystkie możliwości funkcjonalne związane z bocznym panelem nawigacyjnym. Właśnie na tym polega zasada jednej odpowiedzialności — klasa powinna robić tylko jedną rzecz.

Teraz możemy już zarejestrować ten nowy komponent w pliku `app.module.ts`, tak samo jak zrobiliśmy to poprzednio, a następnie dodać znacznik `<sidebar-component>` do pliku `app.component.html`.

Komponent RightScreen

Zajmijmy się zatem ostatnim komponentem naszej aplikacji, który będzie odpowiadał za wyświetlanie i obsługę panelu szczegółów. Nadamy mu nazwę `RightScreenComponent`. Także dla niego utworzymy plik `rightscreen.component.ts` oraz odpowiednie pliki HTML i CSS. Plików HTML i CSS tego komponentu nie będziemy tu przedstawiać, gdyż zawierają one jedynie skopiowane i wklejone fragmenty kodu z wcześniejszych plików `app.component.html` i `app.component.css`. Przyjrzymy się natomiast kodowi klasy tego komponentu:

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

@Component( {
  selector: 'rightscreen-component',
  templateUrl: './rightscreen.component.html',
  styleUrls: ['./rightscreen.component.css']
})

export class RightScreenComponent
{
  @Output() notify: EventEmitter<string>
    = new EventEmitter<string>();

  closeClicked() {
    this.notify.emit('Click from nested component');
  }

  closeRightWindow() {
    document.getElementById('myRightScreen')
      .style.transform = "translateX(100%)";
  }

  openRightWindow() {
    document.getElementById('myRightScreen')
      .style.transform = "translateX(0%)";
  }
}
```

Ta klasa jest bardzo podobna do poprzednich, gdyż także ona określa w dekoratorze `@Component` używany znacznik (we właściwości `selector`), plik szablonu (we właściwości `templateUrl`) oraz plik CSS. Dalsza część powyższego kodu to klasa komponentu `RightScreenComponent`, która definiuje trzy funkcje: `closeClicked`, `closeRightWindow` oraz `openRightWindow`. Funkcje `closeRightWindow` oraz `openRightWindow` ustawiają wartości właściwości CSS `style.transform`, tak samo jak robiliśmy to w pierwotnej wersji aplikacji opisanej w poprzednim podrozdziale.

Znacznie bardziej interesująca jest natomiast funkcja `closeClicked` oraz zastosowanie czegoś nowego, co ma nazwę `EventEmitter`. Należy zwrócić uwagę, że zarówno `EventEmitter`, jak i dekorator `Output` zaimportowaliśmy na samym początku pliku w instrukcji `import`. Angular 2 używa dekoratora właściwości `@Output` oraz klasy ogólnej `EventEmitter`, by zapewnić komponentom możliwość informowania innych komponentów o zdarzeniach.

Pamiętamy zapewne, że panel szczegółów ma w lewym, górnym rogu przycisk, który pozwala na jego zamknięcie i wyświetlenie panelu głównego. Oto fragment kodu HTML generujący ten przycisk:

```
<button class="btn button-no-borders" (click)="closeClicked()">
  <span class="fa fa-chevron-left"></span>
</button>
```

Jak wiemy, składnia używana we frameworku Angular 2 do obsługi zdarzeń `click` DOM ma następującą postać: `(click)="<funkcjaObslugi>`". W przedstawionym kodzie HTML taka funkcja obsługi ma nazwę `closeClicked`, dlatego też funkcję o takiej nazwie musimy zdefiniować w naszym komponencie.

Niemniej jednak klasa `RightScreenComponent` kontroluje jedynie obszar swojego kodu HTML i z tego względu nie może implementować możliwości funkcjonalnych związanych z innymi obszarami interfejsu użytkownika aplikacji. Oznacza to, że jedyną operacją, jaką możemy w tym komponencie wykonać, jest zgłoszenie zdarzenia informującego o kliknięciu przycisku — inne fragmenty aplikacji będą musiały na to zdarzenie jakoś zareagować. Także to rozwiązanie doskonale pasuje do zasady projektowej jednej odpowiedzialności.

Przyjrzyjmy się nieco dokładniej składni zastosowania klasy `EventEmitter`:

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

closeClicked() {
  this.notify.emit('Kliknięto w komponencie zagnieżdżonym.');
```

Emiter zdarzeń przygotowujemy, dodając dekorator `@Output` do właściwości `notify`. Następnie określamy, że typem tej właściwości będzie `EventEmitter<string>` i od razu tworzymy nową instancję klasy `EventEmitter`. Klasa `EventEmitter` jest klasą ogólną, a to oznacza, że podczas tworzenia jej instancji moglibyśmy użyć parametru typu `<number>` czy `<boolean>`, a nawet zamiast typu prostego skorzystać z jakiejś klasy złożonej.

Ponieważ właściwość `notify` jest instancją `EventEmitter` typu `string`, zatem w kodzie funkcji `closeClicked` możemy użyć wywołania `this.notify.emit` i przekazać w nim łańcuch znaków. To jedno wywołanie spowoduje wyemitowanie zdarzenia, kiedy użytkownik kliknie przycisk prezentowany przez nasz komponent.

Teraz musimy zdefiniować funkcję, która będzie te zdarzenia obsługiwać. Ponieważ za tworzenie oraz kontrolowanie komponentu `RightScreenComponent` odpowiada klasa `AppComponent`, w pliku `app.component.html` musimy zarejestrować funkcję obsługującą te zdarzenia. W tym przypadku znacznik wyświetlający nasz komponent będzie mieć następującą postać:

```
<rightscreen-component (notify)='onNotifyRightWindow($event)'\>
</rightscreen-component>
```

Jak widać, do znacznika `<rightscreen-component>` dodaliśmy atrybut pozwalający zarejestrować zdarzenie (`notify`), a następnie wywołaliśmy funkcję `onNotifyRightWindow`. Funkcja ta zostanie zdefiniowana w klasie `AppComponent`. Na razie funkcja ta może jedynie wyświetlać komunikat informujący o obsłudze zdarzenia — w ten sposób zyskamy pewność, że zdarzenie jest zgłaszane i obsługiwane.

```
onNotifyRightWindow(message:string):void {
  alert('kliknięto przycisk');
}
```

Niebawem podłączymy tę funkcję obsługi zdarzeń do klasy naszego Mediatora — w ten sposób wymusimy przejście aplikacji do innego stanu.

Komponenty podrzędne

Teraz klasa `AppComponent` jest właścicielem całej aplikacji. Wyświetla kod HTML całej strony, w tym także komponentów `navbar`, `sidenav`, `rightscreen` oraz panelu głównego. Jako taka jest także elementem nadrzędnym dla tych podkomponentów. Innymi słowy, wszystkie utworzone wcześniej komponenty są komponentami podrzędnymi klasy `AppComponent`. Czasami są one także nazywane komponentami potomnymi. Obecnie potrzebujemy jeszcze jakiegoś sposobu, który pozwoliłby na odwoływanie się w kodzie klasy `AppComponent` do komponentów `SideNavComponent` oraz `RightScreenComponent`. Chodzi tu o powiązanie instancji klas utworzonych przy użyciu znaczników HTML `<sidenav-component>` oraz `<rightscreen-component>`.

Angular udostępnia dekorator właściwości o nazwie `@ViewChild`, który służy właśnie do tego celu. Aby go zastosować w naszej aplikacji, musimy wprowadzić w kodzie klasy `AppComponent` zmiany przedstawione na poniższym przykładzie:

```
import { Component, ViewChild } from '@angular/core';
import { SideNavComponent } from './sidenav.component';
import { RightScreenComponent } from './rightscreen.component'
... @Component ...
export class AppComponent
{
```

```

@ViewChild(SideNavComponent)
private sideNav : SideNavComponent;
@ViewChild(RightScreenComponent)
private rightScreen: RightScreenComponent;

```

... dalszy kod klasy ...

Jak widać, w kodzie klasy wprowadziliśmy całkiem sporo zmian. Przede wszystkim zaimportowaliśmy dekorator `ViewChild` z modułu `@angular/core`, a następnie moduły `SideNavComponent` oraz `RightScreenComponent`. Oprócz tego zdefiniowaliśmy dwie prywatne właściwości o nazwach `sideNav` oraz `rightScreen`, które będą przechowywać instancje komponentów podrzędnych.

Następnie zastosowaliśmy dekorator `@ViewChild`, przekazując do niego nazwę klasy, do której chcemy się odwoływać. Oznacza to, że dekorator `@ViewChild(SideNavComponent)` powiąże prywatną właściwość `sideNav` z odpowiednią instancją klasy `SideNavComponent`.

Analogicznie sprawimy, że framework powiąże instancję klasy `RightScreenComponent` użytą w kodzie HTML aplikacji z prywatną właściwością `rightScreen`. W ten sposób nasza klasa `AppComponent` dysponuje już programowym dostępem do obu klas, które zostały użyte w kodzie HTML.

Implementacja interfejsu mediatora

Skoro klasa `AppComponent` dysponuje już dostępem do swoich komponentów podrzędnych, możemy przystąpić do implementacji interfejsu `IMediatorImpl`. Kod tej implementacji został przedstawiony poniżej:

```

showNavPanel() {
  this.sideNav.showNav();
  document.getElementById('main').style.marginLeft = "250px";
}
hideNavPanel() {
  this.sideNav.closeNav();
  document.getElementById('main').style.marginLeft = "0px";
}
showDetailPanel() {
  this.rightScreen.openRightWindow();
  document.getElementById('main').style.transform = "translateX(-100%)";
}
hideDetailPanel() {
  this.rightScreen.closeRightWindow();
  document.getElementById('main').style.transform = "translateX(0%)";
}
changeShowHideSideButton(fromClass: string, toClass: string) {
  if (fromClass.length > 0 && toClass.length > 0) {
    document.getElementById('show-hide-side-button')
      .classList.remove(fromClass);
    document.getElementById('show-hide-side-button')
      .classList.add(toClass);
  }
}

```

Pierwszą z funkcji zdefiniowanych w powyższym fragmencie kodu jest `showNavPanel`. Funkcja ta wywołuje implementację funkcji `showNav` komponentów podrzędnego `sideNav`, a następnie modyfikuje styl `marginLeft` elementu DOM o identyfikatorze `'main'`. Funkcja `hideNavPanel` jest bardzo podobna, lecz ma odwrotne działanie. Kolejna ze zdefiniowanych funkcji `showDetailPanel` wywołuje implementację funkcji `openRightWindow` komponentu podrzędnego `rightScreen`, a następnie ustawia wartość właściwości `transform` elementu DOM o identyfikatorze `'main'`. Po dodaniu do kodu implementacji tych funkcji możemy skoncentrować się na samej klasie `Mediator`.

Klasa Mediator

Klasa `Mediator` odpowiada za koordynację ogólnego stanu aplikacji oraz za interakcje pomiędzy poszczególnymi klasami jej interfejsu użytkownika. Oznacza to, że musi zawierać trzy kluczowe składniki. Przede wszystkim musi dysponować konkretną implementacją interfejsu `IMediatorImpl`, tak by mogła wywoływać odpowiednie funkcje, kiedy pojawi się konieczność aktualizacji interfejsu użytkownika aplikacji. Przed chwilą zaimplementowaliśmy ten interfejs w klasie `AppComponent`, a zatem będziemy musieli przekazać do klasy `Mediator` referencję do instancji klasy `AppComponent`.

Poza tym klasa `Mediator` będzie potrzebować konkretnych instancji każdej z klas stanu, tak by mogła odtworzyć zarówno bieżący stan aplikacji, jak i następny stan, w jakim ma się na znaleźć. Dzięki temu klasa będzie mogła porównać bieżący stan z następnym i na tej podstawie określić zmiany, jakie należy wprowadzić, by przejść do tego następnego stanu.

I w końcu klasa `Mediator` będzie musiała przechowywać bieżący stan aplikacji. Ponieważ odpowiada ona za przechodzenie od jednego stanu do drugiego, zatem całkiem sensowne jest potraktowanie jej jako jedynej i w pełni wiarygodnego źródła wszelkich informacji dotyczących stanu aplikacji. Oprócz tego wszędzie tam, gdzie możliwości funkcjonalne interfejsu użytkownika są zależne od bieżącego stanu aplikacji, możemy przekazywać do klasy `Mediator` wszelkie zapytania dotyczące tego, co należy zrobić, by to ona podjęła za nas odpowiednie decyzje.

Pamiętając o tych wszystkich założeniach, przeanalizujmy teraz właściwości oraz funkcję konstruktora naszej klasy `Mediator`:

```
export class Mediator {
  private _mainPanelState = new MainPanelOnly();
  private _detailPanelState = new DetailPanel();
  private _sideNavState = new MainPanelWithSideNav();

  private _currentState: IState;
  private _currentMainPanelState: IState;
  private _mediatorImpl: IMediatorImpl;

  constructor(mediatorImpl: IMediatorImpl) {
    this._mediatorImpl = mediatorImpl;
    this._currentState = this._currentMainPanelState = this._sideNavState;
  }
}
```

Na samym początku tej klasy umieściliśmy definicje trzech właściwości, w których zapisane zostaną instancje konkretnych klas stanu, mają one nazwy: `_mainPanelState`, `_detailPanelState` oraz `_sideNavState`. Kolejne dwie właściwości to `_currentState` oraz `_currentMainPanelState` i obie są typu `IState`. Będą one używane do przechowywania bieżącego stanu samej aplikacji i stanu panelu głównego. Musimy pamiętać, że jeśli przełączymy się z panelu głównego na panel szczegółów i z powrotem na panel główny, to lewy panel nawigacyjny powinien pojawić się w takim samym stanie, w jakim był na początku tej sekwencji czynności. Właśnie do tego celu będzie służyć właściwość `_currentMainPanel`.

Kolejną funkcją klasy `Mediator`, którą się zajmiemy, będzie prosta funkcja wytwórcza o nazwie `getStateImpl`, zwracająca konkretne instancje obiektów typu `IState`, wybierane na podstawie parametru typu `StateType`. Kod tej funkcji został przedstawiony na poniższym przykładzie:

```
getStateImpl(stateType: StateType) : IState {
    var stateImpl : IState;
    switch(stateType) {
        case StateType.DetailPanel:
            stateImpl = this._detailPanelState;
            break;
        case StateType.MainPanelOnly:
            stateImpl = this._mainPanelState;
            break;
        case StateType.MainPanelWithSideNav:
            stateImpl = this._sideNavState;
            break;
    }
    return stateImpl;
}
```

Jak widać, jest to prosta funkcja pomocnicza, zwracająca odpowiednią implementację obiektu stanu w zależności od przekazanej wartości typu wyliczeniowego `StateType`.

W końcu możemy przejść do kluczowego elementu klasy `Mediator` — funkcji zarządzającej zmianami interfejsu użytkownika podczas przechodzenia aplikacji z jednego stanu do drugiego. Kod tej funkcji został przedstawiony na poniższym przykładzie:

```
moveToState(stateType: StateType) {
    var previousState = this._currentState;
    var nextState = this.getStateImpl(stateType);

    if (previousState.getPanelType() == PanelType.Primary &&
        nextState.getPanelType() == PanelType.Detail ) {
        this._mediatorImpl.showDetailPanel();
    }
    if (previousState.getPanelType() == PanelType.Detail &&
        nextState.getPanelType() == PanelType.Primary) {
        this._mediatorImpl.hideDetailPanel();
    }
}
```

```

    if (nextState.isSideNavVisible())
        this._mediatorImpl.showNavPanel();
    else
        this._mediatorImpl.hideNavPanel();

    this._mediatorImpl.changeShowHideSideButton(
        previousState.getPanelButtonClass(),
        nextState.getPanelButtonClass() );

    this._currentState = nextState;
    if (this._currentState.getPanelType() == PanelType.Primary ) {
        this._currentMainPanelState = this._currentState;
    }
}

```

Powyższa funkcja `moveToState` implementuje całą logikę konieczną do obsługi trzech stanów aplikacji. Jej kod rozpoczyna się od zadeklarowania dwóch zmiennych — `previousState` oraz `nextState`. Pierwsza z nich, czyli `previousState`, określa stan, w którym aplikacja znajduje się obecnie; z kolei zmienna `nextState` zawiera stan następny, określony na podstawie parametru `stateType`. Po przygotowaniu tych dwóch obiektów możemy już zacząć porównywać ich właściwości oraz odpowiednio wywoływać funkcje interfejsu `IMediatorImpl`.

Przeanalizujmy pierwszą instrukcję warunkową `if`. Jej logika stwierdza, co następuje.

- Jeżeli jest wyświetlony panel główny, a my chcemy przejść do panelu szczegółów, musimy poinstruować interfejs użytkownika aplikacji, że należy wyświetlić panel szczegółów.

A to stwierdza druga instrukcja `if`.

- Jeżeli jesteśmy na stronie szczegółów i chcemy wyświetlić panel główny, musimy poinstruować interfejs użytkownika aplikacji, że należy zamknąć panel szczegółów.

Oto, co twierdzi trzecia z instrukcji `if`.

- Jeśli stan aplikacji informuje, że lewy panel nawigacyjny powinien być widoczny, należy go wyświetlić; w przeciwnym razie trzeba ten panel ukryć.

Następnie wywołujemy funkcję, która odpowiednio zmieni ikonę wyświetlaną na przycisku do ukrywania i wyświetlania bocznego panelu nawigacyjnego. Spowoduje to zmianę znaku wyświetlanego na tym przycisku z `<` na `>` lub na odwrót, w zależności od właściwości bieżącego i następnego stanu.

Po zakończeniu wprowadzania zmian w interfejsie użytkownika musimy jeszcze zapisać nowy stan aplikacji.

I w końcu ostatnia instrukcja `if` tej funkcji stwierdza, że jeśli aktualnie jest widoczny panel główny, należy odpowiednio zaktualizować wartość właściwości `_currentMainPanelState`. Musimy zapisać tę wartość, abyśmy po wyświetleniu panelu szczegółów i powrocie do panelu głównego mogli prawidłowo odtworzyć stan lewego panelu nawigacyjnego.

Funkcja ta zawiera wszystkie operacje niezbędne do zmieniania stanu aplikacji, wyrażone w formie prostych i zrozumiałych instrukcji. Niezbędną logikę udało się sprowadzić do zadawania kilku prostych pytań i wykonywania równie prostych czynności w zależności od uzyskanych odpowiedzi.

Stosowanie klasy Mediator

Ostatnim etapem implementacji wzorców projektowych Stan oraz Mediator jest wywołanie zmiany stanu. Operację tę może wywołać jawne żądanie przejścia do konkretnego stanu umieszczone w kodzie bądź też może stanowić ona reakcję na jakieś zdarzenia zachodzące w interfejsie użytkownika. Przede wszystkim musimy utworzyć nową instancję klasy Mediator i zarejestrować klasę AppComponent jako implementację interfejsu IMediatorImpl. Oto zmiany, jakie musimy w tym celu wprowadzić:

```
export class AppComponent implements IMediatorImpl
{
  ... istniejący kod klasy ...
  mediator: Mediator = new Mediator(this);
}
```

W powyższym fragmencie kodu zaznaczyliśmy, że klasa AppComponent implementuje interfejs IMediatorImpl, a następnie zdefiniowaliśmy zmienną lokalną o nazwie mediator. W tej zmiennej zapisujemy nową instancję klasy Mediator, do której przekazaliśmy referencję this (czyli instancję klasy AppComponent). W praktyce oznacza to, że rejestrujemy klasę AppComponent jako implementację interfejsu IMediatorImpl, której nasz Mediator będzie używać do modyfikowania interfejsu użytkownika aplikacji.

Po zarejestrowaniu klasy AppComponent w Mediatorze, możemy go już użyć do wywołania zmiany stanu aplikacji. W ramach przykładu zagwarantujemy, że bezpośrednio po uruchomieniu aplikacji będzie widoczny jedynie jej panel główny; innymi słowy, przejdziemy do stanu StateType ↪.MainPanelOnly. W tym celu musimy odwołać się do cyklu wyświetlania komponentów frameworka Angular i zaimplementować funkcję o nazwie ngAfterViewInit. Niezbędne modyfikacje zostały przedstawione na poniższym listingu:

```
export class AppComponent implements IMediatorImpl, AfterViewInit
{
  ... istniejący kod klasy ...
  ngAfterViewInit() {
    this.mediator.moveToState(StateType.MainPanelOnly);
  }
}
```

W powyższym fragmencie kodu zaznaczyliśmy, że klasa AppComponent implementuje interfejs AfterViewInit. Interfejs ten definiuje jedną funkcję ngAfterViewInit i to właśnie jej użyjemy, by przejść do stanu MainPanelOnly. Funkcja ta jest automatycznie wywoływana przez framework Angular po zainicjalizowaniu początkowego widoku komponentu. Oznacza to, że Angular już przeanalizował kod HTML komponentu, utworzył wszelkie widoki podrzędne i przesłał cały kod

HTML do przeglądarki. Dopiero w tym momencie dysponujemy referencjami do komponentów podrzędnych `SideNavComponent` oraz `RightScreenComponent`, niezbędnymi do działania klasy `Mediator`.

Teraz po wczytaniu nasza aplikacja będzie wyświetlana w odpowiednim stanie.

Reagowanie na zdarzenia DOM

Już niemal udało się nam dotrzeć do końca implementacji wzorców projektowych `Stan` i `Mediator`. Ostatnim elementem układanki jest nasłuchiwanie zdarzeń `click` DOM i wykorzystanie ich do zmiany stanu aplikacji. Zaczniemy od zmodyfikowania funkcji `buttonClickedDetail`, której nowy kod został przedstawiony poniżej:

```
buttonClickedDetail() {
    this.mediator.moveToState(StateType.DetailPanel);
}
```

Funkcja ta jest wywoływana, kiedy użytkownik kliknie przycisk *szczegóły* umieszczony na panelu głównym. Obecnie jedyną operacją, jaką ta funkcja musi wykonać, jest wywołanie funkcji `moveToState` obiektu `Mediator`, aby zmienić stan aplikacji na `DetailPanel`. Naprawdę trudno sobie wyobrazić coś prostszego.

Dodatkowo musimy także zmodyfikować funkcję obsługującą kliknięcia przycisku umieszczonego na panelu szczegółów. Musimy pamiętać, że skojarzyliśmy zdarzenia generowane przy użyciu klasy `EventEmitter` w komponencie `RightScreenComponent` z funkcją obsługi zdarzeń `onNotifyRightWindow` w klasie `AppComponent`. Teraz możemy zmodyfikować tę funkcję obsługi zdarzeń w następujący sposób:

```
onNotifyRightWindow(message:string):void {
    this.mediator.moveToState(this.mediator.getCurrentMainPanelState());
}
```

Jak widać, działanie tej funkcji sprowadza się do przejścia do poprzedniego zapamiętanego stanu panelu głównego. Również w tym przypadku trudno sobie wyobrazić jakieś prostsze rozwiązanie.

Ostatnią interakcją, którą musimy obsłużyć, są kliknięcia przycisku pokazującego i ukrywającego lewy panel nawigacyjny. Trzeba pamiętać, że rezultat kliknięcia tego przycisku będzie nieco inny w zależności od tego, czy pasek nawigacyjny będzie w danym momencie widoczny, czy nie. A zatem to nie klasa `AppComponent` powinna podejmować decyzję, co zrobić po kliknięciu tego przycisku, gdyż decyzja ta zależy od bieżącego stanu aplikacji.

A zatem sensownym rozwiązaniem będzie przechwytywanie zdarzeń kliknięcia tego przycisku w klasie `AppComponent`, a następnie przekazanie podjęcia decyzji, jak je obsłużyć, klasie `Mediator`, gdyż to właśnie ona przechowuje wszystkie informacje o bieżącym stanie aplikacji.

Kod funkcji obsługującej kliknięcie tego przycisku umieszczony w klasie AppComponent ma następującą postać:

```
showHideSideClicked() {
  this.mediator.showHideSideNavClicked();
}
```

Jak widać, całe działanie tej funkcji sprowadza się do wywołania funkcji showHideSideNavClicked klasy Mediator, która z kolei jest zaimplementowana w następujący sposób:

```
showHideSideNavClicked() {
  switch (this._currentState.getStateType()) {
    case StateType.MainPanelWithSideNav:
      this.moveToState(StateType.MainPanelOnly);
      break;
    case StateType.MainPanelOnly:
      this.moveToState(StateType.MainPanelWithSideNav);
      break;
  }
}
```

Funkcja ta sprawdza stan aplikacji, odwołując się do obiektu _currentState, i na podstawie uzyskanych informacji powoduje przejście aplikacji do stanu MainPanelOnly lub MainPanelWithSideNav.

Teraz nasza implementacja wzorców Stan i Mediator jest już kompletna.

Podsumowanie

W tym rozdziale dokładnie przyjrzelśmy się tworzeniu aplikacji Angular 2. Przy okazji poeksperymentowaliśmy trochę z układem strony wykorzystującym dwa panele przesuwane w poziomie i przekonaliśmy się, jak można korzystać ze stylów i efektów przejść CSS do tworzenia atrakcyjnych wizualnie stron WWW. Niestety nasze początkowe próby napisania aplikacji zakończyły się sporym zamieszaniem i problemami ze zmiennymi lokalnymi, których używaliśmy do kontrolowania stanu poszczególnych elementów interfejsu użytkownika.

Aby rozwiązać ten problem, cofnęliśmy się nieco i poznaliśmy wzorce projektowe Stan i Mediator; dowiedzieliśmy się także, w jaki sposób mogą one pomóc w zarządzaniu zmianami stanu aplikacji. Następnie poddaliśmy naszą aplikację gruntownej refaktoryzacji, w ramach której podzieliliśmy jej kod na sensowne komponenty. Następnie zastosowaliśmy wzorce projektowe Stan i Mediator do zarządzania stanem aplikacji oraz złożonymi przejściami występującymi podczas zmian tego stanu.

W następnym rozdziale przyjrzymy się z kolei pojęciu wstrzykiwania zależności oraz możliwościom wykorzystania nowych cech języka TypeScript w celu implementacji tego użytecznego, a jednocześnie prostego paradygmatu programowania obiektowego.

Skorowidz

A

akcesory
 prywatne, 34
 publiczne, 34
 właściwości klas, 109
aktualizacje widoków, 283
analiza parametrów
 konstruktora, 405
Angular, 203
 atrapa modułu HTTP, 478
 dziedziczenie, 207
 klasy, 205
 zmienna \$scope, 205
Angular 2, 234
 komponenty aplikacji, 437
 konfiguracja, 234, 434
 aplikacji, 363
 testów, 305
 modele, 235
 przesyłanie danych, 442
 testowanie DOM, 307
 testy
 modelu, 305
 wyświetlania, 306
 udostępnianie stron
 aplikacji, 434
 widoki, 235
 wydajność, 237
 zdarzenia, 237
Apache Cordova, 25
aplikacja
 Angular 2, 461
 Aurelia, 421
 Backbone, 227
 Express, 422
 SurfDechy, 460–497

architektura aplikacji, 496
asynchroniczność, 153
atrapa modułu Http, 478, 481
Aurelia
 formularze, 427
 komponenty, 291
 aplikacji, 424
 widoku, 293
 komunikaty, 431
 konfiguracja, 228
 testów, 295
 konwencje nazewnictwa, 294
 model widoku
 komponentów, 292
 modele, 230
 przesyłanie danych, 430
 testowanie, 291
 testy
 jednostkowe, 296
 przekrojowe, 299
 wyświetlania, 297
 udostępnianie aplikacji, 421
 wczytywanie aplikacji, 232
 widoki, 231
 wydajność, 230
 wyświetlanie komponentu,
 293
 zastosowanie, 419
 zdarzenia, 233

B

Backbone, 198, 220
 aktualizacje widoków, 283
 aplikacja, 227
 dziedziczenie, 207
 konfiguracja, 222

modele, 223
stosowanie
 dziedziczenia, 199
 interfejsów, 201
 języka ECMAScript 5, 202
 składni typów ogólnych,
 201
testowanie, 280
testy
 modeli, 285
 modelu złożonego, 287
 wyświetlania, 288
 zdarzeń DOM, 289
widok
 CollectionView, 225
 ItemView, 224
wydajność, 221
zdarzenia DOM, 283
złożone modele, 280
biblioteka, 187, 197
 Backbone, 198
 Handlebars, 347, 422
bloki kodu, 170
błędy, 161
 404, 333
 konfiguracji Require, 333
Bootstrap, 364
Bower, 196
Brackets, 413

C

Chrome, 47
ciągła integracja, 275
 serwer budowy, 277
 zalety, 276

D

dane
 JSON, 425, 440, 456, 474
 strukturalne, 171

debugowanie
 w przeglądarce Chrome, 48
 w Visual Studio, 43
 w Visual Studio Code, 52

definicje funkcji interfejsów,
 105

dekoratory, 130
 fabryka, 132
 metod, 137
 parametrów, 139
 właściwości, 135
 właściwości statycznych,
 136
 metadane, 140
 parametry, 133
 składnia, 131

DOM, 266
 obsługa zdarzeń, 283
 testy zdarzeń, 289

domknięcia, 33, 121

dziedziczenie, 113, 207
 interfejsów, 114
 klas, 114

E

ECMAScript Harmony, 27

edytor
 Brackets, 413
 kodu, 43

efekty przejść, 371

eksportowanie
 modułów, 320
 zmiennych, 323

Emmet, 415

Espruino, 25

Express, 340
 konfiguracja, 341
 obsługa tras, 344
 przekierowywanie żądań
 HTTP, 353
 stosowanie
 modułów, 342
 szablonów, 346

strony frameworka Aurelia,
 422

zdarzenia POST, 350

ExtJS, 208
 kompilator TypeScript, 211
 rzutowanie typów, 210
 tworzenie klas, 208

F

Fabryka, 95, 123

fabryki dekoratorów, 132

filtr, 489

formularze, 427

framework, 25, 213
 Angular, 203
 Angular 2, 234
 Aurelia, 228
 Backbone, 220
 Bootstrap, 364
 Express, 340
 ExtJS, 208
 Jasmine, 254
 React, 238
 Require, 326

frameworki testów
 jednostkowych, 253

funkcje, 77
 anonimowe, 78
 globalne, 184
 interfejsów, 105
 klas, 102, 183
 przeciążanie, 86, 116
 statyczne, 111, 183
 sygnatury, 184
 typy wartości zwracanych,
 77
 zwrotne, 82, 262

G

gruba strzałka, fat arrow, 84

H

Handlebars, 347

hermetryzacja, 32

I

IDE, Integrated Development
 Environment, 29, 36
 edytory, 55
 Node, 36
 Visual Studio 2017, 38
 Visual Studio Code, 48
 WebStorm, 44

implementacja
 interfejsów, 100
 interfejsu mediatora, 381
 wstrzykiwania zależności,
 399

importowanie modułów, 320

informacje o modułach, 318

inicjalizacja Typings, 194

instalacja
 pakietów, 192
 plików
 definicji, 194
 deklaracji, 191
 SystemJS, 335
 Visual Studio Code, 50

instancje klas ogólnych, 145

integracja paneli, 471

IntelliSense, 47

interfejs użytkownika, 360, 412
 edytor Brackets, 413
 panel logowania, 417
 projektowanie, 361
 rozszerzenie Emmet, 415
 testowanie zdarzeń, 484

interfejsy, 96, 100, 177, 399
 funkcji, 105
 ogólne, 150
 stanu, 372

J

Jasmine, 254
 funkcja done(), 265
 kończenie testów, 258
 mechanizmy wykonawcze,
 269
 modyfikacje DOM, 266
 obiekty dopasowujące, 257
 stosowanie szpiegów, 261,
 263

szpiegowanie funkcji
zwrotnych, 262
testy, 254
asynchroniczne, 264
bazujące na danych, 259
uruchamianie, 258
wczytywanie, 340
zdarzenia DOM, 268
jawne rzutowanie, 70
JSON, 425, 440, 456, 474

K

kacze typowanie, 65
Karma, 271
klasa Mediator, 382
klasy, 98, 182
abstrakcyjne, 118
Angular, 205
fabrykujące, 127
funkcje, 102, 183
globalne, 184
statyczne, 111, 183
konstruktory, 101
modyfikatory, 106, 108
przeciążanie konstruktora,
182
przestrzeń nazw, 182
składowe chronione, 117
właściwości, 99, 183
opcjonalne, 184
statyczne, 111
tylko do odczytu, 109
kompilacja, 28
frameworka Aurelia, 420
interfejsów, 98
modułów AMD, 324
Node, 420
w środowisku Node, 36
komponent
Angular 2, 437
Aurelia, 291, 293, 424
BoardList, 476
Navbar, 376
panelu logowania, 452
React, 448
RightScreen, 378
SideNav, 377
komunikaty, 163, 431

konfiguracja
Angular 2, 234, 363, 434
Aurelia, 228
Backbone, 222
Express, 341
modułów AMD, 326
React, 238
Require, 326
SystemJS, 335
konsola menedżera pakietów, 191
konstruktor, 101, 182
kontroler, 216
koordynacja efektów przejść, 371

L

lokalizacja usługi, 389, 396

Ł

łańcuchy szablonów, 66

M

mechanizm IntelliSense, 47
Mediator, 374
implementacja interfejsu, 381
komponent
Navbar, 376
RightScreen, 378
SideNav, 377
komponenty podrzędne, 380
zdarzenia DOM, 386
menedżer pakietów, 190, 191
metadane dekoratorów, 140
mikrokontroler
Espruino, 25
model, 215
frameworka
Angular 2, 235
Aurelia, 230
Backbone, 223
widoku komponentów
Aurelia, 292
modele
schemat, 281
testowanie, 285, 305, 313
złożone, 280
modularny kod, 375

modularyzacja, 317
moduły
AMD, 324
konfiguracja, 326
konfiguracja przeglądarki,
327
wczytywanie, 324
zależności, 328
eksportowanie, 320
zmiennych, 323
eksporty domyślne, 322
importowanie, 320
scalanie, 180
wczytywanie, 334
zarządzanie, 340
zmiana nazwy, 321
modyfikacje obsługi zdarzeń
DOM, 283
modyfikatory
dostępu w konstruktorach,
108
klas, 106
MVC, Model-View-Controller,
214
MVVM, Model View View
Model, 249

N

nakładka, 369
narzędzie, 25
JSLint, 29
Karma, 271
Protractor, 273
SystemJS, 334
Testem, 269
Typings, 192
nazwane właściwości, 339
nazwy zastępcze typów, 90
nieprawidłowe zależności, 333
Node, 340, 355
NuGet, 190

O

obiekty dopasowujące, 257
obietnice, 153, 162
składnia, 155, 158
zwracanie wartości, 158

obsługa
 błędów, 161
 szablonów, 346
 tras, 344
 zdarzeń DOM, 283
 ograniczenia typu T, 148
 określanie typów, 61, 406

P

pakiet nodemailer, 390
 pakiety, 191
 instalowanie, 192
 poszukiwanie, 192, 193
 panel
 boczny, 365
 logowania, 417, 452, 468, 492
 parametr reszty, 80
 parametry
 dekoratorów klas, 133
 domyślne, 80
 konstruktora, 405
 opcjonalne, 78
 pętla
 for...in, 68
 for...of, 68
 plik
 launch.json, 51
 package.json, 447
 SpecRunner, 255
 tasks.json, 50
 tsconfig.json, 37
 pliki
 definicji, 30
 instalowanie, 194
 pobieranie, 188
 ponowna instalacja, 195
 deklaracji, 167
 instalowanie, 191
 składnia, 180
 własne, 173
 DLL, 190
 pobieranie plików definicji, 188
 poczta elektroniczna, 390
 ustawienia konfiguracyjne, 393
 polecenie npm, 196
 poszukiwanie pakietów, 193

program
 NuGet, 190
 Bower, 196
 programowanie, 250
 asynchroniczne, 153
 obiektowe, 357
 w oparciu
 o interfejsy, 358
 o testy, 250
 projektowanie interfejsu
 użytkownika, 360
 Protractor, 273
 przeciążanie
 funkcji, 86, 116
 konstruktora klas, 182
 przeglądarka Chrome, 47
 przekierowywanie żądań
 HTTP, 353
 przesłanie funkcji, 181
 przestrzenie nazw, 112
 klas, 182
 zagnieżdżone, 181
 przesyłanie danych, 430, 442
 JSON, 456
 punkt końcowy REST, 451

R

raportowanie testów
 integracyjnych, 278
 React, 238
 komponenty, 448
 konfiguracja, 238
 testowanie, 308
 testy
 jednostkowe, 312
 modelu, 313
 widoków, 313
 zdarzeń, 315
 udostępnianie aplikacji, 445
 wczytywanie aplikacji, 243
 wiązanie danych, 454
 widoki, 240
 zdarzenia, 245
 repozytorium DefinitelyTyped, 32
 Require
 błędy konfiguracji, 333
 konfiguracja, 326
 wczytywanie, 331

REST, 451
 reszta obiektu, 93
 rozproszenie, 93
 rozszerzenie
 .d.ts, 30
 Emmet, 415
 rzutowanie typów, 70, 210

S

scalanie funkcji i modułów, 180, 185
 schemat
 aplikacji, 281
 modeli, 281
 Selenium, 273
 serwer
 Apache Cordova, 25
 Jenkins, 277
 SMTP, 395
 Team Foundation Server, 277
 TeamCity, 277
 składnia
 dekoratorów, 131
 funkcji zwrotnych, 158
 obietnic, 155, 158
 określania typów, 62
 plików deklaracji, 180
 typów ogólnych, 144
 składowe chronione, 117
 słowo kluczowe
 async, 160
 await, 160, 162, 163
 interface, 399
 let, 75
 module, 175
 super, 115
 stałe, 74
 Stan, 372
 stan panelu logowania, 468
 standard EcmaScript, 27
 stosowanie
 atrapy modułu Http, 481
 bloków kodu, 170
 ciągłej integracji, 275
 definicji klasy, 404
 dekoratorów metod, 138
 dziedziczenia, 199
 edytora Brackets, 413

filtra, 489
 frameworka
 Angular 2, 234
 Aurelia, 228
 Backbone, 220
 Bootstrap, 364
 React, 238
 funkcji `done()`, 265
 Handlebars, 347
 interfejsów, 201
 języka ECMAScript 5, 202
 klasy Mediator, 385
 komunikatów, 431
 konsoli menedżera
 pakietów, 191
 menedżera pakietów, 190
 metadanych dekoratorów,
 142
 narzędzia Typings, 192
 npm i `@types`, 196
 NuGet, 190
 obietnic, 156
 programu Bower, 196
 rozszerzenia Emmet, 415
 rzutowania typów, 210
 Selenium, 273
 składni typów ogólnych, 201
 szablonów w Express, 346
 szpiegów, 261
 szpiegów jako imitacji, 263
 typu T, 146
 wielu dekoratorów, 132
 wstrzykiwania zależności,
 408
 wzorca MVC, 218
 strażniki typów, 88
 struktura danych JSON, 474
 sygnatury funkcji, 84, 184
 syntaktyczne usprawnienia
 języka, 30
 SystemJS, 334
 instalacja, 335
 konfiguracja, 335
 wyczytywanie modułów, 334
 zależności modułów, 337
 szablony, 346
 projektów TypeScript, 39
 szpiegowanie funkcji
 zwrotnych, 262

Ś

ścisła kontrola typów, 29
 środowisko Node, 340

T

tablice, 67
 TDD, Test Driven
 Development, 250
 test Jasmine, 254
 Testem, 269
 testowanie, 278
 frameworka, 277
 Angular 2, 303
 Aurelia, 291
 Backbone, 280
 React, 308
 implementacji wzorców
 projektowych, 464
 modeli, 285, 305, 313
 modeli złożonych, 287
 wyświetlania, 288, 297, 306
 zdarzeń, 315
 DOM, 289, 307
 interfejsu użytkownika,
 484
 testy
 akceptacyjne, 253
 asynchroniczne, 264
 bazujące na danych, 259
 integracyjne, 252
 jednostkowe, 252, 296, 312,
 463
 jednostkowe żądań HTTP,
 477
 przekrojowe, 299
 raportowanie, 278
 tworzenie
 aplikacji, 411
 aplikacji webowej, 40
 instancji klas ogólnych, 145
 nakładki, 369
 obiektów w klasach
 ogólnych, 151
 panelu
 bocznego, 365
 logowania, 417

pliku `tsconfig.json`, 37
 projektu
 Visual Studio, 39
 WebStorm, 45

typ

any, 69
 T, 146, 148

Typings, 192

typy

kaczce typowanie, 65
 nazwy zastępcze, 90
 ogólne, 143, 144
 określanie, 61
 podstawowe, 60
 składnia, 62
 strażniki, 88
 unii, 88, 179
 ustalone, 73
 wnioskowanie, 64
 wyliczeniowe, 71, 400
 zaawansowane, 87

U

unie, 179

ustawienia konfiguracyjne
 poczty, 393

V

Visual Studio

debugowanie, 43
 dodawanie pliku, 41
 domyślne ustawienia
 projektu, 41
 edytor kodu, 43
 pasek narzędzi
 Standardowy, 44
 szablony projektów, 39
 tworzenie
 aplikacji webowej, 40
 projektu, 39

Visual Studio 2017, 38

Visual Studio Code, 48
 debugowanie, 52
 instalacja, 50
 VSCode, 50

W

wartość
 null, 91
 undefined, 91
 wczytywanie
 aplikacji React, 243
 frameworka Require, 331
 Jasmine, 340
 WebStorm, 44
 mechanizm IntelliSense, 47
 pliki domyślne, 45
 tworzenie
 aplikacji, 46
 projektu, 45
 wyświetlanie strony w
 Chrome, 47
 wiązanie danych, 412, 454
 widok, 215
 CollectionView, 225
 ItemView, 224
 szczegółów, 487
 widoki
 Angular 2, 235
 Aurelia, 231
 React, 240
 witryna
 Angular 2, 434
 Express i React, 443
 własne pliki deklaracji, 173
 właściwości
 klas, 99, 183
 opcjonalne, 97, 184

statyczne, 111
 tylko do odczytu, 109
 wnioskowanie typów, inferred
 typing, 64
 wstrzykiwanie
 rekurencyjne, 409
 właściwości, 407
 zależności, 203, 389, 396,
 399
 implementacja, 399
 przy użyciu dekoratora,
 403
 przy użyciu konstruktora,
 402
 wybór frameworka JavaScript,
 197
 wyczytywanie modułów, 334
 wydajność frameworka
 Angular 2, 237
 Aurelia, 230
 wysyłanie poczty
 elektronicznej, 390
 wyświetlanie listy, 483
 wyznaczanie
 interfejsów, 399
 nazwy klasy, 401
 wzorzec
 Lokalizacja usługi, 396
 Mediator, 374
 MVC, 214
 MVVM, 249
 projektowy Fabryka, 123
 Stan, 372

Z

zaawansowane typy, 87
 zagnieżdżone przestrzenie
 nazw, 181
 zalety języka, 28
 zarządzanie modułami, 340
 zasada
 jednej odpowiedzialności,
 359
 otwarte-zamknięte, 359
 podstawienia Liskova, 360
 segregacji interfejsów, 360
 wstrzykiwania zależności,
 360
 zasady SOLID, 359
 zdarzenia
 DOM, 268, 283, 386
 frameworka
 Angular 2, 237
 Aurelia, 233
 React, 245
 interfejsu użytkownika, 484
 POST, 350
 zintegrowane środowisko
 programistyczne, *Patrz* IDE
 zmienna \$scope, 205
 zmienne globalne, 168

Ż

żądania HTTP, 477
 żądanie POST, 456

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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>

TypeScript: poznaj język najlepszych projektantów!

Język TypeScript, który wraz z kompilatorem i zestawem narzędzi jest udostępniany na zasadach *open source*, zyskuje ogromne uznanie tysięcy projektantów aplikacji. TypeScript pozwala na pracę w zgodzie ze standardami języka JavaScript (ES5, ES6 i ES7), co pozwala programistom na używanie klas, interfejsów, typów ogólnych itd. Okazuje się, że TypeScript umożliwia tworzenie solidnych aplikacji przy wykorzystaniu technik obiektowych — i są to nie tylko aplikacje WWW, lecz także aplikacje serwerowe, aplikacje dla urządzeń mobilnych, a nawet oprogramowanie do sterowania urządzeniami w internecie rzeczy (IoT).

Niniejsza książka jest przewodnikiem po TypeScript dla programistów. Przedstawiono tu zarówno podstawy, jak i zaawansowane możliwości języka, takie jak typy ogólne i techniki programowania asynchronicznego. Sporo miejsca poświęcono prezentacji najpopularniejszych frameworków JavaScript. Opisano sposoby korzystania z mechanizmów ścisłej kontroli typów i omówiono techniki programowania obiektowego w języku TypeScript. Nie zabrakło również wskazówek dotyczących najlepszych praktyk projektowania aplikacji. Dzięki lekturze tej książki osiągnięcie profesjonalnego poziomu pisania aplikacji w TypeScript stanie się o wiele łatwiejsze!

Najważniejsze zagadnienia:

- składnia języka TypeScript: podstawy i zagadnienia zaawansowane
- środowisko pracy: kompilator, narzędzia, frameworki
- tworzenie plików deklaracji i korzystanie z bibliotek
- programowanie oparte na testach
- modularyzacja i programowanie zorientowane obiektowo w TypeScript
- podstawowe elementy konstrukcyjne aplikacji internetowych

Nathan Rozentals — może się pochwalić imponującym doświadczeniem w kodowaniu: tworzył programy do analizy statystycznej na komputerach mainframe na długo przed erą internetu. Brał również udział w rozwiązywaniu tzw. problemu roku 2000. Perfekcyjnie poznał wiele obiektowych języków programowania, takich jak C++, Java czy C#. Ostatnio skoncentrował się na nowoczesnym programowaniu aplikacji internetowych, a język TypeScript okazał się jego ulubionym narzędziem. W chwilach wolnych od programowania zajmuje się windsurfingiem lub piłką nożną.

	
księgarnia internetowa	
http://helion.pl	
zamówienia telefoniczne	
	0 801 339900
	0 601 339900
Informatyka w najlepszym wydaniu	
Helion SA ul. Kościuszki 1c, 44-100 Gilwice 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/nawosci	
 KOD KORZYŚCI	
ISBN 978-83-283-3641-4	
 9 788328 336414	
cena: 89,00 zł	

Packt