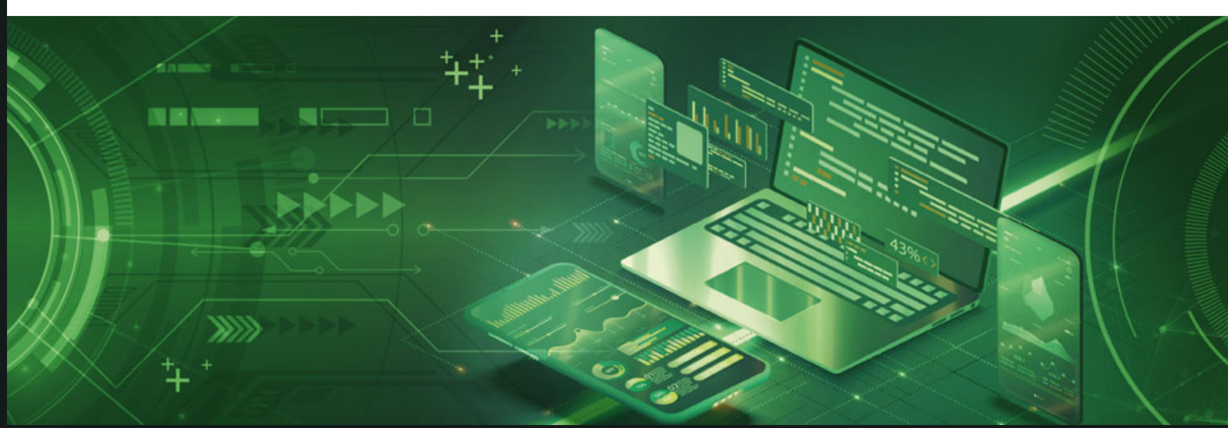
DO NOWEJ PODSTAWY PROGRAMOWEJ

Część 3
Aplikacje webowe

Kwalifikacja INF.04
Projektowanie, programowanie i testowanie aplikacji



Podręcznik do nauki zawodu
technik programista

Łukasz Guziak

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

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

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

Redaktor prowadzący: Joanna Zaręba

Projekt okładki: Jan Paluch

Ilustracja na okładce została wykorzystana za zgodą Shutterstock.

Helion S.A.
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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?inf043_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-8768-3

Copyright © Helion S.A. 2021

- Poleć książkę na Facebook.com
- Kup w wersji papierowej
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp 5

Rozdział 1. Biblioteka jQuery i framework Bootstrap 7

1.1. Biblioteka jQuery 7

1.2. Framework Bootstrap 49

Rozdział 2. Framework Angular 59

2.1. Angular — instalacja i konfiguracja środowiska pracy 59

2.2. TypeScript 68

2.3. Angular — pierwsze kroki 122

2.4. Dyrektywy 136

2.5. Komponenty 152

2.6. Usługi 170

2.7. Zdarzenia i formularze 174

Rozdział 3. Środowisko uruchomieniowe — platforma Node.js 187

3.1. Platforma Node.js 187

3.2. Asynchroniczność w Node.js 198

3.3. Moduły w Node.js 201

3.4. Serwer HTTP — Node.js 214

3.5. Framework Express 226

3.6. Baza danych MongoDB 242

Rozdział 4. Przykład użycia środowiska Node.js i frameworka Express w połączeniu z bibliotekami Bootstrap i jQuery 259

4.1. Utworzenie plików i przygotowanie serwera Node.js w oparciu o framework Express 260

4.2. Przesłanie informacji z backendu do frontendu. Użycie metody .fetch(). Czym jest obietnica? 262

4.3. Odpowiedź frontendu — użycie atrybutu data, definicja zdarzenia i ponownie metoda .fetch() 270

4.4. Szata graficzna — dołączamy framework Bootstrap i bibliotekę jQuery 286

Bibliografia 290

Skorowidz 291

Wstęp

Niniejsza publikacja wchodzi w skład serii podręczników do nauki zawodu technik programista. Poruszono w niej zagadnienia związane z tworzeniem aplikacji webowych.

Przedstawiony w podręczniku materiał obejmuje podstawę programową wymienioną w dziale *INF.04.7. Programowanie aplikacji zaawansowanych webowych* kwalifikacji *INF.04. Projektowanie, programowanie i testowanie aplikacji* i pozwoli postawić następny krok w świecie technologii webowych.

Nieustanne zmiany zachodzące w otaczającym nas świecie nie ominęły zwłaszcza tej dziedziny. Jeszcze niedawno nie było smartfonów, Facebooka, aparatów cyfrowych, dostęp do internetu uzyskiwało się za pomocą modemu z szybkością 56 kb/s, a strony internetowe były tworzone jedynie za pomocą języka HTML i nie zapewniały żadnej interakcji z użytkownikiem. Technologie webowe nieustannie idą do przodu — pojawiają się nowe rozwiązania, a dotychczasowe są stopniowo wypierane przez nowsze. W tej branży spoczęcie na laurach oznacza krok w tył, dlatego programista stale musi się rozwijać i rozbudowywać swój warsztat pracy.

Podręcznik składa się z czterech rozdziałów. W trzech pierwszych omówiono technologie webowe, a w czwartym pokazano sposób ich użycia.

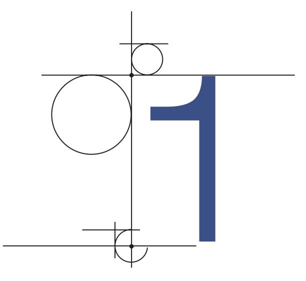
Rozdział 1. przedstawia bibliotekę jQuery, która usprawnia manipulowanie elementami dokumentu HTML i obsługę zdarzeń oraz pozwala tworzyć animacje. Zastosowanie jQuery upraszcza i przyśpiesza pracę programisty, a także sprawia, że napisany kod jest kompatybilny z wieloma przeglądarkami internetowymi. Użycie omówionego w tym rozdziale frameworka Bootstrap sprawi zaś, że strona wyświetli się poprawnie na każdym urządzeniu, niezależnie od wielkości ekranu.

Rozdział 2. został w całości poświęcony frameworkowi Angular, za pomocą którego stworzysz aplikację webową. Lwią część rozdziału zajmuje omówienie składni języka TypeScript, który dzięki swoim zaletom (takim jak m.in. sprawdzanie poprawności typów i lepsza współpraca z edytorami kodu) idealnie nadaje się do tworzenia dużych aplikacji internetowych.

Rozdział 3. przedstawia platformę Node.js, która pozwala uruchomić kod JavaScript poza przeglądarką. Node.js w połączeniu z frameworkiem Express, również omówionym w tym rozdziale, otworzył nowe obszary zastosowań języka JavaScript, który może być wykorzystywany do tworzenia aplikacji działającej po stronie serwera. Ponadto w rozdziale znalazł się opis nierelacyjnej bazy danych MongoDB.

W ostatnim rozdziale wspólnie stworzymy projekt oparty na przedstawionych w podręczniku technologiach webowych.

Wszystkie liczące się witryny internetowe stawiają na interakcję. Użytkownik już nie jest widzem — jest uczestnikiem. Znajomość omówionych w podręczniku rozwiązań pozwoli tworzyć takie interaktywne strony.



Biblioteka jQuery i framework Bootstrap

1.1. Biblioteka jQuery

jQuery jest biblioteką JavaScriptu, która upraszcza tworzenie witryn. Cel, jaki przyświecał twórcom, został zawarty w logo — *write less, do more*.

1.1.1. Czym jest jQuery?

Pomimo upływu lat (pierwszą wersję udostępniono w 2006 r.) jQuery nadal jest bardzo popularną biblioteką, która pozwala tworzyć nowoczesne i funkcjonalne strony. Co zatem zyska programista, który zdecyduje się na jej użycie? Pierwszą zaletą jest wybór elementów. „Chwytanie" elementów strony, które będą modyfikowane, jest bardziej intuicyjne, niż gdy wybieramy je za pomocą JavaScriptu. Korzystamy z czegoś, co już znamy i co opanowaliśmy, a mianowicie operację tę **przeprowadzamy z użyciem selektorów stylu CSS**. Wykonywane zadania stają się łatwiejsze do przeprowadzenia, dzięki czemu wybrany element możemy podmienić, ukryć, usunąć czy nadać mu klasę, a uaktualnienie strony o nowe efekty nie wiąże się z przebudową jej kodu. Docenianym przez wielu atutem jest to, że biblioteka działa spójnie w wielu przeglądarkach internetowych. Pozwala to programiście skupić się na mechanice strony, a nie obsłudze zdarzeń dla różnych przeglądarek, np. na tym, jak użyta funkcjonalność nieobecna w starszej wersji przeglądarki ma zostać przez nią wykonana — jQuery automatycznie zadba o jej obsługę i sprawi, że będzie ona działać.

Wszystko to sprawia, że osoba projektująca stronę ma ułatwione zadanie, gdyż korzysta z gotowych, sprawdzonych i działających rozwiązań i nie musi poświęcać czasu na ich tworzenie.

CIEKAWOSTKA

Aby sprawdzić samemu, na co stać jQuery, należy odwiedzić stronę *https://jqueryui.com/demos/*.

Przygodę z jQuery rozpoczynamy od pobrania biblioteki. Jej ostatnia wersja jest dostępna na stronie projektu, pod adresem *https://jquery.com/*. W dziale *Download* dostępne są dwie opcje: **skompresowana** (poznamy ją po słowie **min** w nazwie pliku) i **development**. Obie zapewniają tę samą funkcjonalność, natomiast różni je wielkość pliku. Pierwsza została pozbawiona znaków białych (spacje, tabulacje, znaki przejścia do nowej linii) oraz komentarzy, przez co jest mało czytelna, ale jej rozmiar został ograniczony do minimum. Wszystko po to, aby wczytywała się możliwie szybko. Tę wersję stosujemy na gotowej, działającej stronie (wersji produkcyjnej). Druga zajmuje więcej miejsca, ale dzięki zastosowaniu wcięć i komentarzy łatwiej poznać jej kod i sposób działania.

Bibliotekę po pobraniu łączymy ze stroną HTML. Połączenie wykonujemy po podaniu ścieżki do pliku, która jest umieszczona pomiędzy znacznikami `<script></script>`, np. `<script src="jquery-3.6.0.js"></script>`. Kod ten może się znajdować w sekcji `head`, choć bardzo często jest umieszczony tuż przed zamknięciem sekcji `body` (ponieważ jQuery operuje na elementach strony, mamy pewność, że istnieją one w drzewie dokumentów).

Aby móc rozpocząć pracę z jQuery, należy sprawdzić, czy strona jest gotowa. Zrobimy to za pomocą kodów przedstawionych na listingach 1.1 i 1.2 (ta wersja, częściej wybierana i oficjalnie rekomendowana, została użyta w przykładach i na listingach).

Listing 1.1. Metoda .ready()

```
$(document).ready(function() {
// Kod skryptu
});
```

Listing 1.2. Skrócona forma metody .ready()

```
$(function() {
// Kod skryptu
});
```

Kod ten umieszczamy w zewnętrznym pliku, który podobnie jak bibliotekę jQuery, łączymy ze stroną lub umieszczamy w sekcji `head`, otaczając znacznikami `<script></script>`. Obie metody zostały pokazane na listingach 1.3 i 1.4. Kod z listingu 1.3 wczytuje bibliotekę jQuery oraz zawartość zewnętrznego pliku *pierwszy_przyklad.js*, a pokazany na listingu 1.4 kod zawarty w *pierwszy_przyklad.js* przenosi do sekcji `head`.

Listing 1.3. Użycie zewnętrznego pliku

```
<html>
<head>
    <title>Tytuł strony</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
</head>

<body>
    <!-- Kod strony HTML -->
    <script src="jquery-3.6.0.js"></script>
    <script src="pierwszy_przyklad.js"></script>
</body>
</html>
```

Listing 1.4. Sekcja head

```
<html>
<head>
    <title>Tytuł strony</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">
        // Kod skryptu
    </script>
</head>

<body>
    <!-- Kod strony HTML -->
</body>
</html>
```

Który kod lepiej wybrać? Ten z listingu 1.3, ponieważ **skrypt nie blokuje wczytania pozostałych elementów strony, a elementy, na których operuje, zostały już wczytane**.

Większość listingów przedstawionych w tym rozdziale opiera się na schemacie z listingu 1.4. Ponieważ działanie jQuery jest omawiane jako pierwsze, kod skryptu, aby nie trzeba go było szukać, został umieszczony najbliżej jego opisu, w sekcji `head`. Użyty kod, który nie był zmieniany w celu skrócenia listingów, np. CSS, został przeniesiony do oddzielnego pliku i jest powiązany za pomocą odnośnika z dokumentem HTML.

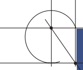
Istnieje jeszcze jeden sposób powiązania biblioteki jQuery ze stroną projektu — **użycie serwerów CDN** (ang. *Content Delivery Network*). CDN to system serwerów, którego zadaniem jest dostarczenie w jak najkrótszym czasie żądanych treści. Serwery systemu są rozlokowane w różnych zakątkach globu i pobierają one, zapisują i przechowują zawartość oryginalnej

witryny. Dzięki temu użytkownik nie musi łączyć się z serwerem źródłowym — zostaje automatycznie wybrany najbliższy serwer. Użycie CDN zmniejsza opóźnienia i skraca czas ładowania zasobu.

Przykładowo dołączenie do strony kodu:

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js" integrity="sha256-/xU
j+3OJU5yExlq6GSYGSHk7tPXikynS7ogEvDej/m4=" crossorigin="anonymous"></script>
```

spowoduje pobranie biblioteki jQuery z zasobów zewnętrznych.



CIEKAWOSTKA

Użycie atrybutu `integrity` umożliwia przeglądarkom zweryfikowanie, czy pobrany zasób nie był zmieniany wbrew zamierzeniom, np. przez dołączenie złośliwego kodu. Jego działanie opiera się na użyciu funkcji skrótu, tzw. hasha — jakakolwiek ingerencja w dane spowoduje, że hash się zmieni i nie będzie zgodny z podanym.

1.1.2. Wyszukiwanie elementów i operacje na nich

Wyszukiwanie elementów i pobieranie ich zawartości

Wyszukiwanie elementów

Chwytanie elementów strony to czynność, którą będziemy wykonywali nader często. W poprzednim rozdziale na wstępie zaznaczono, że tę operację przeprowadza się za pomocą **selektorów CSS**, prześledźmy zatem i omówmy wykonanie.

Listing 1.5 przedstawia kod HTML strony, który wyświetla najpopularniejsze języki programowania w postaci listy punktowanej (znaczniki `<ul></ul>`). To nasza bazowa strona, która posłuży do zademonstrowania treści omawianych w tym rozdziale. Pogrubioną czcionką zaznaczono kod, który wprowadza nowe funkcje, nieobecne w poprzednich przykładach.

Listing 1.5. Strona Najpopularniejsze języki programowania

```
<html>
<head>
    <title>Najpopularniejsze języki programowania</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
</head>

<body>
    <div id="container">
        <div id="page">
            <h2>Najpopularniejsze języki programowania</h2>
            <ul>
```

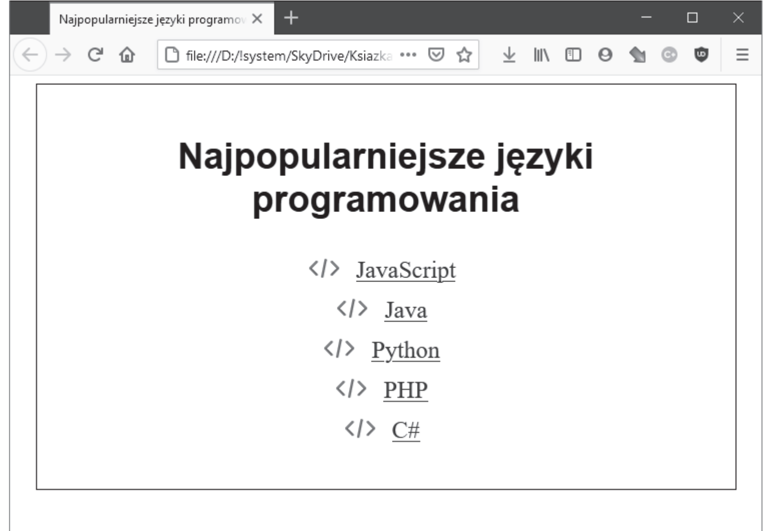
```
                <li><a class="link" href="#">JavaScript</a></li>
                <li><a class="link" href="#">Java</a></li>
                <li><a class="link" href="#">Python</a></li>
                <li><a class="link" href="#">PHP</a></li>
                <li><a class="link" href="#">C#</a></li>
            </ul>
        </div>
    </div>
    <script src="jquery-3.6.0.js"></script>
    <script src="pierwszy_przyklad.js"></script>
</body>
</html>
```

Ze stroną został połączony arkusz stylów przedstawiony na listingu 1.6.

Listing 1.6. Arkusz stylów strony Najpopularniejsze języki programowania

```
h2 {
    font-family: Arial;
}

#container {
    width: 700px;
    margin: 0 auto;
}

#page {
    font-size: 150%;
    text-align: center;
    padding: 20px;
    border: 1px solid black;
    background-color: white;
}

ul {
    margin: 20px;
    list-style-image: url('code.png');
    // Użyty selektor list-style-image określa plik graficzny, który ma zostać użyty jako znacznik elementu listy.
}

li {
    padding: 5px;
    margin-bottom: 2px;
}

img {
    width: 300px;
}
```

Widok surowej strony (bez jQuery) jest pokazany na rysunku 1.1.



Rysunek 1.1. Ranking języków programowania

Przed zamknięciem sekcji `body` ze stroną zostają połączone jeszcze dwa pliki zawierające skrypty. Pierwszy, *jquery-3.6.0.js*, to biblioteka jQuery, a drugi, *pierwszy_przyklad.js*, to plik, który z niej korzysta. Zawartość tego drugiego pliku jest przedstawiona na listingu 1.7. Pokazuje on sposoby wybierania elementów strony.

WSKAZÓWKA

Z jQuery można korzystać, używając pełnej nazwy funkcji, *jQuery()*, jak również formy skróconej, *$()*.

Pierwszy sposób to użycie selektora `#`, który chwyta element `div` o identyfikatorze `page`. Użycie metody `.css()` powoduje zmianę koloru tła.

Drugi sposób to wybranie znacznika. W tym przykładzie został wybrany nagłówek drugiego stopnia (`<h2>`) i tu również został zmieniony styl — dodano 5-pikselowy odstęp pomiędzy literami.

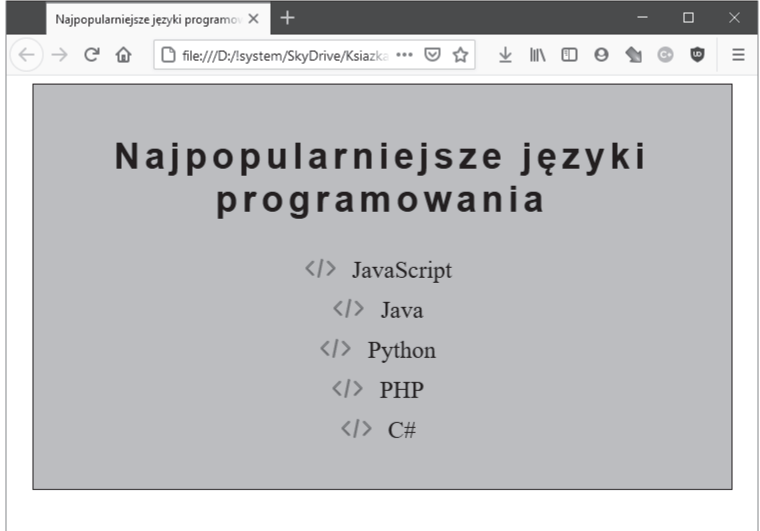
Trzeci sposób jest oparty na wyborze klasy. Do wszystkich odnośników została przypisana klasa `.link`. Skrypt ponownie zmienia styl — ustawia kolor czcionki na czarny i usuwa podkreślenie.

Ostatni przykład wyszukuje elementy strony, które mają ustawiony atrybut `href="#"`, a następnie za pomocą metody `.attr()` zostaje on zamieniony na `'https://pl.wikipedia.org/wiki/J%C4%99zyk_programowania'` — do wszystkich niezdefiniowanych odnośników zostaje przypisany adres strony.

Listing 1.7. Metody chwytania elementów strony

```
$(function () {
    $('#page').css('background-color', '#00FFFF');
    $('h2').css('letter-spacing', '5px');
    $('.link').css({
        'color': 'black',
        'text-decoration': 'none',
    });
    $('[href="#"]').attr('href', 'https://pl.wikipedia.org/wiki/J%C4%99zyk_
programowania');
});
```

Skrypt zmodyfikował wygląd strony. Jej nowa wersja jest pokazana na rysunku 1.2.



Rysunek 1.2. Chwytanie elementów strony

Oprócz przedstawionych przykładów istnieją dodatkowe metody, które umożliwiają filtrowanie. Ich opis jest zawarty w tabeli 1.1.

Tabela 1.1. Metody pomocne w chwytaniu wybranych elementów strony

Nazwa	Opis
`.first()`	Pierwszy element z wybranych
`.last()`	Ostatni element z wybranych
`.even()`	Parzyste elementy zbioru
`.odd()`	Nieparzyste elementy zbioru
`.eq(nr)`	Elementy o podanym numerze indeksu
`.lt(nr)`	Elementy o numerze indeksu mniejszym niż podany
`.gt(nr)`	Elementy o numerze indeksu większym niż podany

Nazwa	Opis
`.not(selektor)`	Wszystkie elementy z wyjątkiem wskazanego
`.find(selektor)`	Wyszukuje wewnątrz danego elementu inny element określony przez selektor
`.filter(fn*)`	Filtruje listę elementów przez zadane kryterium

UWAGA

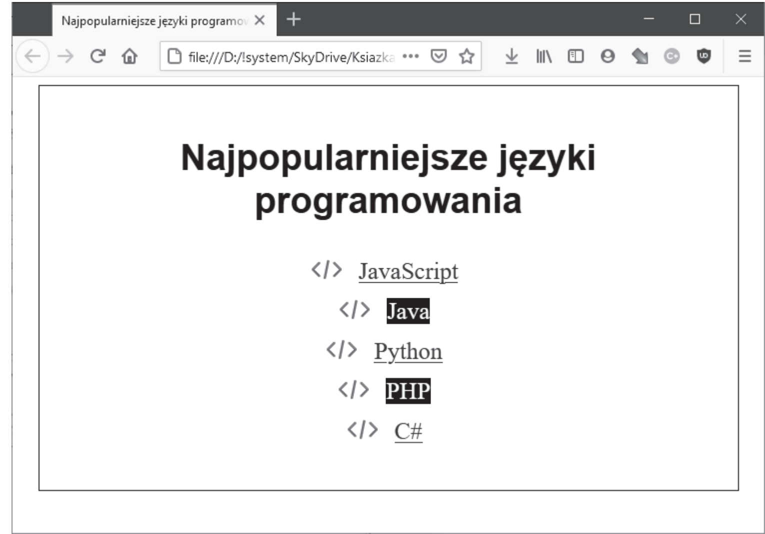
Przy użyciu np. selektora `.eq()` zostaje pobrana cała kolekcja elementów, a to, który zostanie zastosowany, określa zdefiniowany indeks. Użycie np. kodu `$('button').eq(1)` sprawi, że z kolekcji wszystkich przycisków wybrany zostaje drugi.

Zastosowanie kodu pokazanego na listingu 1.8 spowoduje zmianę koloru tła wszystkich odnośników, których indeks jest nieparzysty (pamiętamy, że indeks rozpoczyna się od 0).

Listing 1.8. Nieparzysty odnośnik

```
$(function () {
    $('a').odd().css({
        'color': 'white',
        'text-decoration': 'none',
        'background-color': 'black',
    })
});
```

Rezultat działania kodu zademonstrowano na rysunku 1.3.



Rysunek 1.3. Nieparzyste odnośniki — użycie metody .odd()

Listing 1.9 przedstawia jeszcze jeden przykład wybierania elementów, tym razem z użyciem metody `.not()`. Zmieniany jest styl wszystkich odnośników z wyjątkiem tego, którego `id` zostało zdefiniowane jako `pierwszy` (w kodzie HTML strony przy pierwszym odnośniku dopisano `id="pierwszy"`).

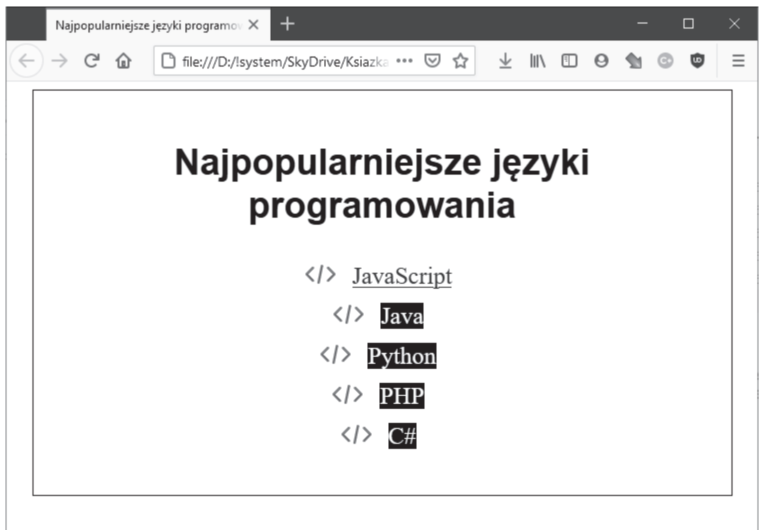
Listing 1.9. Użycie metody .not()

```
$(function () {
    $('a').not('#pierwszy').css({
        'color': 'white',
        'text-decoration': 'none',
        'background-color': 'black',
    })
});
```

WSKAZÓWKA

Każda metoda biblioteki jQuery, która nie zwraca wartości, zwraca obiekt jQuery, nowy lub poprzedni, dlatego jest możliwe wywoływanie łańcuchowe (ang. *chain* lub *method chaining*). Zastosowano je np. w powyższym kodzie — w pierwszym kroku chwytamy wszystkie elementy, które nie mają identyfikatora `pierwszy`, a w drugim zmieniamy styl.

Rezultat działania kodu jest pokazany na rysunku 1.4.



Rysunek 1.4. Użycie metody .not()

UWAGA

We wcześniejszych wersjach jQuery filtrowanie odbywało się z wykorzystaniem pseudoklasy, co oznaczało użycie po nazwie selektora znaku dwukropka. Od wersji jQuery 3.4 ta metoda jest przestarzała — dotyczy zastosowania `:eq()`, `:even`, `:first()`, `:gt()`, `:last`, `:lt()` oraz `:odd`. To oznacza, że nowa wersja zapisu przedstawiona na listingu 1.8, `$('a').odd().css({ ... })`, zastąpiła `$('a:odd').css({ ... })`.

Pobieranie zawartości elementu

Zadaniem metod `.html()` i `.text()` jest pobranie lub uaktualnienie zawartości elementu. Pierwsza pobiera lub ustawia kod HTML elementu, druga zaś — tekst elementu.

Zastosowanie metody `.html()` jest pokazane na listingu 1.10. Za jej pomocą zostaje pobrana zawartość listy `<ul>` i element tej listy, `<li>`. Pobrana zawartość zostaje przypisana do zmiennych `$lista_ul` i `$lista_li`, a następnie wyświetlona w konsoli (rysunek 1.5).

Jeśli wybrane elementy zamierzamy wykorzystywać w dalszej części kodu, **swój wybór możemy przypisać do zmiennej**.

UWAGA

Na listingu 1.10 nazwy zmiennych rozpoczynają się od znaku dolara. Nie jest to wymagane, lecz ma jedynie zaznaczyć, że w zmiennej znajduje się obiekt jQuery.

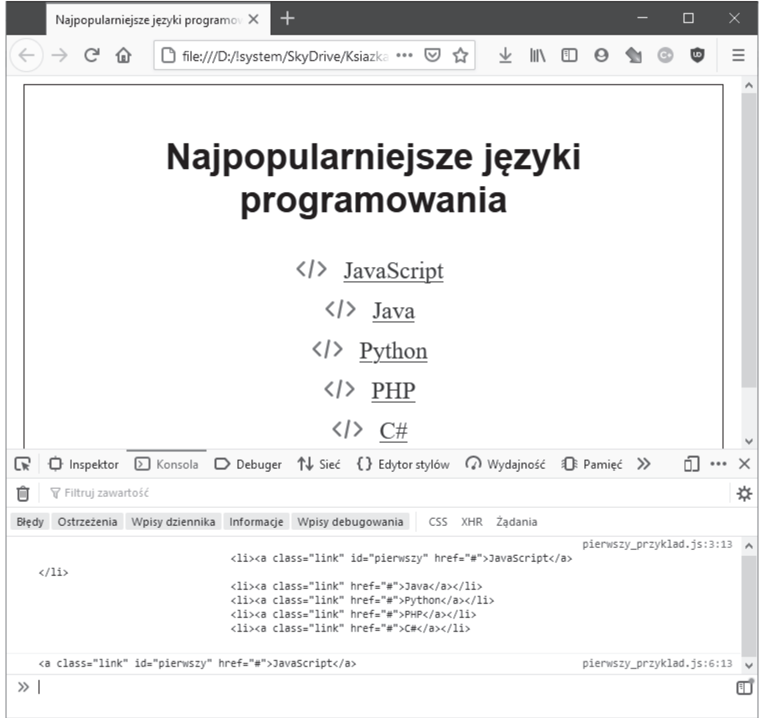
Listing 1.10. Pobranie zawartości elementu — metoda .html()

```
$(function () {
    const $lista_ul = $('ul').html();
    console.log($lista_ul);

    const $lista_li = $('li').html();
    console.log($lista_li);
});
```

Jak można zauważyć, użycie metody `.html()` na liście `<ul>` spowodowało zwrócenie zawartości całej listy, a gdy została ona połączona z elementem `<li>` — wartości tylko pierwszego elementu tej listy. Stało się tak, ponieważ metoda `.html()` pobiera zawartość pierwszego dopasowania wraz ze wszystkimi elementami potomnymi. Wszystkie punkty `<li>` są częścią listy `<ul>`, dlatego wszystkie zostały zwrócone. Ponieważ punkt `<li>` nie jest rodzicem dla innych elementów (nie zagnieżdża ich), zwracany jest pierwszy dopasowany element.

UWAGA

jQuery za pomocą metody `.each()` pozwala pobrać zawartość wielu elementów (listing 1.17).



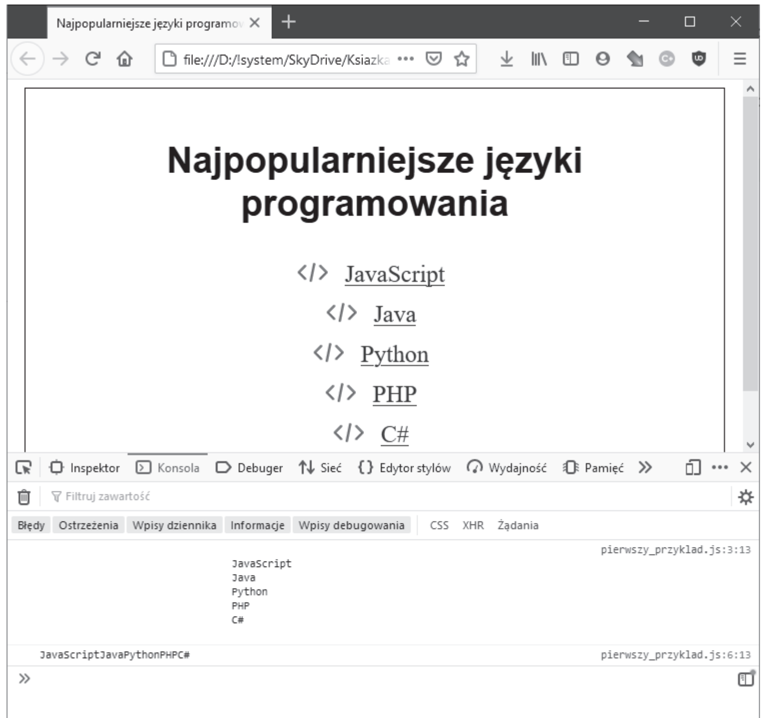
Rysunek 1.5. Pobranie zawartości elementu za pomocą metody .html()

Listing 1.11 pokazuje użycie metody `.text()`, która w sposobie działania różni się od opisanej powyżej metody `.html()`. Ponownie zostaje pobrana zawartość listy `<ul>` oraz jej elementu, `<li>`. Ta zawartość zostaje przypisana do zmiennej, której wartość jest wyświetlana w konsoli (rysunek 1.6).

Listing 1.11. Pobranie zawartości elementu — metoda .text()

```
$(function () {
    let $lista_ul = $('ul').text();
    console.log($lista_ul);

    let $lista_li = $('li').text();
    console.log($lista_li);
});
```

Tym razem użyta metoda zwraca tekst z dopasowanego zbioru i zawartych w nim elementów (lista `<ul>`), jak też zawartość wszystkich elementów (punkty `<li>`).

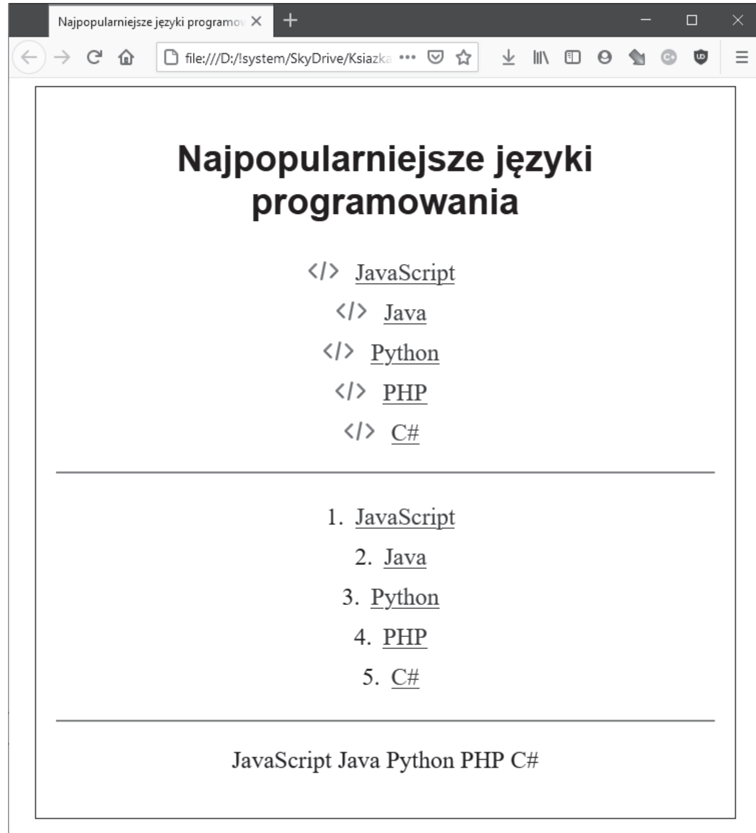


Rysunek 1.6. Pobranie zawartości elementu za pomocą metody .text()

Przykład użycia obu metod jest pokazany na listingu 1.12. Zawartość listy pobrana za pomocą metody `.html()` zostaje przypisana do zmiennej `$lista_ul`, a następnie umieszczona pomiędzy znacznikami `<ol></ol>` — powstaje lista numerowana. W następnym kroku zawartość listy zostaje pobrana ponownie, tym razem za pomocą metody `.text()`, i umieszczona w akapicie. Oba znaczniki, lista numerowana (`<ol></ol>`) i akapit (`<p></p>`), zostały dodane do kodu HTML strony. Elementy te rozdziela pozioma linia (znacznik `<hr>`). Do wstawienia pobranej zawartości elementu użyto metody `.append()`, która wstawia zawartość wewnątrz wybranego selektora przed jego zamknięciem (rysunek 1.7).

Listing 1.12. Przykład użycia metod .html() i .text()

```
$(function () {
    let $lista_ul = $('ul').html();
    $('ol').append($lista_ul);

    $lista_ul = $('ul').text();
    $('p').append($lista_ul);
});
```



Rysunek 1.7. Przykład użycia metod .html() i .text()

Wstawianie i uaktualnianie elementów

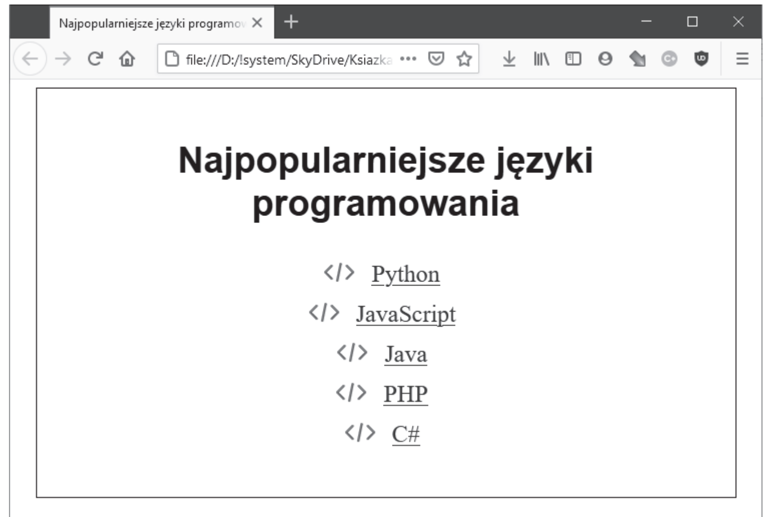
Wstawianie elementu

W poprzednim przykładzie została użyta metoda `.append()`, która wstawia pobrany kod przed znacznikiem zamykającym. jQuery udostępnia również metodę `.prepend()`, która umieszcza kod po znaczniku otwierającym.

Skrypt przedstawiony na listingu 1.13 w pierwszym kroku przypisuje do zmiennej `$zmiana` kod HTML trzeciego punktu listy. Następnie punkt ten jest usuwany i na samym końcu wstawiany po znaczniku `<ul>`, który otwiera listę (rysunek 1.8).

Listing 1.13. Użycie metody .prepend()

```
$(function () {
    let $zmiana = $('li').eq(2).html();
    $('li').eq(2).remove();
    $('ul').prepend('<li>' + $zmiana + '</li>');
});
```



Rysunek 1.8. Przykład użycia metody .prepend() w połączeniu z metodami .remove(), .html() oraz .eq()

CIEKAWOSTKA

Oprócz przedstawionych metod `.prepend()` i `.append()` istnieją jeszcze dwie: `.prependTo()` i `.appendTo()`. Obie odwracają sposób działania funkcji bazowych.

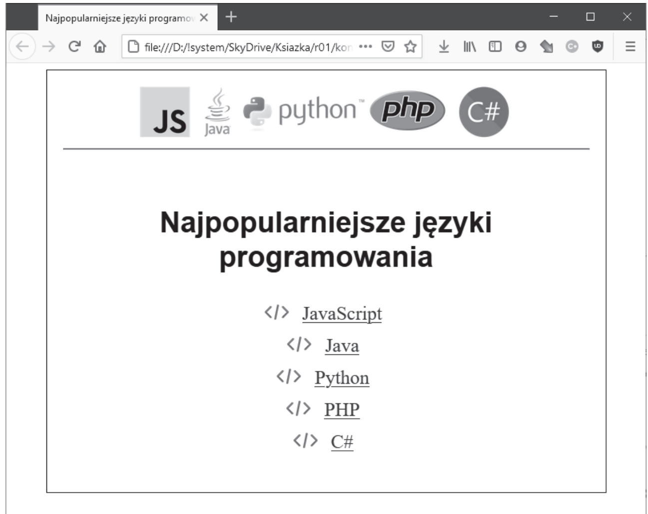
W jQuery do wstawiania elementów służą jeszcze dwie metody: `.before()` i `.after()`.

Pierwsza z nich wstawia element przed wybranym znacznikiem, druga zaś — po. Pokazany na listingu 1.14 skrypt umieszcza przed znacznikiem `<h2>` rysunek *best5.png*, a następnie wstawia po nim poziomą linię (znacznik `<hr>`) i nowy wiersz (znacznik `<br>`).

Listing 1.14. Wstawianie elementów — użycie metod .before() i .after()

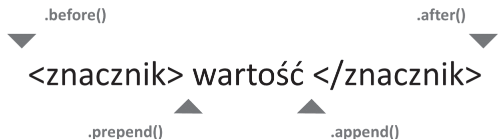
```
$(function () {
    $('h2').before('<img src="best5.png">');
    $('img').after('<hr><br>');
});
```

Wygląd strony jest pokazany na rysunku 1.9.



Rysunek 1.9. Wstawienie elementu za pomocą metod .before() i .after()

Sposób działania wszystkich omówionych metod jest pokazany na rysunku 1.10.



Rysunek 1.10. Metody wstawiania

Zadanie 1.1.

Napisz skrypt, który po naciśnięciu przycisku doda do istniejącej nienumerowanej listy nowy punkt.

Ustawienie właściwości CSS

Na listingu 1.7 została użyta metoda `.css()`, która pozwala ustawić właściwości stylu. Robi się to przez podanie właściwości CSS — argumentu w połączeniu z wartością. Oba te elementy są od siebie oddzielone **znakiem przecinka**, np. `$('p').css('background-color', 'orange')`.

Aby ustawić wiele właściwości, należy je **umieścić w nawiasie klamrowym**. Argument od wartości oddzielamy tak jak w pliku arkusza stylów, czyli za pomocą **znaku dwukropka**, a poszczególne właściwości **przecinkami**, np.:

```
$('p').css({
    'background-color': 'orange',
    'text-align': 'center',
});
```

Użycie wartości atrybutu

Ostatni przykład na listingu 1.7 pokazuje, jak wybrać element za pomocą selektora, który używa atrybutu. jQuery pozwala wyszukać i pobrać przy użyciu uproszczonych wyrażeń regularnych dowolny atrybut. Możliwe do zastosowania filtry atrybutów przedstawia tabela 1.2.

Tabela 1.2. Definicje atrybutu

Nazwa	Opis
`[atrybut]`	Element o zadanym atrybucie, którego wartość jest dowolna
`[atrybut = 'wartość']`	Element o zadanym atrybucie, którego wartość jest określona
`[atrybut != 'wartość']`	Element o zadanym atrybucie, który nie zawiera określonej wartości
`[atrybut ^= 'wartość']`	Element o zadanym atrybucie, który rozpoczyna się od podanej wartości
`[atrybut $= 'wartość']`	Element o zadanym atrybucie, który kończy się podaną wartością
`[atrybut *= 'wartość']`	Element o zadanym atrybucie, który zawiera podaną wartość
`[atrybut ~= 'wartość']`	Element o zadanym atrybucie, którego wartość znajduje się na liście (wartości rozdzielone spacjami)

UWAGA

Stosowanie tego typu rozwiązania może spowolnić działanie strony. Wybór elementów oparty na nazwie, identyfikatorze bądź klasie poprawia wydajność działania kodu JavaScript.

Aby usunąć atrybut (wraz z wartością), należy posłużyć się metodą `.removeAttr()`, np. użycie kodu `$('p').removeAttr('style');` spowoduje usunięcie stylu przypisanego do akapitu.

Zadanie 1.2.

Przygotuj stronę wyświetlającą pojedyncze pole typu *checkbox* z napisem *Akceptuję* oraz przycisk *Wyślij* (nieaktywny). Napisz skrypt, który aktywuje przycisk *Wyślij* po zaznaczeniu pola *Akceptuję*.

Dwie inne metody, `.addClass()` i `.removeClass()`, pozwalają dodać lub usunąć nazwy klasy w wartości atrybutu `class`. Są przydatne, ponieważ nie wpływają na zdefiniowane nazwy klas. Przykład użycia obu tych metod jest pokazany na listingach 1.15 i 1.16.

W pierwszym kroku do pliku arkusza stylów zostają dodane trzy nowe klasy: `.link`, `.inny_link` oraz `.jeszcze_inny_link`. Ustawiane właściwości przedstawia listing 1.15. Krokiem drugim jest modyfikacja kodu HTML — do wszystkich odnośników zostaje przypisana klasa `.link`, a czwarty odnośnik, *PHP*, dodatkowo zostaje połączony z kolejną klasą, `.jeszcze_inny_link`.

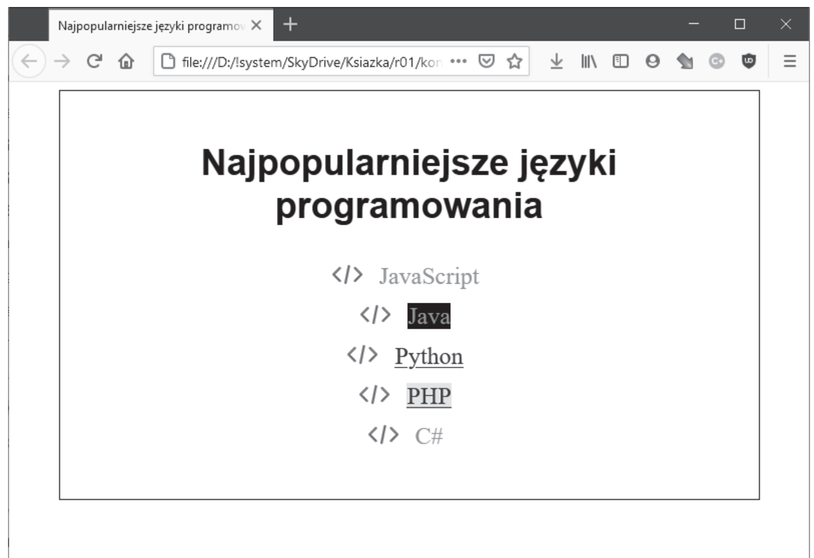
Listing 1.15. Dodanie i usunięcie klasy — plik arkusza stylów

```
.link {
    color: grey;
    text-decoration: none;
}

.inny_link {
    background-color: black;
}

.jeszcze_inny_link {
    background-color: yellow;
}
```

Użycie metody `.addClass()` dodaje do drugiego odnośnika klasę `.inny_link` (zmiana koloru tła na czarny), a metoda `.removeClass()` usuwa klasę `.link` z odnośników trzeciego i czwartego. Ponieważ do czwartego odnośnika nadal jest przypięta klasa `.jeszcze_inny_link`, formatowanie z jej wykorzystaniem wciąż obowiązuje (żółte tło). Efekt zastosowania obu metod jest pokazany na rysunku 1.11.

Listing 1.16. Dodanie i usunięcie klasy — kod jQuery

```
$(function () {
    $('a').eq(1).addClass('inny_link');
    $('a').eq(2).removeClass('link');
    $('a').eq(3).removeClass('link');
});
```



Rysunek 1.11. Efekt dodania i usunięcia klasy z użyciem metod .addClass() i .removeClass()

Zadanie 1.3.

W kodzie strony umieść akapit i przypisz do niego dwie klasy. Pierwsza niech ustawia niebieski kolor tła, a druga — wielkość czcionki na 150%. Napisz skrypt, który po wciśnięciu przycisku usunie klasy CSS formatujące akapit.

Pętla w jQuery — użycie metody .each()

Metoda `.each()` pozwala pobrać wiele elementów jednocześnie i na każdym z nich po kolei wykonać wybrane operacje. Listing 1.17 przedstawia dwa przykłady użycia tej metody.

Pierwszy fragment kodu wyświetla w konsoli przeglądarki numer pobranego elementu i jego identyfikator — wybranym elementem jest `div`. Jak można zauważyć, parametrem metody `.each()` jest **funkcja**, która wykonuje się w kontekście pobranego elementu. Dzięki temu dostęp do tego elementu może być realizowany za pomocą słowa kluczowego `this`.

WSKAZÓWKA

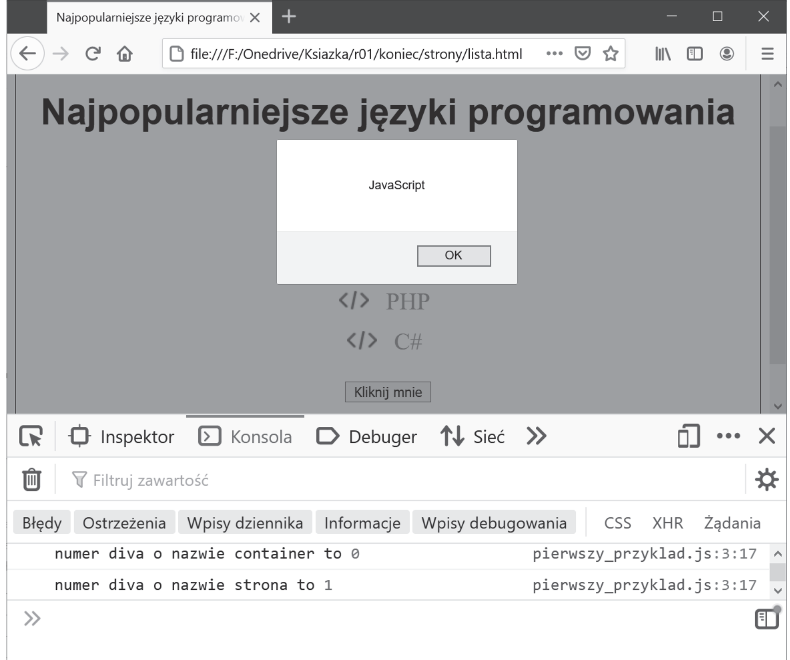
Znacznie wygodniejszym sposobem wyświetlenia tekstu w oknie konsoli jest zastosowanie notacji ze znakiem akcentu (na klawiaturze — klawisz tyldy), zatem równoznacznym zapisem jest ``console.log(`numer diva o nazwie ${this.id} to ${numer}`);``.

W drugim fragmencie kodu po naciśnięciu przycisku (do strony HTML należy najpierw dodać kod `<button>Kliknij mnie</button>`) zostanie pobrany tekst odnośnika, który zostanie wyświetlony w oknie typu *alert*. W tym przykładzie również zastosowano słowo kluczowe `this`, ale przekazano wskazywany przez `this` obiekt do jQuery za pomocą zapisu `$(this)`. Użycie takiej konstrukcji zapisu sprawia, że na elemencie bieżącym mogą być zastosowane metody jQuery (w tym przykładzie jest to metoda `.text()`). Dlatego kod z listingu 1.17 zadziała także wtedy, gdy `this.id` zamienimy na `$(this).attr('id')`.

Listing 1.17. Użycie metody .each()

```
// Pobranie id i numeru elementu
$('div').each(function(nr) {
    console.log('numer diva o nazwie', this.id, 'to', nr);
});

// Użycie przycisku
$('button').click(function() {
    $('a').each(function() {
        alert($(this).text());
    });
});
```

Oba przykłady w działaniu przedstawia rysunek 1.12.



Rysunek 1.12. Użycie metody .each()

1.1.3. Zdarzenia

Z zastosowaniem dostępnych w bibliotece jQuery metod możemy łatwo **podpiąć procedurę obsługi zdarzeń do wybranych elementów**. Kiedy następuje określone zdarzenie, zostaje wykonana przypisana do niego funkcja.

Metody zdarzeń

Aby nasłuchiwać zdarzenie do pobranego lub utworzonego elementu, używamy metody będącej **nazwą zdarzenia** lub używamy **metody** `.on()`**, do której przekazujemy procedurę obsługi zdarzeń.**

Zdarzenia w jQuery możemy podzielić na związane z **elementami interfejsu**, **klawiaturą**, **myszą**, **użyciem formularza** czy samą **przeglądarką**.

Zdarzenia związane z elementami interfejsu

Metody odpowiedzialne za efekty to `.focus()`, `.blur()` oraz `.change()`.

Na przykład wybranie za pomocą klawisza *TAB* kolejnego elementu wywołuje zdarzenie *focus*. W chwili gdy opuszczamy taki element, wykonywane jest *blur*. W momencie edytowania elementu następuje zaś zdarzenie *change*.

Obsługę wszystkich trzech zdarzeń za pomocą metod jQuery demonstruje kod pokazany na listingu 1.18.

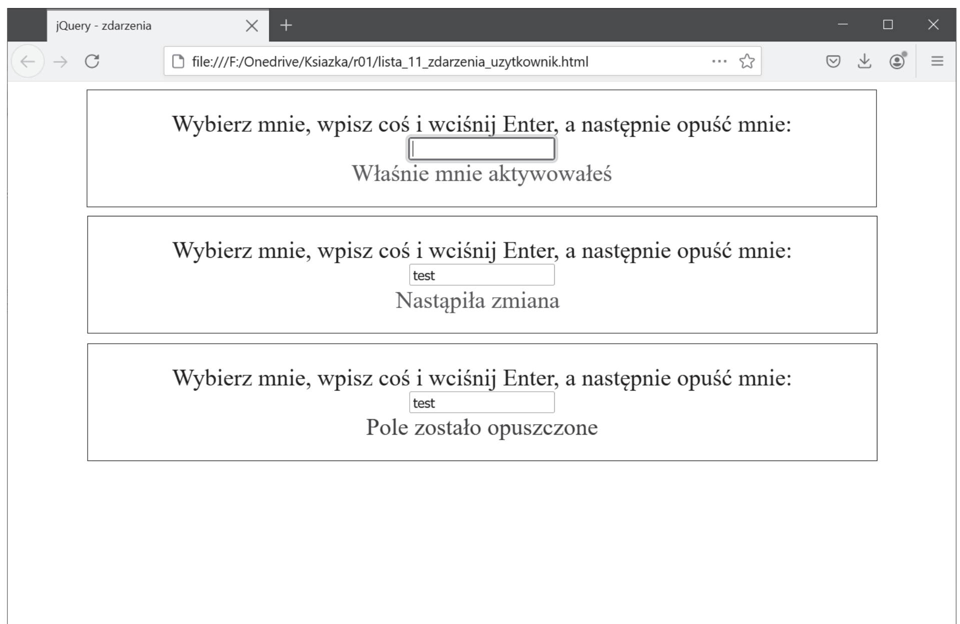
Wszystkie te metody zostały powiązane z polem formularza. Kliknięcie pola tekstowego spowoduje wyświetlenie komunikatu *Właśnie mnie aktywowałeś* (metoda `.focus()`), opuszczenie pola będzie skutkowało komunikatem *Pole zostało opuszczone* (metoda `.blur()`), a zmiana zawartości pola zatwierdzona klawiszem *Enter* wywoła komunikat *Nastąpiła zmiana* (metoda `.change()`). Treść danego komunikatu została zdefiniowana w elemencie `<span>`, a za jego wybór odpowiada metoda `.eq()`. Aby pola `<span>` po otwarciu strony nie były wyświetlane, został ustawiony styl `display: none;`. Wywołanie danej metody oprócz wyboru elementu `<span>` ustawia styl CSS. Użyty efekt `.fadeOut()` sprawi, że po czasie 4,5 s komunikat zostanie ukryty.

Listing 1.18. Użycie metod .focus(), .blur() oraz .change()

```
<html>
<head>
    <title>jQuery - zdarzenia</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <style>
        span {
            display: none;
        }
    </style>

    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function () {

            $('input').on('focus', function() {
                $('span').eq(0).css('display', 'inline').fadeOut(4500);
            });

            $('input').on('blur', function() {
                $('span').eq(1).css('display', 'inline').fadeOut(4500);
            });

            $('input').on('change', function() {
                $('span').eq(2).css('display', 'inline').fadeOut(4500);
            });

        });
    </script>
</head>
```

```
<body>
    <div id="container">
        <div id="page">
            Wybierz mnie, wpisz coś i wciśnij Enter, a następnie opuść mnie:
<input type="text"><br>
            <span>Właśnie mnie aktywowałeś</span>
            <span>Pole zostało opuszczone</span>
            <span>Nastąpiła zmiana</span>
        </div>
    </div>
</body>
</html>
```

Sposób działania kodu przedstawia rysunek 1.13.



Rysunek 1.13. Sposób działania metod .focus(), .blur() oraz .change()

Zdarzenia związane z użyciem myszki

Metody obsługujące zdarzenia związane z użyciem myszki to:

- `.click()` — następuje po kliknięciu przycisku myszy;
- `.mouseover()`— następuje po umieszczeniu wskaźnika myszy w obrębie wybranego elementu;
- `.mouseout()` — następuje po usunięciu wskaźnika myszy z wybranego elementu.

Kod pokazany na listingu 1.19 wykorzystuje wszystkie trzy metody.

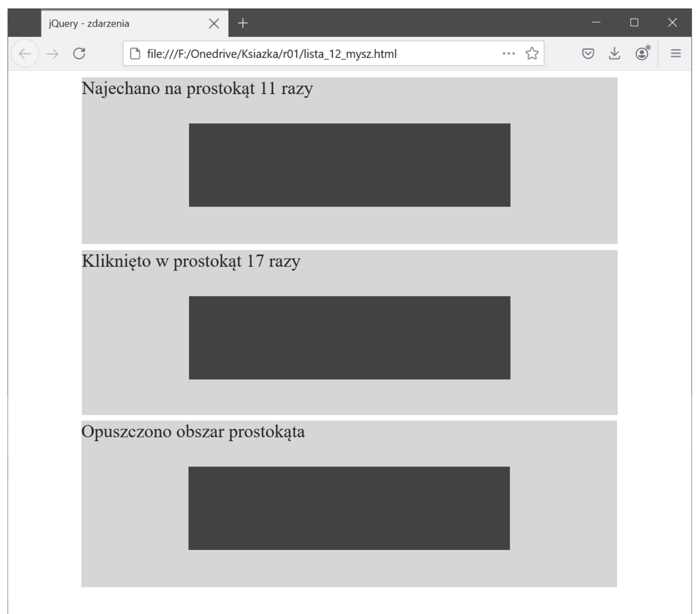
Zostają utworzone dwa prostokąty. Nakierowanie wskaźnika na wewnętrzny niebieski prostokąt, kliknięcie go bądź opuszczenie jego obszaru spowoduje wygenerowanie odpowiedniego komunikatu. Liczba zdarzeń związana z ustawieniem wskaźnika w obrębie prostokąta oraz kliknięć jest rejestrowana — treść wyświetlanego komunikatu uwzględnia liczbę wystąpień zdarzenia.

Listing 1.19. Użycie metod .click(), .mouseover() oraz .mouseout()

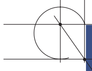
```
<html>
<head>
    <title>jQuery - zdarzenia</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <style>
        #zew {
            width: 80%;
            height: 200px;
            margin: 0 auto;
            background-color: #ccccff;
        }

        #wew {
            width: 60%;
            height: 100px;
            background-color: blue;
            margin: 30px auto;
        }

        span {
            font-size: 22px;
        }
    </style>

    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function () {

            let i = 0;
            let n = 0;
            $('#wew').mouseover(function() {
                i += 1;
                $('#komunikat').text('Najechano na prostokąt ' + i + ' razy');
            }).mouseout(function() {
                $('#komunikat').text('Opuszczono obszar prostokąta');
            });
```

```
            $('#wew').click(function() {
                n += 1;
                $('#komunikat').text('Kliknięto prostokąt ' + n + ' razy);
            });
        });
    </script>
</head>

<body>
    <div id="container">
        <div id="zew">
            <span id="komunikat">Najedź kursorem na niebieski prostokąt,
kliknij i opuść jego obszar.</span>
            <div id="wew">
            </div>
        </div>
    </div>
</body>
</html>
```

Rysunek 1.14 ilustruje sposób działania kodu.



Rysunek 1.14. Sposób działania metod .click(), .mouseover() oraz .mouseout()



WSKAZÓWKA

Oprócz omówionych metod są jeszcze inne: `.contextmenu()`, `dblclick()`, `.hover()`, `mousedown()`, `.mouseenter()`, `.mouseleave()`, `.mousemove()` oraz `.mouseup()`.

Zdarzenia związane z użyciem klawiatury

jQuery zapewnia również metody obsługi zdarzeń związanych z klawiaturą: `keydown()` i `.keypress()` — wciśnięcie klawisza, `oraz .keyup` — jego zwolnienie.

Przykładem użycia wszystkich trzech metod jest kod z listingu 1.20.

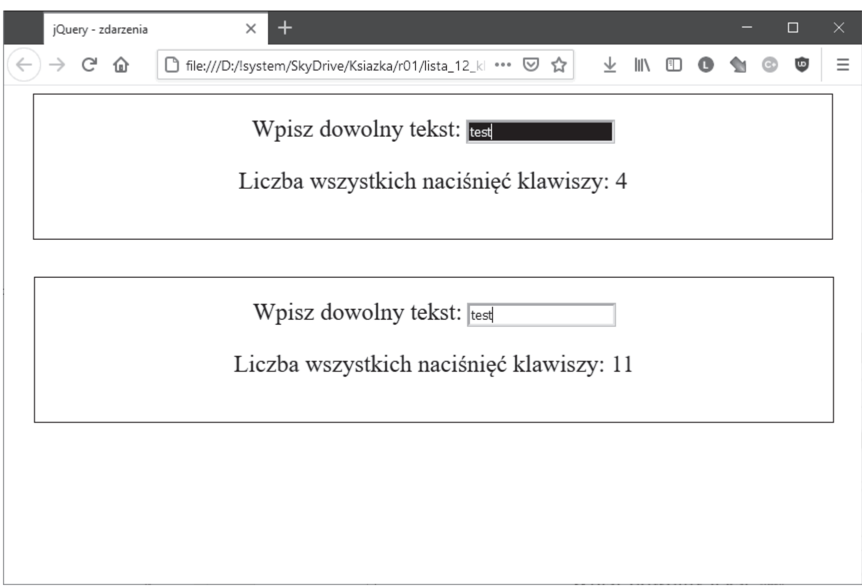
Wciśnięcie w polu formularza klawisza aktywuje metodę `.keydown()`, a powiązana z nią metoda `.css()` powoduje zmianę stylu pola `<input>` — efekt negatywu. Zwolnienie klawisza przywraca styl do ustawień początkowych. Dodatkowo metoda `.keypress()` zlicza wszystkie wciśnięcia klawiszy (nie mylić z liczbą znaków znajdujących się w polu formularza).

Listing 1.20. *Użycie metod .keydown(), .keyup() oraz .keypress()*

```
<html>
<head>
    <title>jQuery - zdarzenia</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />

    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function () {

            $('input').on('keydown', function() {
                $('input').css({
                    'background-color': 'black',
                    'color': 'white'
                });
            });

            $('input').on('keyup', function() {
                $('input').css({
                    'background-color': 'white',
                    'color': 'black'
                });
            });

            let i = 0;
            $('input').keypress(function() {
                $('span').text(i += 1);
            });

        });
    </script>
</head>
```

```
<body>
    <div id="container">
        <div id="page">
            Wpisz dowolny tekst: <input type="text"><br>
            <p>Liczba wszystkich naciśnięć klawiszy: <span>0</span></p>
        </div>
    </div>
</body>
</html>
```

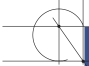
Działanie kodu z listingu 1.20 jest zilustrowane na rysunku 1.15.



Rysunek 1.15. Sposób działania metod .keydown(), .keyup() oraz .keypress()

Zadanie 1.4.

Napisz skrypt, którego zadaniem będzie wyświetlenie w konsoli tekstu wprowadzonego do pola formularza.



UWAGA

Mogłoby się wydawać, że metody `.keydown()` i `.keypress()` realizują to samo zdarzenie — rejestrują wciśnięcie klawisza na klawiaturze. Jest jednak między nimi różnica. Otóż klawisze niedrukowalne, takie jak *Shift*, *Esc* czy *Delete*, wyzwalają zdarzenie *keydown*, ale nie zdarzenie *keypress*.

Zdarzenia związane z oknem przeglądarki

Zdarzenia z tej kategorii są obsługiwane przez metody `.scroll()` i `.resize()`.

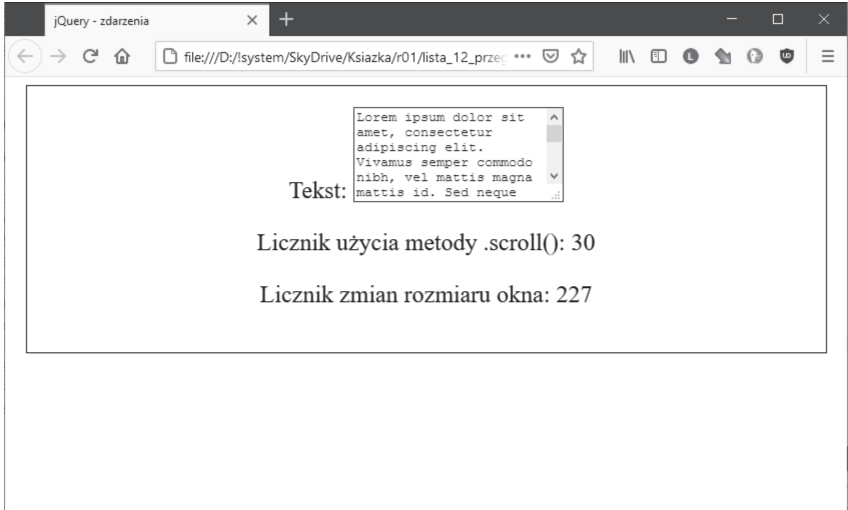
Pierwsza jest stosowana, gdy użytkownik przewija okno elementu, a druga — gdy następuje zmiana rozmiaru okna.

Kod przedstawiony na listingu 1.21 pokazuje sposób działania tych metod. Przewinięcie tekstu w polu `<textarea>` spowoduje zliczenie wystąpień zdarzenia, podobnie jak zmiana rozmiaru okna przeglądarki.

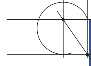
Listing 1.21. Użycie metod .scroll() i .resize()

```
<html>
<head>
    <title>jQuery - zdarzenia</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />

    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function() {
            let i = 0;
            let n = 0;
            $('#tekst').on('scroll', function() {
                $('span').eq(0).text(i += 1);
            });

            $(window).on('resize', function() {
                $('span').eq(1).text(n += 1);
            });
        });
    </script>
</head>

<body>
    <div id="container">
        <div id="page">
            Tekst: <textarea id="tekst" rows="4"
                cols="20">Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Vivamus semper commodo nibh, vel mattis magna mattis id. Sed neque
tellus, cursus quis ultrices ac, sodales id lacus. Nullam feugiat, enim nec
blandit porttitor, lacus mi venenatis felis, et sodales velit lectus vitae
tortor.</textarea><br>
            <p>Licznik użycia metody .scroll(): <span>0</span></p>
            <p>Licznik zmian rozmiaru okna: <span>0</span></p>
        </div>
    </div>
</body>
</html>
```

Działanie kodu jest zilustrowane na rysunku 1.16.



Rysunek 1.16. Sposób działania metod .scroll() i .resize()



WSKAZÓWKA

Przedstawione metody nie wyczerpują listy wszystkich dostępnych. Opis i sposób użycia metod, które nie zostały tu wymienione, znajduje się na stronie *https://api.jquery.com/category/events/*.

Obiekt event

W jQuery **obiekt event** dostarcza informacji o zaistniałym zdarzeniu. Użyta w wywołaniu funkcja pozwala odwołać się do obiektu zdarzenia.

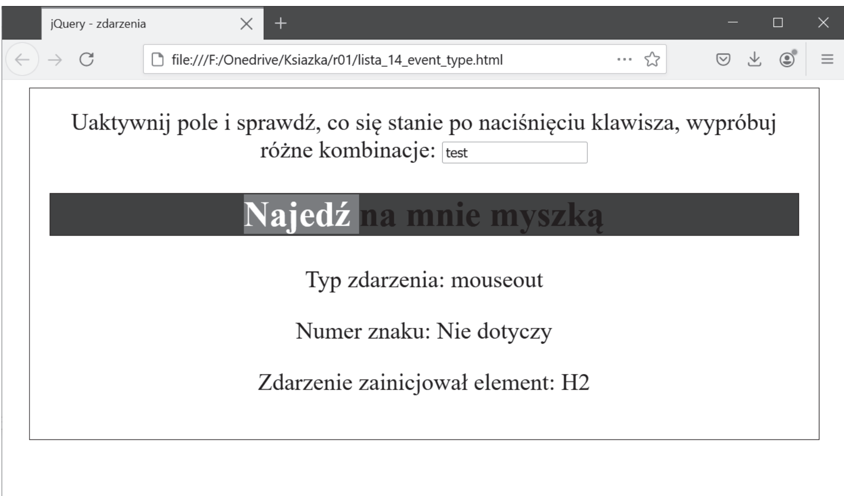
Listing 1.22 przedstawia kod, którego zadaniem jest wykrycie zdarzenia oraz dostarczenie informacji o nim. Za rozpoznanie zdarzenia odpowiada właściwość `type` (pole formularza jest odpowiedzialne za zdarzenia związane z użyciem klawiatury, a nagłówek `<h2>` to zdarzenia powiązane z użyciem myszy). Właściwość `which` identyfikuje wciśnięty klawisz, właściwość `target` dostarcza zaś informacji o elemencie, który zdarzenie zainicjował.

Listing 1.22. Obiekt event

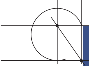
```
<html>
<head>
    <title>jQuery - zdarzenia</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />

    <style>
        h2 {
```

```
            background-color: blue;
            border: 1px solid black;
        }
    </style>

    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function() {

            $('input').on('keydown keyup keypress', function() {
                $('span').eq(0).text(' ' + event.type);
            });
            $('input').on('keydown keypress', function() {
                $('span').eq(1).text(event.which);
            });

            $('h2').on('click dblclick mouseover mouseout', function() {
                $('span').eq(0).text(' ' + event.type);
                const target = $(event.target);
                if (target.is('h2')) $('span').eq(1).text('Nie dotyczy');
            });

            $('input, h2').on('click', function() {
                $('span').eq(2).text(event.target.nodeName);
            });

        });
    </script>
</head>

<body>
    <div id="container">
        <div id="page">
            Uaktywnij pole i sprawdź, co się stanie po naciśnięciu klawisza,
wypróbuj różne kombinacje: <input
                type="text"><br>
            <h2>Najedź na mnie myszką</h2>
            <p>Typ zdarzenia: <span></span></p>
            <p>Numer znaku: <span></span></p>
            <p>Zdarzenie zainicjował element: <span></span></p>
        </div>
    </div>
</body>
</html>
```

Rysunek 1.17 ilustruje działanie skryptu przedstawionego na listingu 1.22.



Rysunek 1.17. Identyfikacja zdarzenia



WSKAZÓWKA

Oprócz przedstawionych właściwości obiektu `event` są jeszcze inne: `data` — przekazanie dodatkowej informacji w momencie wystąpienia zdarzenia; `pageX` i `pageY` — położenie myszy względem lewej i górnej krawędzi; `timeStamp` — liczba milisekund, jakie upłynęły od 1 stycznia 1970 r. Obok właściwości są dostępne również metody, których pełna lista wraz z omówieniem znajduje się pod adresem *https://api.jquery.com/category/events/*.

Zadanie 1.5.

Osadź na stronie kod HTML: `<h1>Dowolny tekst</h1><h2>Dowolny tekst</h2><p>Dowolny tekst</p><span>Dowolny tekst</span><button>Dowolny tekst</button>`. Napisz skrypt, który po kliknięciu danego elementu wypisze w oknie, jaki element został użyty. Przedstaw też drugie rozwiązanie, w którym ta sama informacja będzie wyświetlana w akapicie pod osadzonym kodem.

1.1.4. Efekty

jQuery zapewnia wiele gotowych efektów, które pozwalają uatrakcyjnić wygląd strony internetowej.

Podstawowe efekty

Podstawowe metody obsługujące efekty to:

- `.show()` — wyświetla wybrany element;
- `.hide()` — ukrywa wybrany element;
- `.toggle()` — przełącznik pozwalający wyświetlić ukryty element i ukryć wyświetlany.

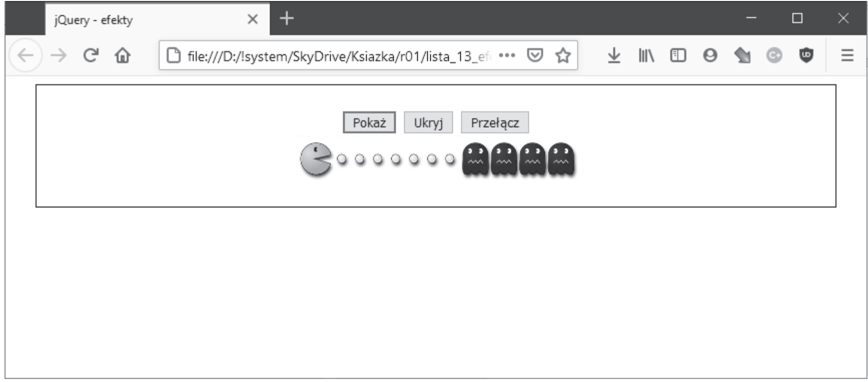
Użycie wszystkich trzech metod zaprezentowano na listingu 1.23.

Na stronie zostały umieszczone trzy przyciski. Pierwszy wyświetla zdjęcie (metoda `.show()`), drugi je ukrywa (metoda `.hide()`), trzeci zaś przełącza widok zdjęcia z ukrytego na wyświetlone i na odwrót (metoda `.toggle()`). Opcjonalnie po nazwie metody podaje się w nawiasie czas trwania efektu (w milisekundach).

***Listing 1.23.** jQuery — efekty. Użycie metod .show(), .hide() oraz .toggle()*

```
<html>
<head>
    <title>jQuery - efekty</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function() {

            $('#pokaz').on('click', function() {
                $('img').eq(0).show(1000);
            });
            $('#ukryj').on('click', function() {
                $('img').hide(500);
            });
            $('#przelacz').on('click', function() {
                $('img').toggle();
            });
        });
    </script>
</head>

<body>
    <div id="container">
        <div id="page">
            <button id="pokaz">Pokaż</button>
            <button id="ukryj">Ukryj</button>
            <button id="przelacz">Przełącz</button>
            <br><img style="display: none" src="game.png"></img>
        </div>
    </div>
</body>
</html>
```

Rysunek 1.18 przedstawia kod z listingu 1.23.



Rysunek 1.18. jQuery — efekty. Użycie metod .show(), .hide() oraz .toggle()

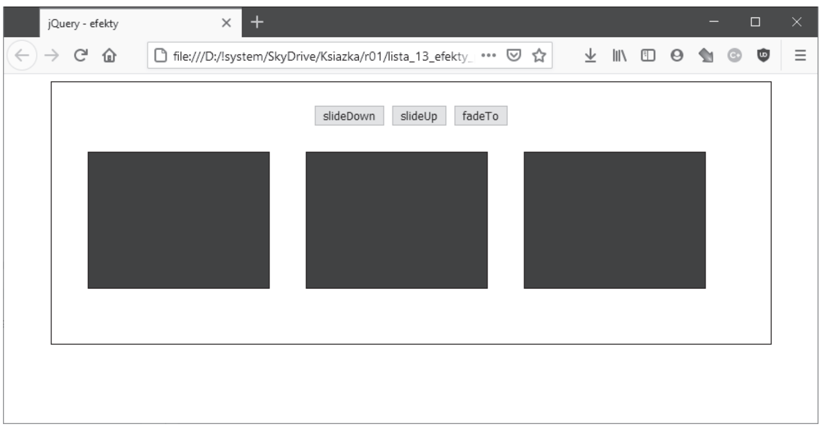
Na listingu 1.24 jest pokazane użycie efektów jQuery typu **slide** (wślizg).

Listing 1.24. jQuery — efekty. Użycie metod .slideUp(), .slideDown() oraz .slideToggle()

```
<html>
<head>
    <title>jQuery - efekty</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <style>
        .prostokat {
            border: 1px solid black;
            width: 200px;
            height: 150px;
            background: blue;
            margin: 0 20px;
            float: left;
        }
    </style>
    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function() {
            $('#wysun').on('click', function() {
                $('#pierwszy').toggle().slideDown(1000);
            });
            $('#wysun2').on('click', function() {
                $('#drugi').toggle().slideUp(2000);
            });
            $('#wysun3').on('click', function() {
                $('#trzeci').slideToggle('slow');
            });
        });
```

```
    </script>
</head>

<body>
    <div id="container">
        <div id="page" style="height: 250px">
            <button id="wysun">slideDown</button>
            <button id="wysun2">slideUp</button>
            <button id="wysun3">fadeTo</button><br><br>

            <div id="pierwszy" class="prostokat"></div>
            <div id="drugi" class="prostokat"></div>
            <div id="trzeci" class="prostokat"></div>
            <div style="clear: both"></div>
        </div>
    </div>
</body>
</html>
```

Kliknięcie przycisku z nazwą efektu uruchamia animację prostokąta (rysunek 1.19).



Rysunek 1.19. jQuery — efekty. Użycie metod .slideUp(), .slideDown() oraz .slideToggle()

Użycie animacji

jQuery umożliwia **animowanie stylów CSS**. Warunek jest jeden: wartość stylu **musi być wyrażana w postaci liczby**, np. `width`, `padding`, `font-size`. Animowane nie mogą być te właściwości, których **wartość to tekst**, takie jak `text-align` czy `font-family`.

Ogólna składnia metody `.animate()` wygląda następująco:

```
.animate({animowany_styl} <, czas_trwania> <, przebieg_animacji> <,
funkcja_wywołania_zwrotnego>),
```

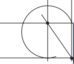
Parametr `czas_trwania` określa długość animacji (w ms), dozwolone jest użycie słów kluczowych `fast` i `slow`. Użycie zapisu `+=` doda odpowiednią wielkość do aktualnej, a `-=` odejmie. Opis i przeznaczenie pozostałych parametrów zostały zamieszczone w dalszej części rozdziału.

Kod z listingu 1.25 symuluje wyścig dwóch kulek. Animowane są takie właściwości CSS jak `left` i `top`.

Animacje są wyzwalane przez przyciski. Pierwszy, *Start niebieska*, włącza animację kulki niebieskiej, drugi, *Start czerwona* — czerwonej, a wybranie przycisku *Start obie* spowoduje jednoczesny start obu kulek. Start obu kulek jest możliwy dzięki użyciu metody `.add()`, która pozwala dodać element do istniejącej grupy elementów. Alternatywnym sposobem tworzenia grupy jest wyliczenie jej elementów, przy czym nazwy oddzielamy przecinkiem, np. `$('#start1, #start2').click();`.

Po analizie kodu można wywnioskować, że wyścig jako pierwsza ukończy kulka niebieska (na pokonanie odległości 700 px potrzebuje ona 600 ms, czerwona zaś — 800 ms), dlatego po zakończeniu animacji kulki niebieskiej uruchamiana jest funkcja (tzw. **funkcja wywołania zwrotnego**) wyświetlająca tekst *Pierwszy!!!*.

Jeśli jednocześnie są animowane dwie właściwości CSS, muszą one znajdować się w **nawiasie klamrowym i być oddzielone od siebie znakiem przecinka**.

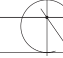


WSKAZÓWKA

Pomiędzy czasem animacji a funkcją wywołania zwrotnego może się znajdować parametr `easing`, który może przyjmować dwie wartości: `linear` i `swing`. Użycie pierwszej sprawi, że prędkość animacji pozostanie stała, a zastosowanie drugiej zwiększy prędkość animacji w środku i jednocześnie zmniejszy na początku i końcu trwania przejścia. Efekt użycia obu przełączników został zademonstrowany na stronie *https://jqueryui.com/easing/*.

Wciśnięcie przycisku *Od nowa* spowoduje wczytanie ustawień początkowych — właściwości CSS `left` i `top` zostają wyzerowane.

Ostatni przycisk, *Stop*, zatrzymuje animację. Wykorzystana metoda `.clearQueue()` usuwa z kolejki wszystkie efekty, które nie zostały jeszcze uruchomione.



WSKAZÓWKA

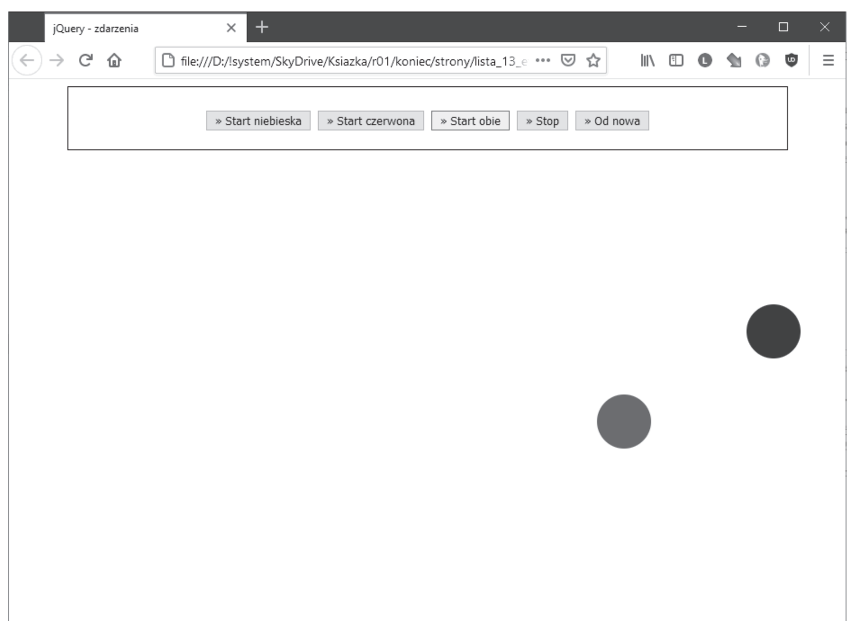
W kodzie z listingu 1.25 zostały użyte właściwości CSS, których nazwy składają się z jednego słowa. W przypadku właściwości, których nazwy składają się z wielu słów oddzielonych od siebie znakiem łącznika, np. `border-radius`, w połączeniu z metodą `.animate()` używamy notacji **camelCase** — pierwsze słowo rozpoczynamy od **małej litery**, a każde następne od **dużej**. Dla właściwości CSS `border-radius` prawidłowym zapisem będzie `borderRadius`.

Listing 1.25. jQuery — animacja

```
<html>

<head>
    <title>jQuery - zdarzenia</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <style>
        #pierwszy {
            background: blue;
            height: 60px;
            width: 60px;
            border-radius: 100px;
            left: 50px;
            top: 200px;
            position: absolute;
        }

        #drugi {
            background: red;
            height: 60px;
            width: 60px;
            border-radius: 100px;
            left: 50px;
            top: 300px;
            position: absolute;
        }
    </style>

    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">

        $(function() {

            $('#start1').on('click', function() {
                const div = $('#pierwszy');
                div.animate({ 'left': '+=300px' }, 'slow');
                div.animate({ 'top': '+=50px' }, 'slow');
                div.animate({ 'left': '+=700px' }, 600);
            });

            $('#start2').on('click', function() {
                const div = $('#drugi');
                div.animate({ 'left': '+=300px' }, 'slow');
```

```
                div.animate({ 'top': '+=50px' }, 'slow');
                div.animate({ 'left': '+=700px' }, 800);
            });

            $('#start3').on('click', function() {
                $('#start1, #start2').click();
            });

            $('#reset').on('click', function() {
                $('#log').text('');
                $('#pierwszy, #drugi').css({ left: '', top: '' });
            });

            $('#stop').click(function() {
                const kulka = $('.kulka');
                kulka.clearQueue();
                kulka.stop();
            });
        });

    </script>
</head>

<body>
    <div id="container">
        <div id="page">
            <button id="start1">&raquo; Start niebieska</button>
            <button id="start2">&raquo; Start czerwona</button>
            <button id="start3">&raquo; Start obie</button>
            <button id="stop">&raquo; Stop</button>
            <button id="reset">&raquo; Od nowa</button>

            <div id="pierwszy" class="kulka"></div>
            <div id="drugi" class="kulka"></div>
            <div id="log"></div>
        </div>
    </div>
</body>

</html>
```

Kod z listingu 1.25 jest zilustrowany na rysunku 1.20.



Rysunek 1.20. jQuery — animacja

Zadanie 1.6.

Utwórz akapit z wpisanym własnym imieniem. Za pomocą kodu CSS ustaw szerokość akapitu, tło oraz obramowanie. Pod akapitem dodaj dwa przyciski: *Start* i *Przywróć*. Wykonaj animację akapitu zmieniającą szerokość akapitu, wielkość użytej czcionki oraz wielkość obramowania. Animacja jest uruchamiana po wciśnięciu przycisku *Start*, a resetowana po naciśnięciu przycisku *Przywróć*.

Zadanie 1.7.

Przygotuj stronę, która będzie wyświetlać dwa przyciski: *W prawo* i *W lewo*. Pod przyciskami ma zostać umieszczony kwadrat (utworzony za pomocą znacznika `<div>`). Napisz skrypt, który przesunie kwadrat o wartość 80 px. Kierunek przesunięcia zależy od tego, który przycisk zostanie wybrany.

1.1.5. Formularz

W jQuery nie zapomniano o obsłudze **formularzy** — szczególnie ważnego elementu każdej strony internetowej. Dlatego udostępniono metody, które to ułatwiają.

Elementy, zdarzenia i metody formularza sieciowego

jQuery zapewnia szereg selektorów, które pozwolą uchwycić żądany element formularza. Stosuje się je w połączeniu z **nazwą elementu CSS bądź jQuery**.

Do najczęściej wykorzystywanych selektorów należą:

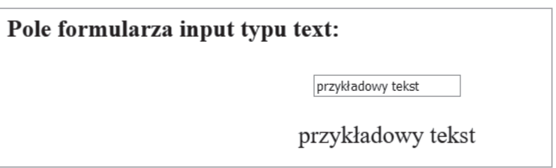
- :button — elementy <button> i <input> z atrybutem type, który ma wartość button;
- :checkbox — element <input>, którego atrybut type ma wartość checkbox;
- :disabled/:enabled — elementy, które zostały wyłączone/włączone;
- :input — elementy <button>, <input>, <select> oraz <textarea>;
- :password — element <input>, którego atrybut type ma wartość password;
- :radio — element <input>, którego atrybut type ma wartość radio;
- :submit — element <input> bądź <button>, którego atrybut type ma wartość submit;
- :text — element <input>, którego atrybut type ma wartość text.

W połączeniu z przedstawionymi selektorami najczęściej wykorzystywanymi metodami będą .val(), której zadaniem jest pobranie wartości z pola formularza, oraz .is(), która sprawdzi, czy zaznaczono bądź wybrano pole formularza.

Praca z formularzem

Listing 1.26 pokazuje szereg pól formularza, od typu text, poprzez pole listy i pole wielokrotnego wyboru, po pola typu radio i checkbox.

Pierwszym polem formularza jest pole tekstowe. Skrypt pobiera zawartość pola tekstowego po każdym zwolnieniu klawisza i wyświetla ją w akapicie umieszczonym pod polem (rysunek 1.21).



Rysunek 1.21. jQuery — formularz, pole tekstowe

Drugim polem formularza jest lista jednokrotnego wyboru. Wybranie z listy pozycji *Stron internetowych* spowoduje wyświetlenie formularza z polami typu *radio.* Nazwa wybranej opcji zostanie wyświetlona w wierszu pod listą (rysunek 1.22).

Pole jednokrotnego wyboru:

Języki programowania wykorzystujesz do tworzenia: Stron internetowych

Twój wybór to: Stron internetowych

PHP ○ JavaScript ○

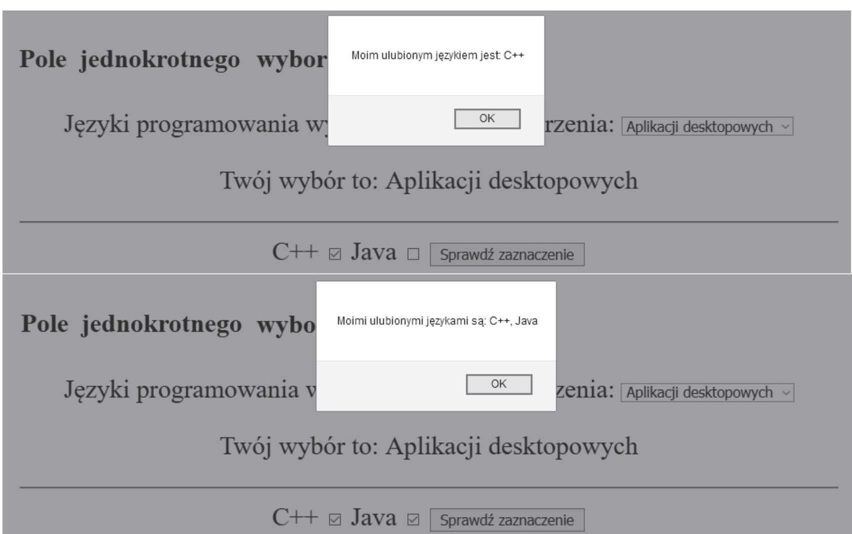
Rysunek 1.22. jQuery — formularz, lista jednokrotnego wyboru oraz pola typu radio

WSKAZÓWKA

Aby wyświetlić pojedynczy apostrof lub cudzysłów, należy użyć **znaku ucieczki** — odwrotnego ukośnika (\).

Wybranie z listy pozycji *Aplikacji desktopowych* spowoduje wyświetlenie formularza z polami typu *checkbox*. Wybranie języka (bądź języków) i kliknięcie przycisku *Sprawdź zaznaczenie* spowoduje wyświetlenie w oknie typu *alert* wybranych pozycji. Za przechowywanie wartości zaznaczonych pól odpowiada tablica (zmienna `favorite`) w połączeniu z metodą `.push()`, która dodaje zaznaczone pola do tablicy. Użycie liczby pojedynczej bądź mnogiej w komunikacie wyświetlanym w oknie typu *alert* jest możliwe dzięki **operatorowi warunkowemu**, który skraca instrukcję `if`. Metoda `.join()` scala wartości zaznaczonych pól, oddzielając jedno od drugiego przecinkiem.

Efekt działania tego fragmentu skryptu jest pokazany na rysunku 1.23.



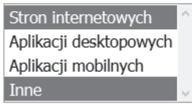
Rysunek 1.23. jQuery — formularz, lista jednokrotnego wyboru oraz pola typu checkbox

Wybranie z listy każdej innej pozycji spowoduje wyświetlenie rysunku.

Ostatnim polem formularza jest lista wielokrotnego wyboru, która pozwala wybrać kilka opcji z listy. Nazwa wybranej opcji jest wyświetlana również poniżej listy (rysunek 1.24).

Pole wielokrotnego wyboru:

Pytanie pozostaje to samo, lecz można wybrać kilka opcji:



Twój wybór to: Stron internetowych, Inne

Rysunek 1.24. jQuery — formularz, lista wielokrotnego wyboru

Listing 1.26. jQuery — formularz

```
<html>

<head>
    <title>jQuery - formularz</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="styl.css" />
    <style>
        h4 {
            text-align: left;
        }
    </style>
    <script src="jquery-3.6.0.js"></script>
    <script type="text/javascript">
        // Formularz — pole tekstowe
        $(function () {
            $('input').on('keyup', function () {
                const wartosc = $(this).val();
                $('p').eq(0).text(wartosc);
            }).keyup();
            // Pole listy jednokrotnego wyboru
            function wyswietlPojedycznyWybor() {
                const pojedynczyWybor = $('#tylkoJeden').val();
                $('p').eq(1).text('Twój wybór to: ' + pojedynczyWybor);
            }
            $('select').change(wyswietlPojedycznyWybor);
            wyswietlPojedycznyWybor();
            // Formularz z polami radio
            function wyborJezyka() {
                const jakiJezyk = $('#tylkoJeden').val();
                if (jakiJezyk == 'Stron internetowych') {
```

```
                    $('#log').text('');
                    $('#ulubionyJezyk').html('PHP <input type="radio"
name="www" value="PHP" > JavaScript <input type="radio" name="www"
value="JavaScript" >');
                    $('input').on('click', function () {
                        $('#log').html('Wybrano: ' + $('input:checked').val());
                    });
                }
                // Formularz z polami checkbox
                else if (jakiJezyk == 'Aplikacji desktopowych') {
                    $('#log').text('');
                    $('#ulubionyJezyk').html('C++ <input type="checkbox"
name="desktop" value="C++" > Java <input type="checkbox" name="desktop"
value="Java">  <button type="button">Sprawdź zaznaczenie</button>');
                    // Komunikat w oknie alert
                    $('button').on('click', function () {
                        const favorite = [];
                        $.each($('input[name="desktop"]:checked'), function ()
{
                            favorite.push($(this).val());
                        });

                        const ilosc = function () {
                            const n = $('input:checked').length;
                            const x = (n === 1 ? ' Moim ulubionym językiem
jest: ' : ' Moimi ulubionymi językami są: ');
                            alert(x + favorite.join(', '));
                        };
                        ilosc();
                    });
                }
                else {  // Wyświetlenie rysunku
                    $('#log').text('');
                    $('#ulubionyJezyk').html(' <img src="best5.png">');
                }
            }
            $('select').change(wyborJezyka);
            wyborJezyka();
            // Pole listy wielokrotnego wyboru
            function wyswietlWyborWielokrotny() {
                const wielokrotnyWybor = $('#wiele').val();
                $('p').eq(2).text('Twój wybór to: ' + wielokrotnyWybor.join(',
'));
            }
            $('select').change(wyswietlWyborWielokrotny);
            wyswietlWyborWielokrotny();
```

```
        });
    </script>
</head>

<body>
    <div id="container">
        <div id="page">
            <h4>Pole formularza input typu text:</h4>
            <input type="text" value="przykładowy tekst">
            <p></p>
            <hr>
            <h4>Pole jednokrotnego wyboru:</h4>
            Języki programowania wykorzystujesz do tworzenia: <select
id="tylkoJeden">
                <option>Stron internetowych</option>
                <option>Aplikacji desktopowych</option>
                <option>Aplikacji mobilnych</option>
                <option>Inne</option>
            </select>
            <p></p>
            <hr>
            <div id="ulubionyJezyk"></div>
            <div id="log"></div>
            <hr>
            <h4>Pole wielokrotnego wyboru:</h4>
            Pytanie pozostaje to samo, lecz można wybrać kilka opcji: <br><br>
            <select id="wiele" multiple>
                <option>Stron internetowych</option>
                <option>Aplikacji desktopowych</option>
                <option>Aplikacji mobilnych</option>
                <option>Inne</option>
            </select>
            <p></p>
            <hr>
        </div>
    </div>
</body>

</html>
```

Zadanie 1.8.

Zaproponuj i stwórz własny formularz kontaktowy. Formularz ma zawierać pola *login użytkownika* i *adres e-mail*, listę rozwijaną z polami *dział techniczny*, *dział księgowy* i *inne* oraz pole opisu, w którym będzie możliwe dodanie uwag.

Zadanie 1.9.

Wyświetl pięć pól typu *checkbox* z opisami: *biały*, *czarny*, *zielony*, *niebieski* oraz *czerwony*. Po zaznaczaniu pola/pól w akapicie poniżej niech pojawia się informacja o liczbie zaznaczeń. Po zaznaczeniu jednego pola powinien się wyświetlać komunikat *Zaznaczono 1 pole*, a po zaznaczeniu wielu — *Zaznaczono <liczba_zaznaczeń> pól*.

Zadanie 1.10.

Wyświetl pięć pól typu *radio* z opisami: *biały*, *czarny*, *zielony*, *niebieski* oraz *czerwony*. Po zaznaczeniu pola w akapicie poniżej niech pojawia się informacja z opisem wybranego pola.

1.1.6. jQuery — podsumowanie

Przedstawione w tym rozdziale treści nie wyczerpują tematu wykorzystania biblioteki jQuery. Pełną listę aktualnych funkcji wraz z dokumentacją i przykładami znajdziesz na stronie: *https://api.jquery.com/*. Witryna dostarcza informacji o wszystkich dostępnych właściwościach i metodach — także nowych i przestarzałych (ang. *deprecated*).

Serwis *https://www.w3schools.com/* zawiera ogromną bazę nie tylko przykładów, opisów metod i właściwości biblioteki jQuery, ale również użycia języków HTML, JavaScript czy CSS.

Ponadto funkcjonalność oryginalnej biblioteki jQuery zwiększają wtyczki, inaczej **pluginy** (ang. *plug-in*). Wtyczki to gotowe skrypty, które automatyzują określone zadania, np. usprawniają działanie formularzy, pozwalają odtworzyć treści audio/wideo czy przygotować animację. Można je pobrać ze strony *https://plugins.jquery.com/*.

Zadanie 1.11.

Zapoznaj się z darmowym kursem jQuery dostępnym na stronie *http://www.w3schools.com/jquery/default.asp*.

Pytania kontrolne

1. Jakimi sposobami można chwycić żądany element?
2. Jakimi metodami możemy się posłużyć, aby wybrać dany element z grupy?
3. Co robią metody `.html()` i `.text()`? Wytłumacz różnicę w ich sposobie działania.
4. Jakiej metody użyjesz, aby zaktualizować styl CSS elementu?
5. W jaki sposób pobierzesz element, który ma przypisany atrybut o określonej wartości? A w jaki sposób, gdy atrybut nie zawiera wartości?
6. W jaki sposób użycie metod `.addClass()` i `.removeClass()` wpływa na klasy zdefiniowane w arkuszu stylów?
7. Jaka metoda pozwala zdefiniować szereg instrukcji, które zostaną powtórzone wielokrotnie?

8. Czym jest zdarzenie i z jakimi elementami ten mechanizm może zostać połączony?
9. Jakich informacji dostarcza obiekt `event`?
10. Za co są odpowiedzialne efekty?
11. W jaki sposób przebiega animacja w jQuery?
12. W jaki sposób można uchwycić elementy formularza?
13. Wskaż metody, które mogą posłużyć do obsługi zdarzeń w formularzu.

1.2. Framework Bootstrap

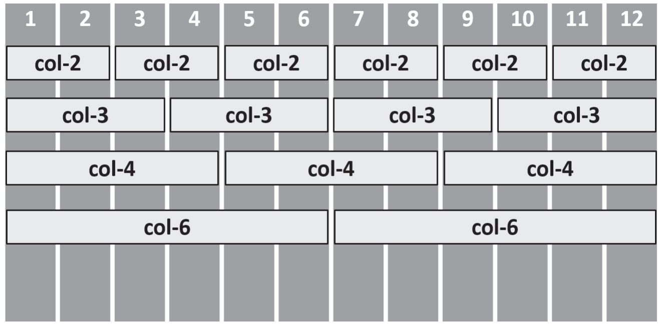
1.2.1. Czym jest Bootstrap?

Bootstrap jest gotowym szkieletem, który ułatwia stworzenie wizualnej części strony internetowej. Jest jednym z najpopularniejszych rozwiązań tego typu. Stworzyli go programiści Twittera Mark Otto i Jacob Thornton. Udostępniono go w 2011 r. i od tego czasu zyskał ogromną popularność, ponieważ umożliwia szybkie budowanie responsywnych stron. Dodatkowym atutem tego frameworka jest jego licencja, która pozwala na jego użycie również w komercyjnych projektach.

Oprócz szybkości tworzenia responsywnych witryn Bootstrap zapewnia możliwość stylizowania z zastosowaniem gotowych klas oraz obsługuje, podobnie jak jQuery, wtyczki. Popularność Bootstrapa sprawia, że w internecie jest dostępna bardzo duża liczba poradników opisujących jego sposób działania, a dzięki wsparciu społeczności w razie problemów szybko znajdziemy rozwiązanie.

1.2.2. Bootstrap — idea siatki

Siatka Bootstrapa standardowo składa się **maksymalnie z 12 kolumn**, które są rozdzielone odstępami. Sprawia to, że niezależnie od szerokości użytego ekranu zostanie on podzielony na dwanaście równych części. Takie podzielenie ekranu pozwala tworzyć kolumny, których szerokość może wahać się od 1/12 ekranu do 12/12 (rysunek 1.25).



Rysunek 1.25. Kolumny w Bootstrapie

Użycie klasy `col-2` spowoduje zarezerwowanie przez pojemnik 2 kolumn siatki, co oznacza, że na całej szerokości ekranu takich pojemników może być 6 — każdy z nich zajmuje 1/6 ekranu.

Zastosowanie klasy `col-3` spowoduje użycie przez pojemnik 3 kolumn, tak więc w wierszu mogą zostać umieszczone 4 identyczne pojemniki, z których każdy zajmuje 1/4 ekranu.

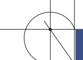
To samo dotyczy użycia klasy `col-4` (każdy pojemnik zajmuje 1/3 ekranu) oraz `col-6` (pojemniki zajmują po połowie ekranu).

Ponieważ siatka ma automatycznie reagować na zmianę szerokości ekranu i dostosowywać swój kształt do rozmiaru ekranu urządzenia, na którym jest wyświetlana, wprowadzono progi zmian. Bootstrap definiuje 6 szerokości ekranu, przy których może nastąpić zmiana układu siatki:

- *extra small*, do 575 px — klasa `.col-`;
- *small*, od 576 px do 767 px — klasa `.col-sm-`;
- *medium*, od 768 px do 991 px — klasa `.col-md-`;
- *large*, od 992 px do 1199 px — klasa `.col-lg-`;
- *extra large*, od 1200 px do 1399 px — klasa `.col-xl-`;
- *extra extra large* — od 1400 px — klasa `.col-xxl-`.

Przykładowo klas `class="col-12 col-sm-6 col-md-4 col-lg-3 col-xl-2"` spowoduje, że pojemnik na urządzeniu o rozmiarze ekranu *extra small* zajmie całą jego powierzchnię, *small* — połowę, *medium* — 1/3, *large* — 1/4, a na największym — 1/6.

Aby uprościć definicję klas, stosuje się skróty. Przykładowo zapis `class="col-md-4"` sprawia, że pojemnik zajmie 1/3 ekranu nie tylko na urządzeniach z wyświetlaczem *medium*, ale także na większych — również tych z wyświetlaczami *large*, *extra large* i *extra extra large*. Zapis klasy `col-12` można pominąć, ponieważ na tego typu ekranie wszystkie bloki siłą rzeczy zostaną ustawione pionowo, jeden pod drugim.



WSKAZÓWKA

Odstępami (tzw. gutterami) pomiędzy blokami/pojemnikami w poszczególnych widokach nie musimy się przejmować — framework sam o nie zadba (ustawi je na domyślną szerokość wynoszącą 30px).

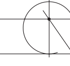
Połączenie Bootstrapa z witryną

Wersję 5.x.x frameworka można pobrać po przejściu do sekcji *Download* na stronie *https://getbootstrap.com/* (dostępne są również wersje wcześniejsze).

Po rozpakowaniu pliku zostaną utworzone dwa katalogi. Pierwszy, *css*, zawiera arkusze stylów, a drugi, *js* — pliki skryptów JavaScript. W obu znajdziesz dwie wersje plików: pełną, z komentarzami do kodu, oraz oznaczoną jako *min*, pozbawioną wszystkich zbędnych znaków.

Lokalne podpięcie arkusza stylów następuje po dodaniu linii `<link href="css/bootstrap.min.css" rel="stylesheet">`, a skrypt dołączamy przed zamknięciem sekcji `<body>`: `<script src="js/bootstrap.min.js"></script>`.

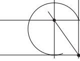
Jeśli do podpięcia zasobów posłużymy się mechanizmem CDN, aktualne odnośniki znajdziemy po przejściu na stronę *https://getbootstrap.com/docs/5.0/getting-started/introduction/*.



WSKAZÓWKA

W sytuacji gdy używamy więcej niż jednego arkusza stylów CSS, arkusz frameworka podpinamy jako pierwszy. Mamy wtedy pewność, że definicje stylów zawarte w arkuszu nie zostaną nadpisane przez wartości domyślne z arkusza frameworka.

Aby framework mógł osiągnąć pełnię swoich możliwości, należy również podpiąć wtyczkę Popper, odpowiedzialną za pozycjonowanie wyskakujących okien. Połączenie lokalne wykonamy po pobraniu niezbędnych elementów i umieszczeniu w kodzie projektu odnośnika do nich. Innym sposobem połączenia frameworka z wymaganymi wtyczkami jest użycie mechanizmu CDN.



WSKAZÓWKA

Wtyczka Popper znajduje się w pliku *bootstrap.bundle.min.js*, dlatego nie musimy dołączać oddzielnie skryptu Bootstrap i wtyczki Popper — wystarczy zamiast *bootstrap.min.js* dołączyć *bootstrap.bundle.min.js*.

Oprócz tego w sekcji `<head>` należy umieścić tag `<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">`. Dodanie kodu sprawi, że szerokość dostępnego miejsca zostanie ustawiona automatycznie w zależności od użytego urządzenia. Zapis `initial-scale=1` nakazuje wyświetlenie strony bez skalowania (przybliżania czy oddalania), a argument `shrink-to-fit` ustawiony z flagą `no` zabrania zmniejszenia w celu dopasowania.

Przygodę z frameworkiem Bootstrap najłatwiej rozpocząć od pobrania szablonu przygotowanego przez jego twórców. Szablon ten jest dostępny na stronie frameworka jako *Starter Template*. Aby móc korzystać z wszystkich funkcjonalności Bootstrapa, wystarczy przekopiować umieszczony tam kod.

1.2.3. Praca z Bootstrapem

Listing 1.27 przedstawia prostą stronę wykonaną z zastosowaniem frameworka Bootstrap (użyto szablonu), a listing 1.28 — dodatkowy arkusz stylów, który połączono ze stroną.

Listing 1.27. Zawartość pliku strony

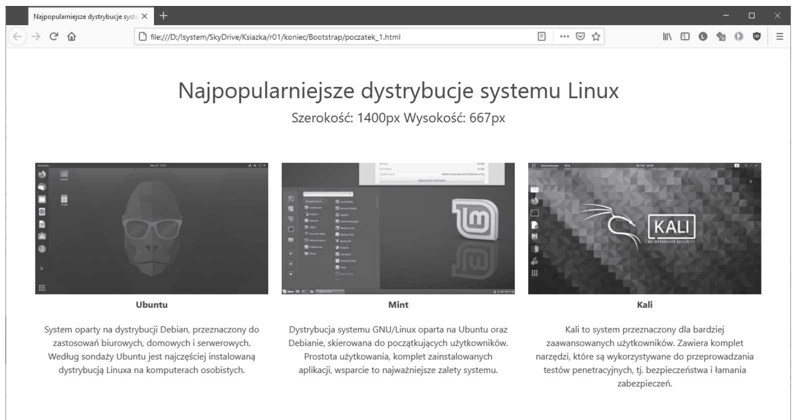
```
<!doctype html>
<html lang="pl">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-
to-fit=no">

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/
bootstrap.min.css" rel="stylesheet"
        integrity="sha384-+0n0xVW2eSR5OomGNYDnhzAbDsOXxcvSN1TPprVMTNDbiYZCxYbOO
l7+AMvyTG2x" crossorigin="anonymous">
    <link rel="stylesheet" href="style.css">

    <title>Najpopularniejsze dystrybucje systemu Linux</title>

    <script>
        function getSize() {
            const w = document.documentElement.clientWidth;
            const h = document.documentElement.clientHeight;

            document.getElementById('wh').innerHTML = "Szerokość: " + w + "px
Wysokość: " + h + "px";
        }
    </script>
</head>

<body onload="getSize()" onresize="getSize()">
    <div class="container">
        <div class="starter-template text-center py-5">
            <h1>Najpopularniejsze dystrybucje systemu Linux</h1>
            <p style="font-size: 24px" id="wh"></p>
        </div>

        <div class="linux row">
            <div class="col-sm-6 col-md-4">
                <a href="#"><img src="img/ubuntu.jpg" alt="Ubuntu"></a>
                <p class="akapit">Ubuntu</p>
                <p class="px-2 py-1 text-center">System oparty na dystrybucji
Debian, przeznaczony do zastosowań
                    biurowych, domowych i serwerowych. Według sondaży Ubuntu
jest najczęściej instalowaną dystrybucją
                    Linuxa na komputerach osobistych. </p>
            </div>
```

```
            <div class="col-sm-6 col-md-4">
                <a href="#"><img src="img/mint.jpg" alt="Mint"></a>
                <p class="akapit">Mint</p>
                <p class="px-2 py-1 text-center">Dystrybucja systemu GNU/Linux
oparta na Ubuntu oraz Debianie,
                    skierowana do początkujących użytkowników. Prostota
użytkowania, komplet zainstalowanych aplikacji,
                    wsparcie to najważniejsze zalety systemu.</p>
            </div>
            <div class="col-sm-6 col-md-4"><a href="#"><img src="img/kali.jpg"
alt="Kali"></a>
                <p class="akapit">Kali</p>
                <p class="px-2 py-1 text-center">Kali to system przeznaczony
dla bardziej zaawansowanych użytkowników.
                    Zawiera komplet narzędzi, które są wykorzystywane do
przeprowadzania testów penetracyjnych, tj.
                    bezpieczeństwa i łamania zabezpieczeń.</p>
            </div>
        </div>
    </div>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/js/
bootstrap.bundle.min.js"
        integrity="sha384-gtEjrD/SeCtmISkJkNUaaKMoLD0//ElJ19smozuHV6z3Iehds+3U
lb9Bn9Plx0x4"
        crossorigin="anonymous"></script>
</body>

</html>
```

Listing 1.28. Zawartość pliku arkusza stylów — style.css

```
.linux img {
    width: 100%;
}

.akapit {
    text-align: center;
    font-weight: 700;
    margin-top: 5px;
}
```

Zawartość strony jest pokazana na rysunku 1.26.



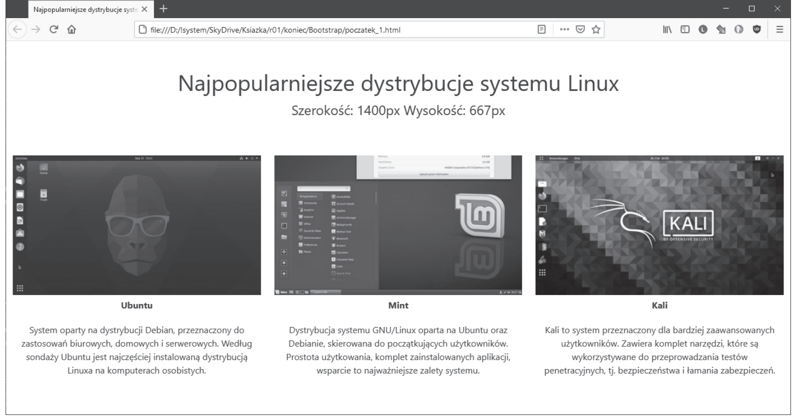
Rysunek 1.26. Wygląd strony

Strona zawiera trzy rysunki z opisami. Dodatkowo dołączono skrypt pokazujący bieżące wymiary okna, tak aby lepiej zobrazować działanie Bootstrapa.

Tworzenie strony rozpoczynamy od utworzenia zbiorczego pojemnika należącego do klasy `container`. Pojemnik ten opakowuje wszystkie użyte elementy strony. Nazwa klasy nie jest przypadkowa, ponieważ została wstępnie zdefiniowana i jest niezbędna do działania strony opartej na frameworku.

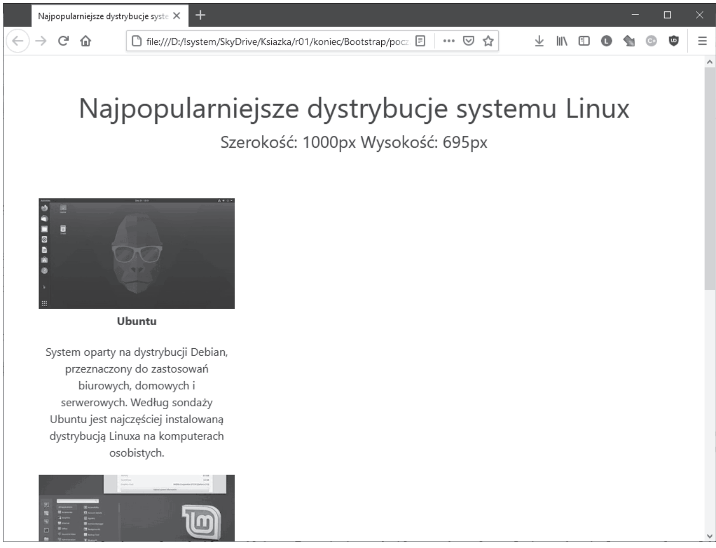
Oprócz klasy `container` istnieje jeszcze jedna klasa — `container-fluid`. Użycie `container` doprowadzi do maksymalnego rozszerzenia zawartości strony, ale tylko do widoku *extra large* — dalsze rozszerzanie okna nie spowoduje skalowania zawartości strony.

Szerokość wyświetlonej strony na rysunku 1.26 to 1400 px i po obu bokach można już zauważyć wolną przestrzeń. Użycie klasy `container-fluid` spowodowało jej wypełnienie przez zawartość strony (rysunek 1.27).



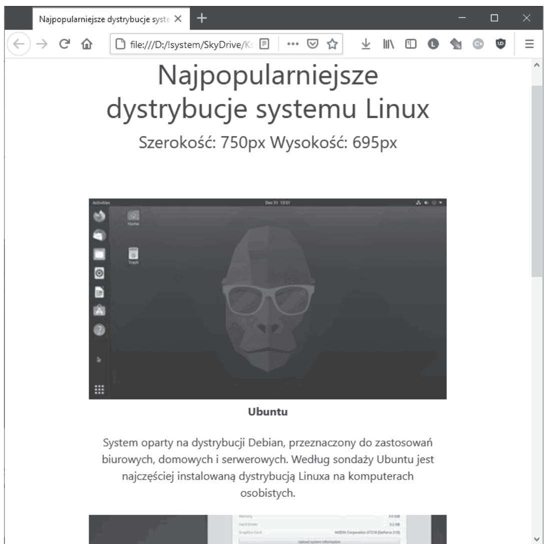
Rysunek 1.27. Użycie klasy container-fluid

Elementy strony dodatkowo muszą zostać umiejscowione w bloku przynależnym do klasy `row`, który należy traktować jako wiersz dla kolumn siatki Bootstrapa. Pominięcie powiązania spowoduje rozmieszczenie elementów strony jeden pod drugim, co pokazano na rysunku 1.28.



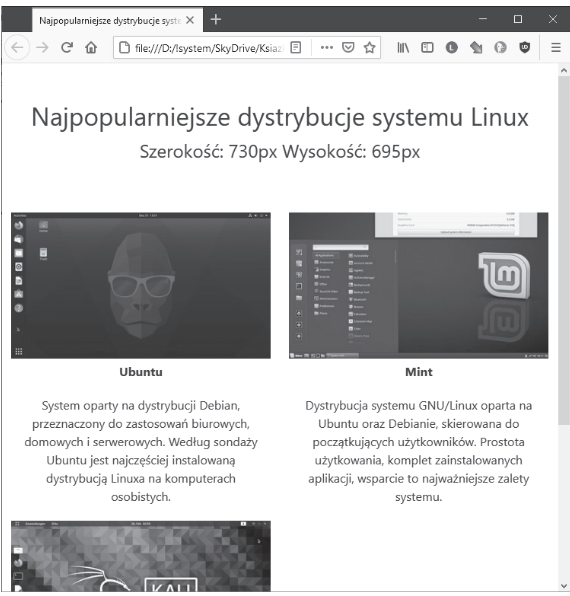
Rysunek 1.28. Efekt braku bloku należącego do klasy row

Blokom, w których znajdują się zdjęcie i opis, zostaje przypisana klasa `col-md-4`, co oznacza, że jeżeli widok strony obejmie szerokość w zakresie ekranu *medium* i większych, bloki te będą zajmować 1/3 ekranu. Poniżej widoku ekranu *medium*, czyli gdy szerokość wyniesie mniej niż 768 px, bloki ułożą się jeden pod drugim (rysunek 1.29).

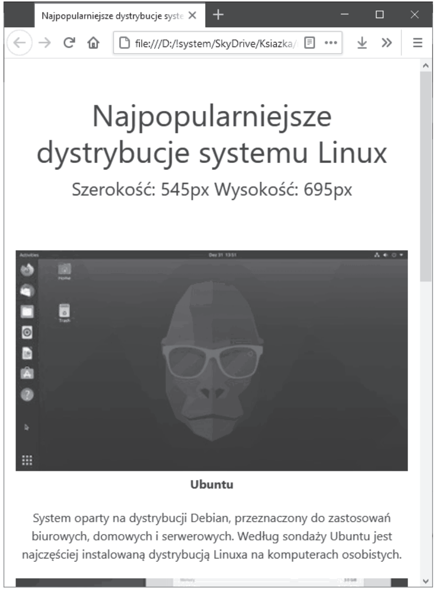


Rysunek 1.29. Zakres szerokości dla ekranu small — bloki zmieniły swoje położenie na jeden pod drugim

Poprzedzenie w każdym bloku klasy `col-md-4` klasą `col-sm-6` sprawi, że układ trzech bloków dla ekranów *medium* i szerszych nadal będzie obowiązywać, zmieni się natomiast zasada dla ekranów *small* — bloki zajmą po połowie ekranu (rysunek 1.30), a dopiero w zakresie szerokości *extra small* ułożą się jeden pod drugim (rysunek 1.31).



Rysunek 1.30. Zakres szerokości dla ekranów small — każdy z bloków zajmuje połowę ekranu



Rysunek 1.31. Zakres szerokości dla ekranów extra small — bloki umiejscowione jeden pod drugim

Framework Bootstrap oprócz opisanych mechanizmów zapewnia wiele klas, które pozwalają stylizować formularze, tworzyć belki nawigacyjne (tzw. navbary) czy zarządzać rozmieszczeniem poszczególnych elementów. W kodzie przedstawionej strony akapitom zostały przypisane klasy `px-2` i `py-1`. Pierwsza ustawia *padding* dla prawej i lewej strony na wartość `2`, druga zaś ustawia *padding* góry i dołu na `1`. Określenie odstępów przebiega zgodnie ze wzorcem:

odstęp-kierunek-wartość — obowiązuje od rozmiaru *extra small*,

lub

odstęp-kierunek-rozmiar-wartość — obowiązuje dla ustawionego rozmiaru,

gdzie:

- `odstęp`: `m` jako *margin* i `p` jako *padding*;
- `kierunek`: `t` — góra, `b` — dół, `l` — strona lewa, `r` — strona prawa, `x` — równocześnie strona lewa i prawa, `y` — równocześnie góra i dół (pominięcie parametru spowoduje zastosowanie wartości z wszystkich stron);
- `wartość`: od `1` do `5` lub `auto` (zwiększanie wartości powoduje wzrost odstępu).

Wygląd elementów strony można określać za pomocą zarówno tradycyjnych deklaracji CSS, jak i klas bootstrapowych. Na przykład wielkość obrazka jest definiowana przez regułę CSS, `width: 100%`, choć można by było użyć klasy Bootstrapa o nazwie `img-fluid`.

1.2.4. Bootstrap — podsumowanie

Podobnie jak w przypadku jQuery, przedstawione w tym rozdziale omówienie możliwości frameworka Bootstrap nie jest wyczerpujące. Pełna dokumentacja tego narzędzia wraz z przykładami użycia znajduje się na stronie projektu, pod adresem *https://getbootstrap.com/*.

Warto również odwiedzić stronę *https://www.w3schools.com/bootstrap/*, na której udostępniono bardzo dużo gotowych przykładów wraz z ich omówieniem.

Zadanie 1.12.

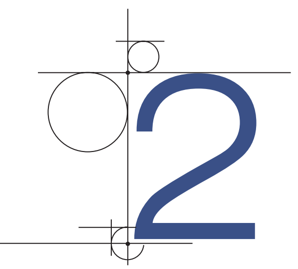
Przygotuj dowolną stronę, która z użyciem frameworka Bootstrap będzie realizowała ideę responsywnej witryny.

Zadanie 1.13.

Odszukaj i sprawdź, co jeszcze ma do zaoferowania Bootstrap. Czy za jego pomocą elementom formularza można nadać styl lub utworzyć menu? Jeśli tak, znajdź przykłady użycia.

Pytania kontrolne

1. W jaki sposób Bootstrap realizuje ideę responsywnych stron?
2. Jaką funkcjonalność zapewnia Bootstrap?
3. Opisz różnice w działaniu klas `container` i `container-fluid`.
4. Co się stanie, jeśli nie użyjesz klasy `row`?



Framework Angular

Angular to obok React, Vue.js i Express jeden z najpopularniejszych frameworków JavaScript, których użycie pozwala stworzyć aplikację webową. Należy on do grupy tzw. frameworków SPA (ang. *Single Page Application*). Główne okno aplikacji ładowane jest tylko raz, a odświeżane są jedynie wymagane fragmenty, i nie jest już konieczne przeładowanie całej strony, gdy następuje zmiana widoku, np. podczas przejścia pomiędzy podstronami.

2.1. Angular — instalacja i konfiguracja środowiska pracy

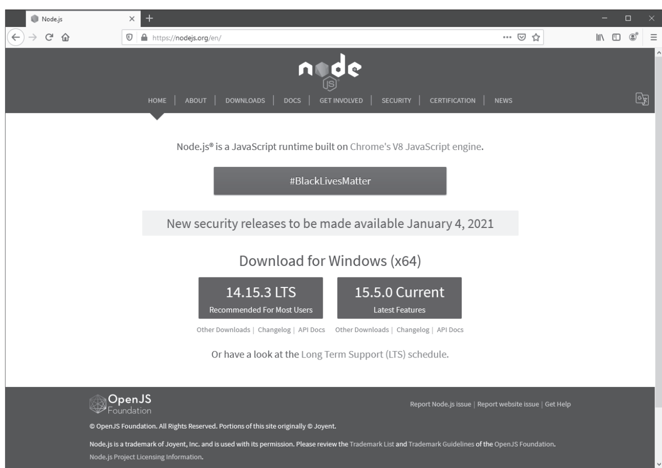
2.1.1. Przygotowanie środowiska

Instalacja Node.js

Konfigurację środowiska pracy rozpoczynamy od instalacji **Node.js**. Node.js jest otwartoźródłowym oprogramowaniem, które pozwala uruchomić kod JavaScript nie lokalnie, w oknie przeglądarki (jak dzieje się zazwyczaj), lecz po stronie serwera. Środowisko to jest szerzej opisane w dalszej części podręcznika.

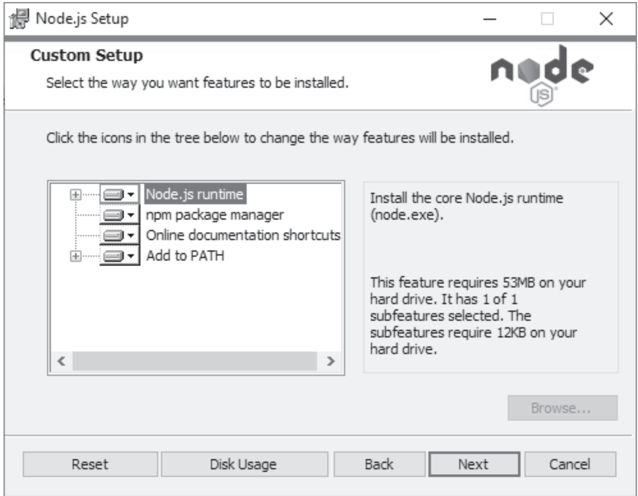
Pakiet instalacyjny można pobrać ze strony *https://nodejs.org* (rysunek 2.1). Udostępnione są zawsze dwie wersje narzędzia: **bieżąca** (*Current*), obsługująca najnowsze funkcje i dodatki, oraz **LTS** (ang. *Long Term Support*), najbardziej dopracowana i najstaranniej przetestowana przez producenta, który zapewnia jej długi okres wsparcia technicznego.

Ponieważ framework Angular jest bardzo często uaktualniany, przed samą instalacją warto się zapoznać z jego wymaganiami, dostępnymi na stronie *https://angular.io/guide/setup-local*.



Rysunek 2.1. Okno strony internetowej, z której należy pobrać instalator Node.js

Instalacja Node.js nie różni się od sposobu, w jaki instaluje się inne programy (rysunek 2.2). Po zaakceptowaniu licencji, określeniu ścieżki docelowej i pozostawieniu opcji domyślnie zaproponowanych przez instalator nastąpi zainstalowanie oprogramowania w systemie. Wraz z Node.js zostanie zainstalowany **menedżer pakietów npm**, który posłuży do zainstalowania i pobrania niezbędnych funkcjonalności wykorzystywanych przez projekt.



Rysunek 2.2. Okno instalacji Node.js — wybór komponentów

Weryfikację instalacji można przeprowadzić za pomocą konsoli systemu Windows. W tym celu należy użyć poleceń `node -v` oraz `npm -v`. Pierwsze sprawdza wersję Node.js, a drugie — wersję menedżera pakietów npm (rysunek 2.3).

```
Administrator: Wiersz polecenia

C:\>node -v
v14.15.3

C:\>npm -v
6.14.9

C:\>
```

Rysunek 2.3. Weryfikacja procesu instalacji Node.js i npm

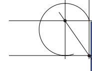
Instalacja Angular CLI

Gdy wszystko przebiegło pomyślnie, możemy przejść do następnego kroku, czyli instalacji **Angular CLI** — narzędzia, które pozwoli utworzyć „szkielet” aplikacji. We wcześniejszych wersjach Angulara niezbędne komponenty i moduły odpowiedzialne za zbudowanie aplikacji należało przygotować samemu, jednak Angular CLI wszystko to automatyzuje, dzięki czemu za pomocą jednego polecenia utworzymy i skonfigurujemy nowy projekt.

Aby zainstalować Angular CLI, należy wydać polecenie `npm install -g @angular/cli` (rysunek 2.4).

```
npm

C:\>npm install -g @angular/cli
npm WARN deprecated debug@4.2.0: Debug versions >=3.2.0 <3.2.7 || >=4 <4.3.1 have a low-severity ReDos r
egression when used in a Node.js environment. It is recommended you upgrade to 3.2.7 or 4.3.1. (https://
github.com/visionmedia/debug/issues/797)
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/
issues/3142
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
[            ......] - extract:lodash: sill extract lodash@^4.17.19 extracted to C:\Users\Administrator
```

Rysunek 2.4. Instalacja narzędzia Angular CLI



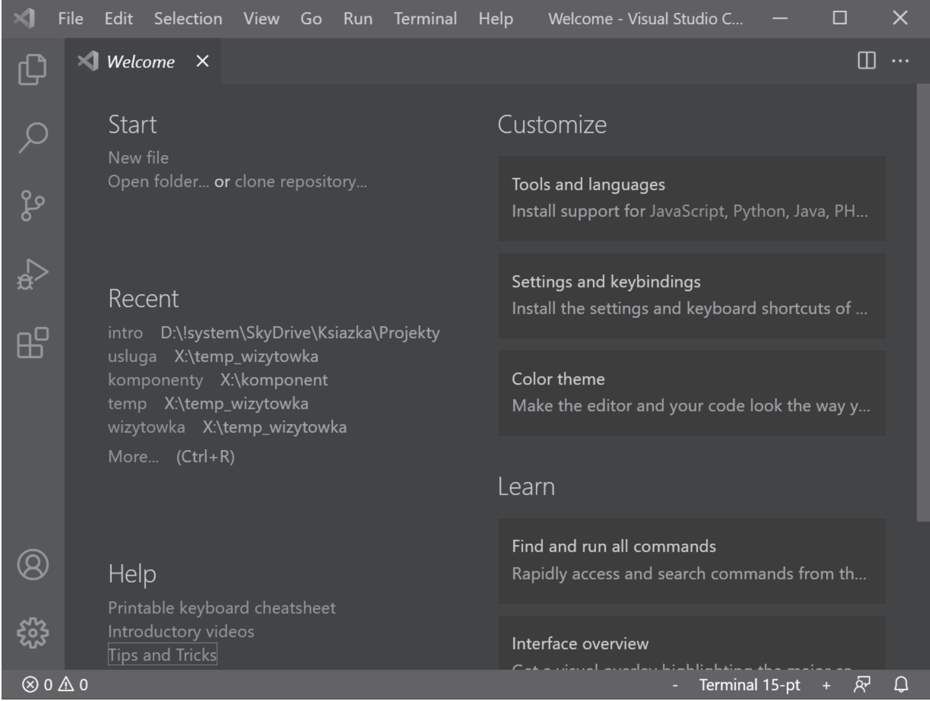
UWAGA

Ponieważ działanie Angulara opiera się na innych modułach, podczas instalacji mogą zostać wyświetlone ostrzeżenia, których źródłem są użyte moduły. Informują one najczęściej o zmianie sposobu działania lub zalecają aktualizację do nowszej wersji.

Użycie w poleceniu flagi `-g` spowoduje, że Angular CLI zostanie **zainstalowany globalnie**. Dzięki dodaniu odpowiedniego wpisu w zmiennej *PATH* systemu polecenia odnoszące się do narzędzia będzie można wydawać niezależnie od bieżącego położenia w strukturze katalogów.

Instalacja Visual Studio Code

Ostatnim krokiem jest instalacja edytora kodu. Jednym z najwyżej ocenianych jest darmowy **Visual Studio Code** (rysunek 2.5). Aplikacja jest niezwykle popularna dzięki dużej społeczności skupionej wokół niej, co zaowocowało powstaniem ogromnej liczby dodatkowych wtyczek i rozszerzeń ułatwiających pracę z kodem. Kolejnym atutem narzędzia jest to, że zostało udostępnione w wersjach przeznaczonych na wszystkie najważniejsze platformy systemowe (Windows, Linux oraz macOS). Dlatego kod aplikacji internetowych będziemy pisać z użyciem właśnie tego edytora. Można go pobrać ze strony *https://code.visualstudio.com/*.



Rysunek 2.5. Okno programu Visual Studio Code

CIEKAWOSTKA

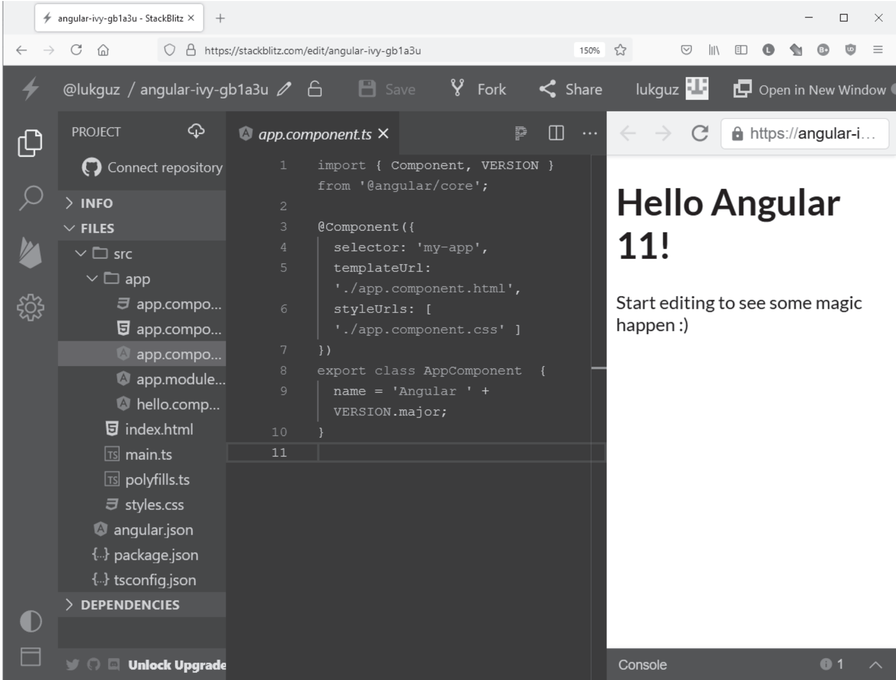
Za wyborem Visual Studio Code przemawia jeszcze jeden atut — producentem narzędzia jest firma Microsoft, która jest także zaangażowana w rozwój języka programowania TypeScript. Tego języka użyjemy wraz z frameworkiem Angular do napisania własnej aplikacji webowej. Oznacza to również, że jeżeli zachodzi jakakolwiek zmiana w języku TypeScript, jest ona bardzo szybko odzwierciedlana w samym edytorze.

Visual Studio Code nie jest oczywiście jedynym dostępnym edytorem kodu. Spośród darmowych do najczęściej wykorzystywanych należą **Atom** (*https://atom.io/*) i **Brackets** (*http://brackets.io/*).

Nasze środowisko pracy zostało przygotowane.

StackBlitz — zdalne środowisko pracy

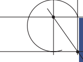
Pokazane rozwiązanie zostało oparte na instalacji lokalnej, ale są też takie, które umożliwiają pracę z kodem z dowolnego miejsca i w każdym systemie operacyjnym. Na pracę zdalną i w szerszej grupie osób pozwala platforma **StackBlitz**, dostępna pod adresem: *https://stackblitz.com/* (rysunek 2.6).



Rysunek 2.6. Platforma StackBlitz

StackBlitz nie tylko pozwala kodować z użyciem frameworka Angular, ale także oferuje przestrzenie robocze (tzw. *workspace*) dla wielu innych języków i frameworków, np. HTML/CSS/JavaScript, React czy Node.js.

Przedstawione rozwiązania (lokalne i online) łączy to, że w obu użyto edytora Visual Studio Code — w pierwszym — jako programu działającego pod kontrolą systemu, w drugim zaś — jako aplikacji wyświetlanej w oknie przeglądarki. Z obu rozwiązań można korzystać równolegle.



WSKAZÓWKA

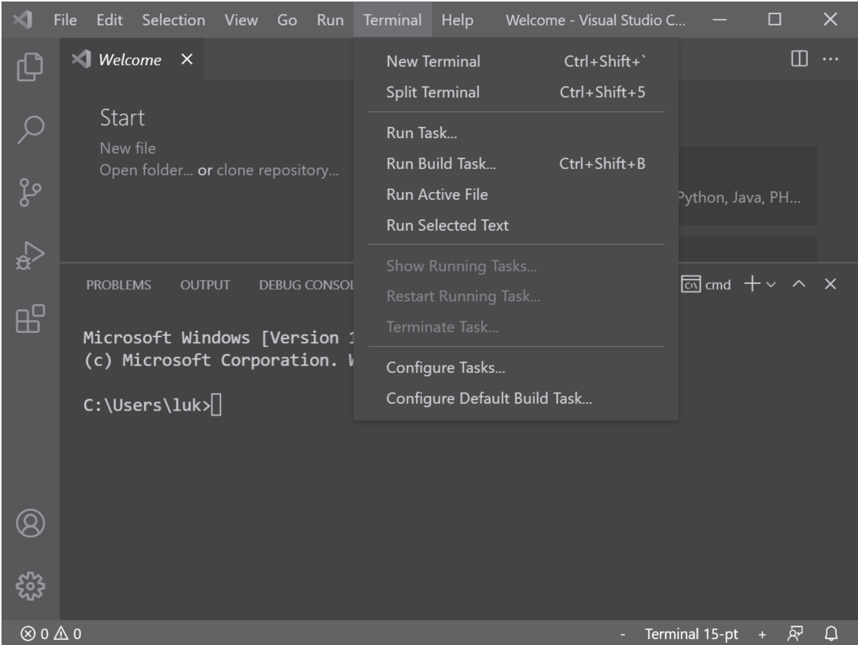
Po zarejestrowaniu się i powiązaniu konta z serwisem GitHub będzie można kodować online, a napisany kod udostępnić innym osobom, np. w sytuacji, gdy trzeba znaleźć rozwiązanie problemu.

2.1.2. Generowanie projektu Angular

Generowanie pierwszego projektu należy rozpocząć od utworzenia katalogu, w którym będą przechowywane pliki. Nazwa katalogu może być dowolna, ale tak jak w przypadku innych języków programowania, wszędzie, gdzie jest to możliwe, **unikamy stosowania polskich znaków oraz spacji** (użyte wyrazy łączymy ze sobą znakiem łącznika bądź podkreślenia).

Na potrzeby podręcznika przyjęto, że wszystkie tworzone projekty zostały zapisane na dysku *D* w katalogu *projekty*.

Ponieważ Angular CLI to narzędzie wiersza poleceń, obsługuje się je za pomocą wiersza poleceń systemu Windows bądź konsoli Windows PowerShell. Najczęściej jednak stosowane jest najwygodniejsze rozwiązanie — terminal zintegrowany z edytorem kodu. Tę funkcjonalność zapewnia również Visual Studio Code. Aby w oknie edytora wyświetlić widok terminala, z górnego menu wybieramy *Terminal*, a następnie *New Terminal* (rysunek 2.7) bądź korzystamy ze skrótu klawiszowego *Ctrl+`* (użycie skrótu *Ctrl+Shift+`* spowoduje dodanie kolejnego okna terminala).



Rysunek 2.7. Visual Studio Code — włączenie okna terminala

W oknie uruchomionego terminala poleceniem `cd` przechodzimy do katalogu, w którym utworzony zostanie projekt (rysunek 2.8).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

Microsoft Windows [Version 10.0.19041.685]
(c) 2020 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\Administrator>d:

D:\>cd projekty

D:\projekty>
```

Rysunek 2.8. Użycie polecenia cd — przejście do katalogu, w którym zostanie utworzony projekt Angular

Aby wygenerować nowy projekt, należy posłużyć się poleceniem:

```
ng new <opcje_dodatkowe> <nazwa_projektu>
```

gdzie:

- `opcje_dodatkowe` — przełączniki pozwalające włączyć/wyłączyć funkcje tworzonego projektu. Do takich przełączników należy np. `--minimal`. Jego dodanie spowoduje, że projekt będzie zawierał tylko pliki niezbędne do utworzenia aplikacji. Pełna lista opcji zostanie wyświetlona po użyciu przełącznika `--help`.
- `nazwa_projektu` — nazwa tworzonego projektu, bez polskich znaków.

Po wydaniu polecenia zostanie uruchomiony kreator, który zada nam kilka pytań.

Pierwsze pytanie brzmi: `Do you want to enforce stricter type checking and stricter bundle budgets in the workspace? This setting helps improve maintainability and catch bugs ahead of time. For more information, see https://angular.io/strict (y/N).`

Odpowiedź twierdząca uruchomi tzw. **tryb ścisły** (ang. *strict mode*). Zostanie włączony mechanizm kontroli kodu, który bardziej restrykcyjnie weryfikuje tworzony kod.

Drugie pytanie to: `Would you like to add Angular routing? (y/N)`. Wpisanie `y` spowoduje uaktywnienie funkcji **routingu**. Na razie ta funkcjonalność nie jest wymagana, dlatego można odpowiedzieć `N`.

Następnie zostanie wyświetlona lista wyboru — należy wskazać preprocesor CSS (język skryptowy, który jest interpretowany lub kompilowany do kaskadowych arkuszy stylów). Wybieramy opcję domyślną — `CSS` (Angular pozwala również formatować treści z użyciem `SCSS`, `Sass`, `Less` lub `Stylus`).

Po odpowiedzeniu na pytania rozpocznie się proces tworzenia projektu. Jego postęp możemy obserwować w oknie terminala. Zainstalowany wraz z Node.js menedżer plików npm pobierze wszystkie niezbędne komponenty (rysunek 2.9).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                node
CREATE witaj/src/environments/environment.prod.ts (51 bytes)
CREATE witaj/src/environments/environment.ts (662 bytes)
CREATE witaj/src/app/app.module.ts (314 bytes)
CREATE witaj/src/app/app.component.html (25725 bytes)
CREATE witaj/src/app/app.component.spec.ts (937 bytes)
CREATE witaj/src/app/app.component.ts (209 bytes)
CREATE witaj/src/app/app.component.css (0 bytes)
CREATE witaj/e2e/protractor.conf.js (904 bytes)
CREATE witaj/e2e/tsconfig.json (274 bytes)
CREATE witaj/e2e/src/app.e2e-spec.ts (656 bytes)
CREATE witaj/e2e/src/app.po.ts (274 bytes)
⠼ Installing packages (npm)...
⊗ 0 ⚠ 0                                   Terminal 15-pt
```

Rysunek 2.9. Tworzenie projektu Angular

Pomyślne utworzenie projektu powinno się zakończyć wyświetleniem komunikatu `Packages installed successfully`.

UWAGA

Ponieważ Angular CLI jest nieustannie rozwijany, wraz z udostępnianiem nowych wersji narzędzia lista pytań może się zmieniać.

Ostatnią czynnością jest sprawdzenie, czy wygenerowany projekt zostanie poprawnie uruchomiony. W tym celu za pomocą edytora otwieramy zawartość katalogu, którego nazwa jest tożsama z nazwą projektu. Aby otworzyć katalog, należy z górnego menu wybrać opcję *File*, a następnie *Open Folder* i w nowo otwartym oknie określić jego położenie. Drugi sposób polega na przeciągnięciu katalogu zawierającego projekt i upuszczeniu go w oknie Visual Studio Code. Lista plików składająca się na projekt zostanie wyświetlona z lewej strony okna edytora.

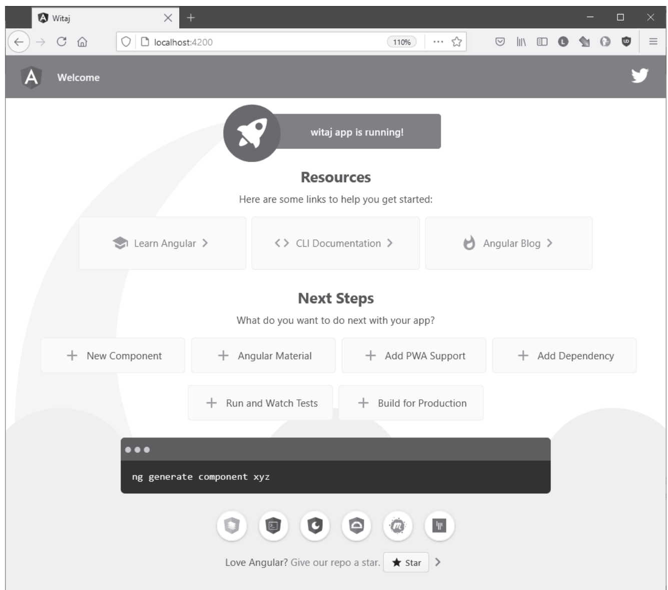
Uruchomienie projektu następuje po otwarciu terminala i wydaniu polecenia `ng serve`.

Po udanym skompilowaniu projektu zostanie uruchomiony serwer hostujący aplikację. Aby sprawdzić jej działanie, w oknie przeglądarki należy wpisać adres `localhost:4200`, gdzie port 4200 to domyślny port, pod którym działa aplikacja Angular (rysunek 2.10). Alternatywną opcją jest zastosowanie w poleceniu `ng serve` flagi `–o`. Jej dodanie spowoduje automatyczne otwarcie okna domyślnej przeglądarki.

WSKAZÓWKA

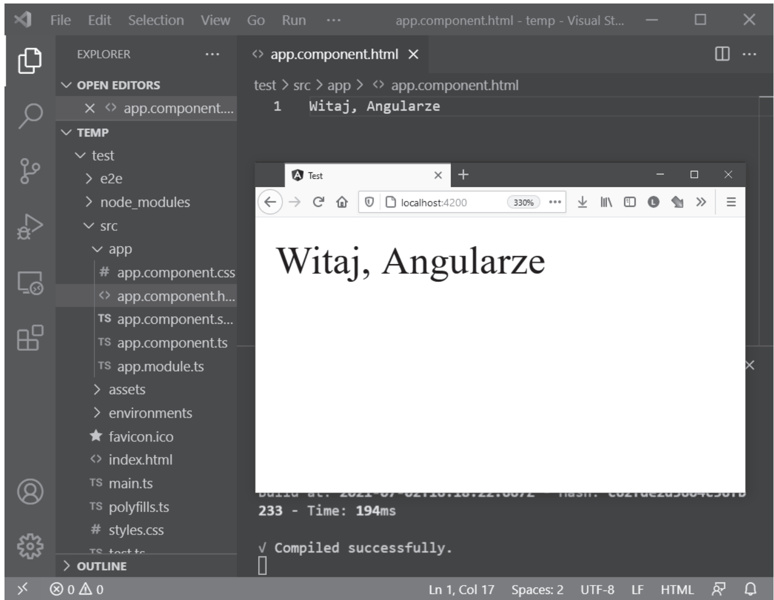
Aby zmienić domyślny port usługi, np. na 5000, należy do polecenia uruchamiającego aplikację dodać: `--port 5000`. Alternatywną opcją jest dodanie w pliku *angular.json* do obiektu `options` znajdującego się w `projects – <nazwa_projektu> – architect – serve` dwóch kluczy z wartościami: `"host": "127.0.0.1"` oraz `"port": 5000` (oddzielonych od siebie przecinkami).

Aplikacja została poprawnie wygenerowana i uruchomiona.



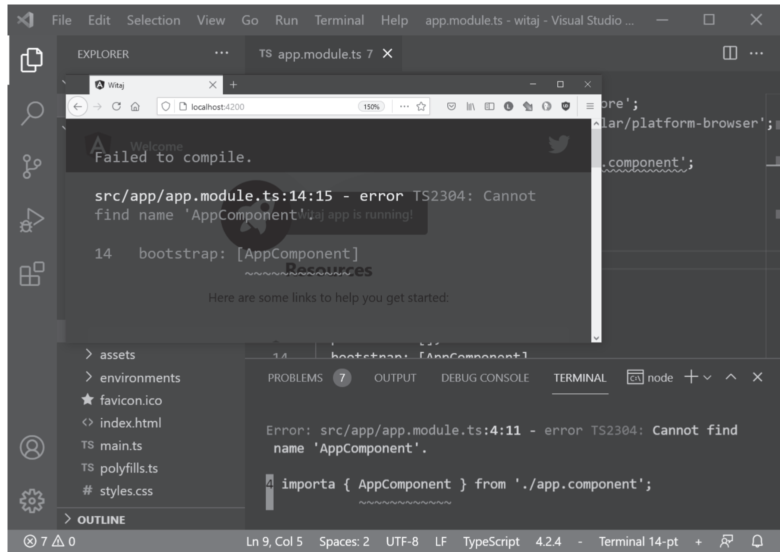
Rysunek 2.10. Domyślny widok okna aplikacji Angular

Uruchomiony serwer działa w **trybie deweloperskim**, co oznacza, że po zapisaniu zmian wprowadzonych w kodzie stan aplikacji jest na bieżąco aktualizowany w oknie przeglądarki. Aby przetestować działanie tego trybu, w drzewie plików odszukujemy katalog *app*, a w nim plik *app.component.html*. Po wyświetleniu zawartości pliku zastępujemy ją np. ciągiem: *Witaj, Angularze*. Zapisanie pliku wymusi zmianę w oknie przeglądarki (rysunek 2.11).



Rysunek 2.11. Sprawdzenie działania trybu deweloperskiego

W trakcie tworzenia kodu aplikacji komunikat o napotkanych błędach kompilacji zostanie wyświetlony w oknie przeglądarki oraz oknie terminala (rysunek 2.12).



Rysunek 2.12. Błędy kompilacji aplikacji Angular — terminal, okno przeglądarki

Pytania kontrolne

1. Jakie narzędzia są niezbędne, aby można było utworzyć projekt Angular?
2. Jakie korzyści daje instalacja globalna? Czy taki sposób instalacji ma wady?
3. Co oznacza, że serwer działa w trybie deweloperskim?
4. Porównaj ze sobą lokalne i zdalne środowiska pracy. Wymień ich wady i zalety.

2.2. TypeScript

TypeScript to język programowania będący nadzbiorem języka JavaScript. To oznacza, że obowiązuje w nim składnia języka JavaScript, ale rozbudowana o dodatkowe elementy. Dzięki temu kod napisany w TypeScripcie jest czytelniejszy, a liczba popełnianych błędów — mniejsza. Wykorzystują go m.in.: Facebook, Microsoft, Google i Netflix. Mieści się on w pierwszej dziesiątce najchętniej wybieranych języków programowania, a jego popularność ciągle wzrasta (według PYPL PopularitY of Programming Language — *https://pypl.github.io/PYPL.html*).

2.2.1. Pierwsze użycie TypeScript. Instalacja niezbędnych narzędzi

Przygotowanie środowiska pracy — TypeScript

TypeScript jest wykorzystywany w połączeniu z Angularem, Node.js czy jako zamiennik czystego JavaScriptu. Kod zapisany w TypeScripcie, aby mógł zostać wykonany przez przeglądarkę, trzeba przekonwertować na JavaScript.

Na samym początku nauki tego języka można by wykorzystać projekt utworzony w poprzednim rozdziale, ale ponieważ użycie TypeScriptu nie ogranicza się tylko do Angulara, często trzeba użyć środowiska, które pozwoli pracować z „czystym" językiem.

Budowanie projektu rozpoczynamy od utworzenia katalogu, w którym będziemy przechowywać jego pliki. W tym celu w lokalizacji *D:\Projekty* został utworzony katalog o nazwie *intro*.

W terminalu po uruchomieniu Visual Studio Code należy przejść do katalogu projektu i wydać polecenie `npm init -y` (po uprzedniej instalacji Node.js). Po zatwierdzeniu polecenia zostanie utworzony plik *package.json*, w którym będą zapisane podstawowe informacje o tworzonym projekcie, m.in. jego nazwa, wersja, opis (rysunek 2.13). Pominięcie flagi `-y` uruchomi kreator, który zapyta nas o te dane.

```
{
  "name": "intro",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

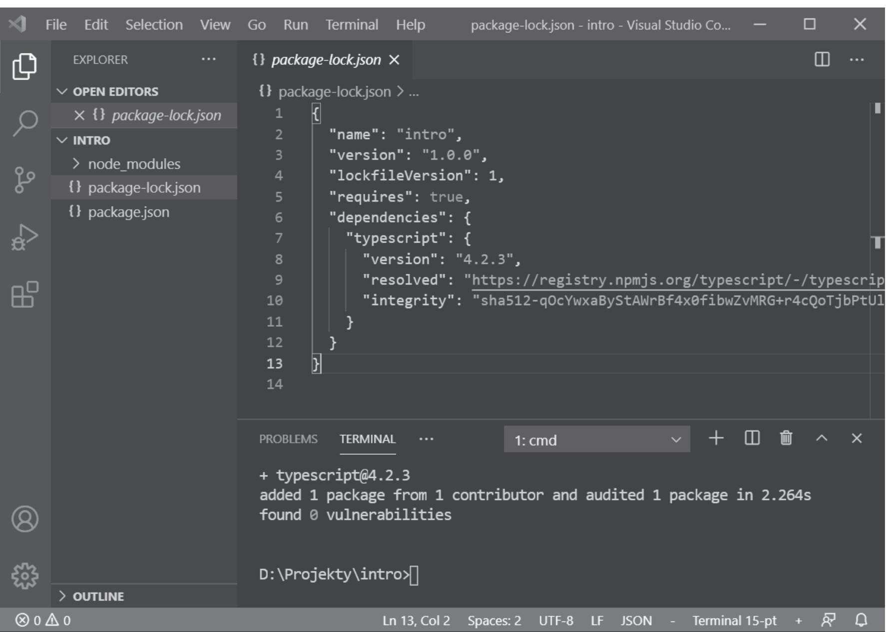
Rysunek 2.13. Podstawowe informacje o projekcie — plik package.json

WSKAZÓWKA

Do przemieszczania się pomiędzy katalogami w systemie operacyjnym służy polecenie powłoki `cd`. Wydanie np. polecenia `cd "c:\Program Files"` spowoduje przejście do katalogu *Program Files* (jeśli w nazwie katalogu występuje znak spacji, ścieżkę obejmujemy znakami cudzysłowu). Zawartość katalogu wyświetlimy za pomocą polecenia `dir`.

Następny krok to instalacja języka TypeScript. Robimy to poleceniem `npm install typescript`.

Zatwierdzenie polecenia spowoduje utworzenie nowego pliku o nazwie *package-lock.json*, w którym zostaje zapisana informacja o użytej wersji języka, oraz katalogu *node_modules*, zawierającego pliki odpowiedzialne za obsługę języka TypeScript (rysunek 2.14).



Rysunek 2.14. Instalacja języka TypeScript — utworzenie pliku package-lock.json oraz katalogu node_modules

UWAGA

Katalog *node_modules* oprócz plików odpowiedzialnych za działanie TypeScriptu będzie zawierał pliki modułów wykorzystywanych przez naszą aplikację, a także pliki modułów użytych do jej stworzenia. Jego wielkość zależy od liczby modułów.

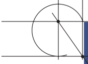
Ostatnią czynnością jest inicjalizacja języka TypeScript. Po wydaniu polecenia `npx tsc -init` zostanie utworzony nowy plik *tsconfig.json* (konfiguruje działanie kompilatora) i będzie można rozpocząć kodowanie (rysunek 2.15).

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this file */

    /* Basic Options */
    // "incremental": true,                   /* Enable incremental compilation */
    "target": "es5",                          /* Specify ECMAScript target version:
    "module": "commonjs",                     /* Specify module code generation: 'n
    // "lib": [],                             /* Specify library files to be includ
    // "allowJs": true,                       /* Allow javascript files to be compi
    // "checkJs": true,                       /* Report errors in .js files. */
    // "jsx": "preserve",                     /* Specify JSX code generation: 'pres
    // "declaration": true,                   /* Generates corresponding '.d.ts' fi
    // "declarationMap": true,                /* Generates a sourcemap for each cor
    // "sourceMap": true,                     /* Generates corresponding '.map' fil
    // "outFile": "./",                       /* Concatenate and emit output to sin
    // "outDir": "./",                        /* Redirect output structure to the d
    // "rootDir": "./",                       /* Specify the root directory of inpu
    // "composite": true,                     /* Enable project compilation */
    // "tsBuildInfoFile": "./",               /* Specify file to store incremental
    // "removeComments": true,                /* Do not emit comments to output. */
```

```
D:\Projekty\intro>npx tsc -init
message TS6071: Successfully created a tsconfig.json file.

D:\Projekty\intro>
```

Rysunek 2.15. Inicjalizacja języka TypeScript — plik tsconfig.json

Użycie polecenia rozpoczynającego się od członu `npx` nakazuje uruchomienie modułu zapisanego w katalogu *node_modules*. Wszystkie pliki niezbędne do jego działania muszą się znajdować w tym katalogu, a on sam jest uruchamiany **lokalnie**, co oznacza, że jest dostępny tylko dla bieżącego projektu. Użycie modułu w innym projekcie wymaga jego ponownego pobrania.



WSKAZÓWKA

Część modułów obsługuje flagę `-g`. Użycie tej flagi podczas instalacji modułu spowoduje, że będzie on dostępny nie tylko dla bieżącego projektu, ale również dla wszystkich pozostałych. Instalacja modułu globalnie jest uznawana za złą praktykę, ponieważ zmiana wersji modułu wpływa na wszystkie projekty.

Możemy rozpocząć pisanie pierwszego programu. Pozostańmy wierni tradycji i wypiszmy tekst (w konsoli przeglądarki) `Hello World` (listing 2.1).

Kod programu napisany z użyciem TypeScriptu wygląda tak:

Listing 2.1. Hello World

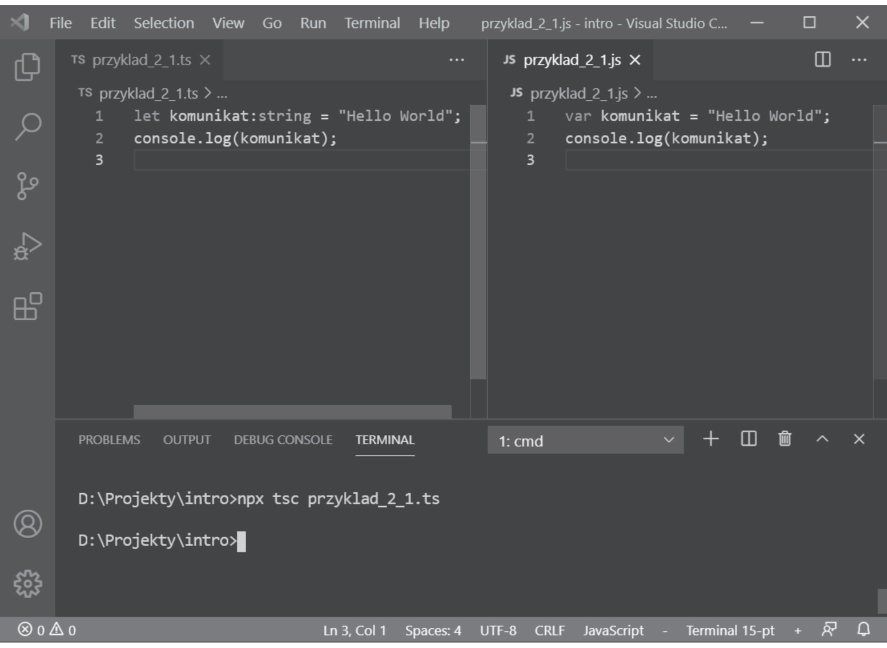
```
let komunikat:string = "Hello World";
console.log(komunikat);
```

Kod skompilujemy za pomocą polecenia **npx tsc <nazwa_pliku.ts>**,

gdzie:

`nazwa_pliku.ts` — plik z kodem TypeScript. Jeśli polecenie zostanie wywołane bez nazwy pliku, zostaną skompilowane wszystkie pliki z rozszerzeniem **.ts* znajdujące się w katalogu.

Wydanie polecenia spowoduje konwersję kodu zapisanego w TypeScripcie (lewy panel) na JavaScript (prawy panel); zostanie utworzony plik z rozszerzeniem **.js*, którego nazwa będzie się pokrywać z nazwą kompilowanego pliku (rysunek 2.16).



Rysunek 2.16. Porównanie napisanego kodu TypeScript z wynikowym kodem JavaScript

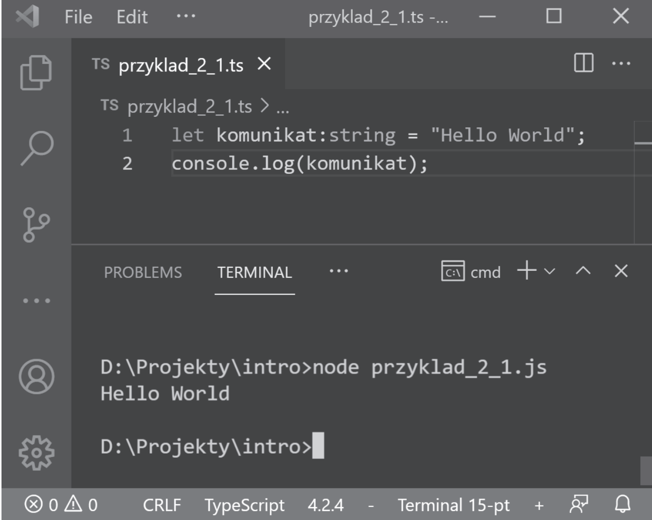
Aby pokazać różnice pomiędzy TypeScriptem a JavaScriptem, za pomocą słowa `let` zadeklarowano zmienną typu `string` (opis typów zmiennych znajduje się w następnym rozdziale) przechowującą łańcuch `Hello World`. Proces kompilacji spowodował zmianę sposobu jej zadeklarowania na `var`.

Druga linijka kodu pozostała bez zmian, ponieważ w TypeScripcie i JavaScripcie sposób wyświetlania w konsoli zawartości zmiennej jest identyczny.

Poprawność otrzymanego kodu sprawdzimy poleceniem **node <nazwa_pliku.js>**,

gdzie:

`nazwa_pliku.js` — wynikowy kod JavaScript (rysunek 2.17).

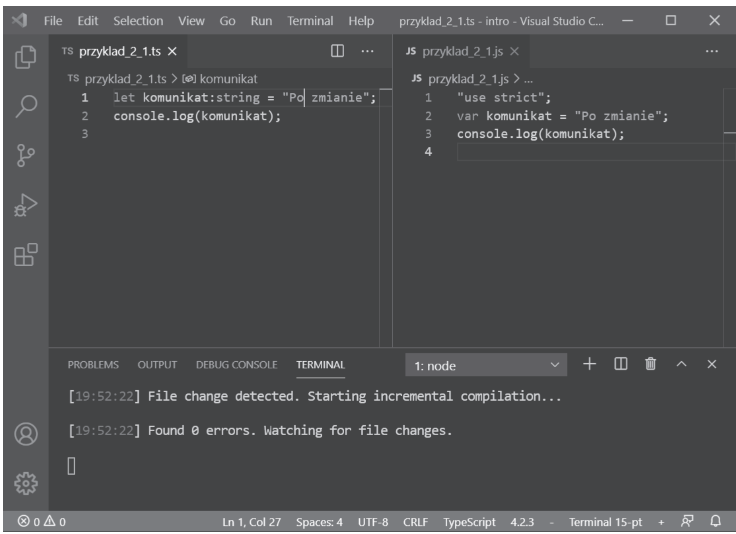


Rysunek 2.17. Użycie polecenia node — sprawdzenie kodu JavaScript

Tryb nasłuchu — automatyczna kompilacja kodu TypeScript

Uruchomienie kompilatora w **trybie nasłuchu** spowoduje automatyczną kompilację[1] pliku TypeScript na JavaScript w momencie wykrycia zmian (po zapisaniu pliku). Tryb nasłuchu włącza się poleceniem `npx tsc --watch`.

Zmiana wartości zmiennej `komunikat` z `Hello World` na `Po zmianie` (lewy panel) i zapisanie pliku uruchamia proces automatycznej kompilacji — kod zawarty w pliku JavaScript zostaje uaktualniony (prawy panel) (rysunek 2.18).



Rysunek 2.18. Tryb nasłuchu

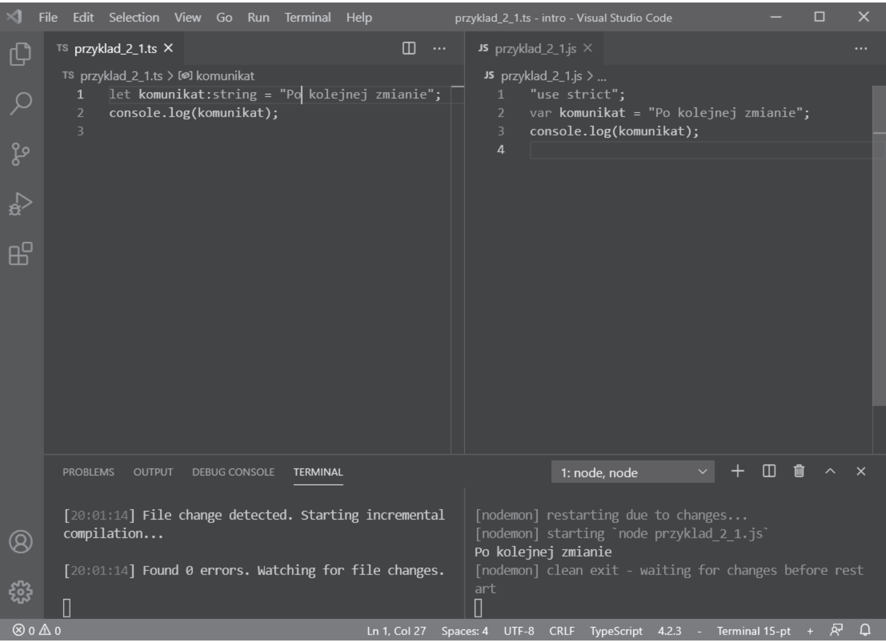
1 W przypadku przetwarzania kodu TypeScript na JavaScript można też użyć terminu *transpilacja*.

Automatyczne uruchomienie wynikowego kodu JavaScript

W sposób automatyczny można nie tylko kompilować pliki, ale również je weryfikować. Wynikowy kod JavaScript po naniesionych zmianach zostanie uruchomiony dzięki modułowi **nodemon**. Moduł ten w domyślnej konfiguracji nie jest instalowany; żeby go pobrać i zainstalować, należy wydać polecenie `npm install nodemon`.

Po zainstalowaniu modułu w zbiorze lokalnym można go uruchomić poleceniem **npx nodemon <nazwa_pliku.js>**. Jakakolwiek zmiana kodu pliku spowoduje wyświetlenie zmian.

Po zmianie łańcucha przypisanego do zmiennej `komunikat` z `Po zmianie` na `Po kolejnej zmianie` i zapisaniu pliku **.ts* (lewy panel) zostaje on automatycznie skompilowany na kod JavaScript (włączony tryb nasłuchu). Zmiana w pliku **.js* wymusza ponowną interpretację kodu (prawy panel) (rysunek 2.19).



Rysunek 2.19. Działanie modułu nodemon

Oba narzędzia zatrzymamy kombinacją klawiszy *Ctrl+C*.

Pytania kontrolne

1. Co oznacza, że dany język programowania jest nadzbiorem innego języka?
2. Jakie narzędzia trzeba mieć, aby móc kodować z użyciem TypeScriptu?
3. Co oznacza, że moduł jest uruchamiany lokalnie?
4. Jak działa tryb nasłuchu?

5. Co powoduje uruchomienie modułu nodemon?
6. Sprawdź, czy kod napisany w TypeScripcie zostanie zinterpretowany przez przeglądarkę.

2.2.2. Typy wbudowane

Typowanie dynamiczne a statyczne

Tym, co różni JavaScript od TypeScriptu, jest mechanizm typowania — w JavaScripcie jest on **dynamiczny**, co oznacza, że wartości przypisanej do zmiennej automatycznie zostaje przydzielony typ. W TypeScripcie jest on **statyczny** — to programista decyduje, jakiego typu mają być wartości. Przeanalizujmy kod na listingu 2.2 i przypomnijmy, jak działa mechanizm typowania dynamicznego w JavaScripcie.

Listing 2.2. Wyświetlenie typu zmiennej w JavaScripcie

```
let moja_zmienna;
console.log(moja_zmienna + " = " + typeof moja_zmienna);
moja_zmienna = "Witaj";
console.log(moja_zmienna + " = " + typeof moja_zmienna);
moja_zmienna = 41;
console.log(moja_zmienna + " = " + typeof moja_zmienna);
moja_zmienna = false;
console.log(moja_zmienna + " = " + typeof moja_zmienna);
```

Pierwsza linijka kodu zawiera definicję zmiennej bez przypisania jej wartości. W trzeciej do zmiennej zostaje przypisany ciąg `Witaj`, w piątej wartość zmienia się na `41`, w siódmej zaś zmienna ostatecznie przyjmuje wartość `false`. Do sprawdzenia typu danych został użyty operator `typeof`.

Interpretacja kodu powinna dać następujący wynik:

```
undefined = undefined
Witaj = string
41 = number
false = boolean
```

Jak widać, JavaScript w sposób w pełni automatyczny określił typy wartości. Słowo `Witaj` jest typu `string`, liczba `41` to `number`, a `false` to `boolean`. Nieprzypisanie wartości do zmiennej dało wynik `undefined`.

Mechanizm ten zapewnia nam elastyczność o tyle, że nie musimy określać typu zmiennej, ale w bardziej rozbudowanych projektach jest źródłem potencjalnych problemów, polegających na złym dopasowaniu typu zmiennej.

Typowanie statyczne w przeciwieństwie do dynamicznego pozwala programiście ustalić typ wartości przypisanej do zmiennej, dzięki czemu kompilator w razie użycia innego typu danych zgłosi błąd.

W TypeScripcie, aby określić typ zmiennej, musimy zapisać go po znaku dwukropka — według schematu:

```
let <nazwa zmiennej>:<typ wartości> = <wartość_zmiennej>
```

gdzie po znaku `:` poprzedzonym słowem kluczowym `let` (służy do definiowania zmiennej; odpowiednik `var` w JavaScripcie) i `<nazwą zmiennej>` określamy `<typ wartości>`.

Użycie kodu przedstawionego na listingu 2.3 spowoduje przypisanie do zmiennej o nazwie `moja_zmienna` wartości `41`, która jest typu `number`. Ustalony typ weryfikuje się przez użycie znanej Ci już instrukcji `typeof`.

Listing 2.3. Ustalenie typu zmiennej w TypeScripcie

```
const moja_zmienna: number = 41;
console.log(moja_zmienna + " = " + typeof moja_zmienna);
```

Wynik działania kodu jest następujący:

```
41 = number
```

UWAGA

Aby zadeklarować stałą, należy użyć słowa kluczowego `const`.

Ustalenie typu w sposób statyczny uniemożliwi przypisanie wartości do zmiennej, która nie jest liczbą.

Próba wykonania kodu przedstawionego na listingu 2.4 spowoduje zgłoszenie błędu — Visual Studio Code wyświetli stosowny komunikat informujący, że nie można przypisać wartości `Witaj` i `false` do zmiennej o nazwie `moja_zmienna` (rysunek 2.20). Stało się tak, ponieważ ta pierwsza wartość jest typu `string`, a druga to `boolean`, my zaś zdefiniowaliśmy zmienną jako `number`.

Listing 2.4. Próba zmiany wartości zmiennej

```
let moja_zmienna: number = 41;
moja_zmienna = "Witaj";
moja_zmienna = false;
```

UWAGA

Podczas pisania kodu Visual Studio Code wyświetla w oknie *Problems* informacje o błędach, ostrzeżenia i komunikaty, a także podkreśla problematyczne fragmenty. Nakierowanie wskaźnika myszy na podkreślenie wyświetli szczegóły błędu.

Rysunek 2.20. Zgłoszenie błędu Visual Studio Code

Dedukcja typów

TypeScript bardzo dobrze radzi sobie z **dedukcją typów**. To oznacza, że sam jest w stanie stwierdzić, z jaką zmienną ma do czynienia. Wykonanie kodu pokazanego na listingu 2.5 nie powiedzie się, ponieważ TypeScript wydedukował, że zmienna `x` jest typu `number`. Przypisanie do zmiennej wartości `Witaj` typu `string` zakończy się zgłoszeniem błędu.

Listing 2.5. Dedukcja typów

```
let x = 8;
x = "Witaj";
```

Sytuacja się powtórzy, gdy do zmiennej `x` w pierwszej kolejności zostanie przypisany wyraz — zmiana na wartość liczbową nie będzie możliwa.

Typ tekstowy — string

Dane tekstowe umieszczamy między cudzysłowami (`"Witaj"`) lub apostrofami (`'Witaj'`).

Tekst może również zostać rozciągnięty na wiele linii i zawierać wartości zmiennych. Używamy do tego znaku ` (dolny znak na klawiszu ze znakiem tyldy).

Zastosowanie wszystkich technik przedstawia kod pokazany na listingu 2.6.

Listing 2.6. Typ tekstowy (string)

```
let imie: string = "Jan";
let nazwisko: string = 'Kowalski';
let jakis_tekst: string = `moje imię to: ${imie},
a nazwisko: ${nazwisko}`;

console.log(`${imie} = ${typeof imie}`);
console.log(`${nazwisko} = ${typeof nazwisko}`);
console.log(`${jakis_tekst} = ${typeof jakis_tekst}`);
```

Ostatni sposób pozwala na użycie wewnątrz napisów **zmiennych**. Nazwę zmiennej umieszczamy w nawiasach klamrowych, poprzedziwszy nawias otwierający znakiem: `$`. Dzięki takiemu zapisowi zostanie podstawiona **wartość użytej zmiennej**. W nawiasach można także stosować wyrażenia (np. operację dodawania) odpowiadające składni języków JavaScript oraz TypeScript. Ta metoda zapisu nazywa się **interpolacją ciągów** (ang. *string interpolation*).

Typ liczbowy — number

Wartości liczbowe umieszczamy po znaku `=` poprzedzonym nazwą zmiennej.

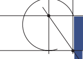
Sposób definiowania zmiennej przechowującej wartość liczbową jest pokazany na listingu 2.7.

Listing 2.7. Typ liczbowy (number)

```
let posesja: number = 17;
let metraz: number = 223.32;
let wartosc: number = 5332100;

console.log(`Wartość domu położonego przy ulicy Mickiewicza ${posesja} o
metrażu ${metraz} m2 wynosi ${wartosc} zł`)
```

Wykonanie kodu da rezultat:

```
Wartość domu położonego przy ulicy Mickiewicza 17 o metrażu 223.32 m2 wynosi
5332100 zł
```



CIEKAWOSTKA

W przypadku dużych liczb, aby szybciej określić wartość, z jaką ma się do czynienia, można użyć znaku podkreślenia (_). Zapis `5_332_100` jest równoznaczny z `5332100`. Wartość zmiennej się nie zmienia, a kod dzięki użyciu podkreślenia staje się czytelniejszy.

TypeScript oprócz liczb w formacie dziesiętnym obsługuje zapisy: binarny, oktalny (ósemkowy) oraz heksadecymalny (szesnastkowy), co ilustruje listing 2.8.

Listing 2.8. Systemy liczbowe

```
let decimal: number = 6;
let hex: number = 0xfff;
let binary: number = 0b11101;
let octal: number = 0o361;

console.log(`${decimal}`);
console.log(`${hex}`);
console.log(`${binary}`);
console.log(`${octal}`);
```

Po wywołaniu kodu wartości zmiennych zostaną wyświetlone w systemie dziesiętnym.

```
6
4095
29
241
```

Typ logiczny — boolean

Kod programu przedstawiony na listingu 2.9 ilustruje sposób działania zmiennych logicznych.

Zmienne `mezczyzna` oraz `kobieta` to zmienne logiczne, które mogą przyjmować wartość `true` albo `false`. To, jaką wartość przyjmą, zależy od warunku zdefiniowanego w instrukcji `if`. Ponieważ instrukcja sprawdza, czy `agnieszka` jest kobietą, status zmiennej `kobieta` zostaje ustawiony na `true`, zmienna `mezczyzna` przyjmuje zaś wartość `false`.

Listing 2.9. Typ logiczny (boolean)

```
let mezczyzna: boolean;
let kobieta: boolean;

const agnieszka: string = "kobieta";

if (agnieszka == "kobieta") {
    kobieta = true;
    mezczyzna = false;
} else {
    kobieta = false;
    mezczyzna = true;
}

console.log(`Zmienna mezczyzna ma wartość ${mezczyzna}, a zmienna kobieta ma
wartość ${kobieta}`)
```

Wynikiem działania kodu jest komunikat:

```
Zmienna mezczyzna ma wartość false, a zmienna kobieta ma wartość true
```

Zadanie 2.1.

Za pomocą instrukcji warunkowej sprawdź, czy podana przez użytkownika liczba całkowita jest poprawnym numerem miesiąca.

Zadanie 2.2.

Za pomocą instrukcji warunkowej sprawdź, czy podana liczba całkowita jest liczbą parzystą, czy nieparzystą.

Zadanie 2.3.

Za pomocą instrukcji warunkowej sprawdź, czy możesz ubiegać się o fotel prezydenta. Prezydentem naszego kraju może zostać każdy, kto ma polskie obywatelstwo i najpóźniej w dniu wyborów kończy 35 lat.

Zadanie 2.4.

Cena biletu zależy od długości trasy według następującego schematu: za przejazdy na odcinku od 0 do 15 km płaci się 3 zł, za przejazdy na odległość od 16 do 40 km stawka wynosi 1,5 zł plus 0,20 zł za każdy kilometr, a za przejazdy dłuższe niż 40 km — 1 zł plus 0,10 zł za każdy kilometr. Przy założeniu, że zmienna `x` (liczba całkowita) oznacza długość trasy w kilometrach, napisz instrukcję wyznaczającą wartość zmiennej `cena` będącej kosztem zakupu biletu za przejazd `x` kilometrów.

Typ any

TypeScript nie zabrania korzystania z dynamicznego systemu typów języka JavaScript. Do przypisywania dowolnego typu zmiennym, stałym czy funkcjom udostępnia typ o nazwie `any`, którego użycie jest pokazane na listingu 2.10.

Listing 2.10. Typ any

```
let cokolwiek: any;

cokolwiek = "szkoła";
console.log(`Zmienna cokolwiek ma wartość ${cokolwiek}
i jest typu ${typeof cokolwiek}`)

cokolwiek = 64;
console.log(`Zmienna cokolwiek ma wartość ${cokolwiek}
i jest typu ${typeof cokolwiek}`)
```

Skompilowanie kodu da wynik:

```
Zmienna cokolwiek ma wartość szkoła i jest typu string
Zmienna cokolwiek ma wartość 64 i jest typu number
```

Działa to również w drugą stronę — zmienną o typie `any` możemy przypisać do dowolnej innej zmiennej, co jest pokazane na listingu 2.11.

Listing 2.11. Typ any — przypisanie do innej zmiennej

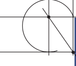
```
let cokolwiek: any = 64;
let wyraz: string = cokolwiek;
console.log(`Zmienna wyraz ma wartość ${wyraz} i jest typu ${typeof wyraz}`)

cokolwiek = "szkoła";
let liczba: number = cokolwiek;
console.log(`Zmienna liczba ma wartość ${liczba} i jest typu ${typeof liczba}`)
```

Skompilowanie kodu da wynik:

```
Zmienna wyraz ma wartość 64 i jest typu number
Zmienna liczba ma wartość szkoła i jest typu string
```

Pomimo że jawnie określiliśmy typ zmiennej `wyraz` jako `string`, przypisanie do niej zmiennej `cokolwiek` o typie `any` z wartością `64` nie wywołuje żadnego błędu. Sytuacja się powtarza, gdy do zadeklarowanej zmiennej `liczba` o typie `number` ponownie przypisujemy zmienną `cokolwiek`, lecz tym razem przechowującą wartość typu `string`.

Typ `any` wprowadzono po to, aby ułatwić przepisywanie kodu w JavaScripcie do TypeScriptu, dlatego tworząc nowy kod, **unikamy tego typu, a stosujemy go wyłącznie w procesie migracji**.



UWAGA

Typu `any` możemy również użyć w połączeniu z funkcją bądź obiektem.

Łączenie typów

TypeScript pozwala także łączyć typy. To oznacza, że dana zmienna może być np. typu `string` lub `number`. Mechanizm ten jest pokazany na listingu 2.12. Zmienna `wspolna` może być typu `number` albo `string`. To, jakiego jest typu, zależy od przypisanej wartości. Typy łączymy ze sobą za pomocą znaku pionowej kreski (`|`).

Listing 2.12. Łączenie typów

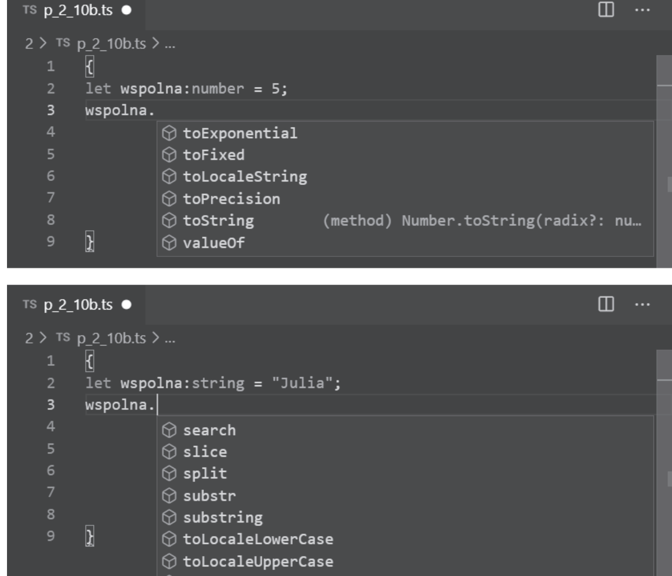
```
let wspolna: string | number;

wspolna = 22;
console.log(`Zmienna wspolna ma wartość ${wspolna}
i jest typu ${typeof wspolna}`)

wspolna = "Ala ma kota";
console.log(`Zmienna wspolna ma wartość ${wspolna}
i jest typu ${typeof wspolna}`)
```

Zmienna `wspolna` użyta w kodzie przedstawionym powyżej używa dwóch typów, ale można łączyć ze sobą dowolną liczbę typów.

UWAGA

W literaturze ten mechanizm jest określany również jako unia typów.

Podczas kodowania Visual Studio Code podpowiada, jakie **metody** są dostępne dla danej zmiennej, i je sugeruje. Dostępny wybór zależy od typu zmiennej, co pokazuje rysunek 2.21.



Rysunek 2.21. Visual Studio Code — metody

Metody dostępne dla zmiennej `wspolna` typu `number` różnią się od tych, których można użyć, gdy zmienna ta jest typu `string`. Połączenie ze sobą typów sprawia, że lista metod kurczy się do tych, które stanowią **część wspólną** obu typów — w omawianym przykładzie są to metody `toLocaleString`, `toString` i `valueOf`. Jest tak, ponieważ edytor nie wie, jaka wartość zostanie przypisana do zmiennej, a w efekcie — jakiego będzie ona typu.

CIEKAWOSTKA

Poznając tajniki języka TypeScript, na pewno wcześniej czy później zechcesz utworzyć zmienną o nazwie, która została już użyta w innym pliku. Ponieważ Visual Studio Code pliki znajdujące się w otwartym katalogu traktuje jako jeden projekt, nie pozwoli nam na to. Jak widać na rysunku 2.22, nie można zadeklarować zmiennej o nazwie `metraz`, gdyż zmienna o takiej nazwie już istnieje w pliku *p_2_5.ts*.

```
TS p_2_10b.ts ●   TS p_2_5.ts
2 > TS p_2_10b.ts > ...
1
2   let metraz
3      let metraz: any
4      Cannot redeclare block-scoped variable 'metraz'. ts(2451)
5      p_2_5.ts(3, 9): 'metraz' was also declared here.
6      Peek Problem (Alt+F8)   No quick fixes available
7
8
9
```

Rysunek 2.22. Próba wykorzystania nazwy zmiennej zadeklarowanej w innym pliku

Dlatego aby poinformować kompilator, że zmienna o użytej nazwie swoim zasięgiem ma obejmować tylko plik, w którym się znajduje, należy objąć cały kod nawiasami klamrowymi (tzw. zasięg). Objęcie kodu nawiasami sprawi, że Visual Studio Code nie zgłosi błędu (rysunek 2.23).

```
TS p_2_10b.ts ●   TS p_2_5.ts
2 > TS p_2_10b.ts > ...
1   {
2
3   let metraz
4
5   }
6
7
8
9
```

Rysunek 2.23. Zasięg

Typy jako zbiory wartości

TypeScript oprócz łączenia typów umożliwia definiowanie **zbiorów wartości**. Zbiór możliwych wartości definiujemy z użyciem pionowej kreski (rysunek 2.24).

Zmienna `ocena` może przyjąć wartości: `1`, `2`, `3`, `4`, `5` oraz `6`. Próba przypisania liczby `7` kończy się wyświetleniem komunikatu o błędzie: `Type '7' is not assignable to type '1 | 2 | 3 | 4 | 5 | 6'`.

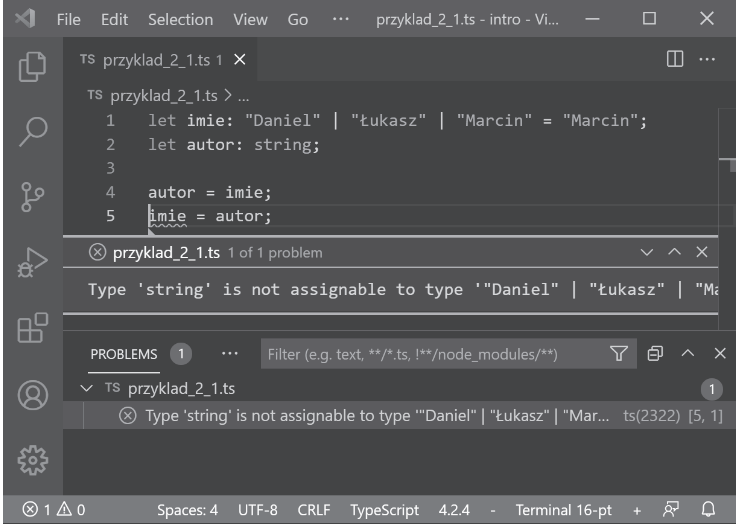
```
TS p_2_11.ts ×
2 > TS p_2_11.ts > ...
1
2    let ocena : 1 | 2 | 3 | 4 | 5 | 6;
3
4    ocena = 5;
5    let ocena: 1 | 2 | 3 | 4 | 5 | 6
6
7    Type '7' is not assignable to type '1 | 2 | 3 | 4 | 5 |
8    6'. ts(2322)
9    Peek Problem (Alt+F8)   No quick fixes available
10   ocena = 7;
11
```

Rysunek 2.24. Typ jako zbiór wartości

Idea zbiorów została również zilustrowana na listingu 2.13. Zmienna `imie` może mieć trzy wartości: `Daniel`, `Łukasz` bądź `Marcin` (została ustawiona wartość domyślna: `Marcin`). Druga zmienna, `autor`, jest typu `string`, można więc do niej przypisać jakąkolwiek wartość będącą łańcuchem znaków. Dlatego uda się przypisać zbiór wartości zmiennej `imie` do zmiennej `autor` (zbiór wartości zmiennej `imie` zawiera się w zbiorze wartości zmiennej `autor`), ale nie na odwrót, gdyż zbiór możliwych wartości zmiennej `autor` jest szerszy niż ten zadeklarowany dla zmiennej `imie`. Visual Studio Code zwróci komunikat o błędzie: `Type 'string' is not assignable to type '"Daniel" | "Łukasz" | "Marcin"'` (rysunek 2.25).

Listing 2.13. Zmienna jako zbiór wartości

```
let imie: "Daniel" | "Łukasz" | "Marcin" = "Marcin";
let autor: string;

autor = imie;
imie = autor;
```



Rysunek 2.25. Zbiory

Pytania kontrolne

1. Czym się różni typowanie statyczne od dynamicznego? Podaj przykłady języków programowania wykorzystujących dany rodzaj typowania.
2. Na czym polega działanie mechanizmu dedukcji typów? Wskaż wady i zalety tego rozwiązania.
3. Z jakich typów podstawowych korzysta TypeScript? Czy te typy są stosowane w innych językach programowania? Podaj przykłady.
4. Dlaczego w TypeScripcie nie używamy typu `any`? A skoro nie używamy, to dlaczego ten typ jest dostępny?
5. Na czym polega łączenie typów?
6. Co nazywamy zbiorem wartości?

2.2.3. Tablice

Sposoby definiowania tablicy

TypeScript, podobnie jak inne języki programowania, zapewnia obsługę **tablic** (ang. *array*). Skoro TypeScript jest nadzbiorem języka JavaScript, powinien zadziałać sposób stosowany w tym drugim. Kod z listingu 2.14 pokazuje przykładową definicję tablicy.

Nazwę tworzonej tablicy poprzedzamy słowem kluczowym `let`, a następnie w nawiasie kwadratowym umieszczamy jej elementy, oddzielone od siebie przecinkami.

Listing 2.14. Tworzenie tablicy

```
let imiona = ["Jan", "Paweł", "Mariusz"];
let nazwiska = ["Kowalski", "Nowak", "Tryla"];

console.log(`Pierwszy element: ${imiona[0]} ${nazwiska[0]}`);
console.log(`Drugi element: ${imiona[1]} ${nazwiska[1]}`);
console.log(`Trzeci element: ${imiona[2]} ${nazwiska[2]}`);
```

Wywołanie kodu spowoduje wypisanie elementów tablicy.

```
Pierwszy element: Jan Kowalski
Drugi element: Paweł Nowak
Trzeci element: Mariusz Tryla
```

UWAGA

Indeksy tablic zaczynają się od zera.

Jeśli trudno Ci zrozumieć ideę tablic, wyobraź sobie komodę z wieloma szufladami ponumerowanymi od 0. Do każdej szuflady trafia jedna rzecz: do szuflady 0 — skarpetki, do szuflady 1 — czapka, do szuflady 2 — rękawiczki itd. Jeśli chcesz założyć np. czapkę, musisz ją wyjąć z odpowiedniej szuflady. Analogicznie należy postąpić z danymi.

Zadziała również sposób ze słowem kluczowym `new Array` (listing 2.15).

Listing 2.15. Tworzenie tablicy za pomocą słowa kluczowego new Array

```
let samochody = new Array("Saab", "Volvo", "BMW");
console.log(`Pierwszy element: ${samochody[0]}`);
```

Wywołanie kodu spowoduje, że zostaną wypisane elementy tablicy.

```
Pierwszy element: Saab
```

TypeScript po przeanalizowaniu kodu `let imiona = [];` przypisze tablicę `imiona` do typu `any`. Sposób określania typu danych przechowywanych w tablicy jest pokazany na listingu 2.16.

Typ określa się (tak samo jak w przypadku zmiennych) za pomocą znaku dwukropka (`:`), po którym następuje nazwa typu zakończona nawiasami kwadratowymi (`[]`).

Listing 2.16. Tablica — określenie typu przechowywanych danych

```
let imiona: string[] = [];

imiona.push("Tomasz");
imiona.push(14);
```

Metoda `push()` dołącza określony element na końcu tablicy, a ponieważ w ostatniej linijce próbujemy do tablicy dodać wartość `14`, która jest typu `number`, powyższy kod nie skompiluje się, a zamiast tego zostanie wyświetlony komunikat o błędzie: `p_2_14.ts:4:13 — error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.`

Do wyświetlenia wszystkich elementów tablicy użyjemy pętli `for of` (listing 2.17). Po słowie `for` definiujemy w nawiasach okrągłych zmienną, do której będą przypisywane kolejne elementy tablicy zdefiniowanej po słowie `of`. W nawiasach klamrowych zawarto instrukcję, która ma być wykonywana w każdym przebiegu pętli.

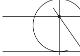
W pierwszym przebiegu pętli do zmiennej `element` zostaje przypisana wartość zapisana pod indeksem `0` (`Jan`) w tablicy `imiona`, a następnie jest ona wyświetlana. W drugim przebiegu wartość zmiennej `element` zostaje zastąpiona wartością zapisaną pod indeksem `1` i ponownie wyświetlona. Pętla kończy działanie po wyświetleniu ostatniego elementu tablicy.

Listing 2.17. Pętla for of — wyświetlenie wszystkich elementów tablicy

```
let imiona: string[] = ["Jan", "Paweł", "Mariusz"];

for (let element of imiona) {
    console.log(element);
}
```

Efektem wywołania kodu jest wyświetlenie wszystkich imion znajdujących się w tablicy `imiona`.

```
Jan
Paweł
Mariusz
```



UWAGA

Zawartość tablicy można również wyświetlić za pomocą metody `forEach()`, ale w połączeniu z użyciem funkcji. Kod takiego programu jest pokazany na listingu 2.37, w podrozdziale poświęconym funkcjom.

Najczęściej wykorzystywane metody tablicowe są przedstawione w tabeli 2.1.

Tabela 2.1. Wybrane metody tablicy

Metoda	Opis
`concat(nazwa_dołączanej_tablicy)`	Po użyciu metody zostanie zwrócona nowa tablica, która powstaje w wyniku połączenia tablicy z tą podaną w argumencie (można łączyć ze sobą kilka tablic)
`join(separator)`	Zwracany jest ciąg tekstowy wszystkich wartości przechowywanych w tablicy; poszczególne elementy są od siebie oddzielone separatorem
`pop()`	Metoda usuwa i zwraca ostatni element w tablicy
`push(element)`	Metoda dołącza określony element na końcu tablicy
`reverse()`	Metoda zwraca tablicę, której elementy są posortowane w kolejności malejącej
`shift()`	Metoda usuwa i zwraca pierwszy element w tablicy
`slice(start, stop)`	Metoda zwraca fragment tablicy; `start` to indeks początku, `stop` zaś to indeks końca
`sort()`	Metoda zwraca tablicę, której elementy są posortowane w kolejności rosnącej
`splice(indeks, liczba)`	Metoda usuwa podaną liczbę elementów od wskazanego indeksu
`unshift(element)`	Metoda powoduje wstawienie nowego elementu na początku tablicy

Użycie trzech wybranych metod przedstawionych w tabeli 2.1 jest pokazane na listingu 2.18.

Listing 2.18. Użycie metod concat(), join() oraz sort()

```
const imiona: string[] = ["Jan", "Paweł", "Mariusz"];
const nazwiska = ["Kowalski", "Nowak", "Tryla"];
const wiek: number[] = [45, 50, 19];
```

```
    const nowaTablica = imiona.concat(nazwiska);
    for (let element of nowaTablica) {
        console.log(element);
    }
    const polaczona = wiek.join(":");
    console.log(`Zmienna polaczona ma wartość ${polaczona}
    i jest typu ${typeof polaczona}`);
    const posortowana = wiek.sort();
    for (let element of posortowana) {
        console.log(element);
    }
```

Kompilacja kodu da następujący wynik:

```
Jan
Paweł
Mariusz
Kowalski
Nowak
Tryla
Zmienna polaczona ma wartość 45:50:19 i jest typu string
19
45
50
```

Użycie metody `concat()` pozwoliło połączyć ze sobą dwie tablice. Tablica `nowaTablica` zawiera elementy tablic `imiona` oraz `nazwiska`.

UWAGA

To, czy pod zmienną kryje się tablica, można też sprawdzić za pomocą metody `Array.isArray()`, która jako argument przyjmuje nazwę sprawdzanej zmiennej. Jeśli metoda zwróci wartość `true`, mamy do czynienia z tablicą, np. `console.log(Array.isArray(nowa_tablica));`.

Metoda `join()` tworzy łańcuch, którego wartością są wszystkie elementy tablicy `wiek` oddzielone od siebie znakiem dwukropka (`:`).

Metoda `sort()` sortuje elementy tablicy w kolejności rosnącej.

Typ tablicy może być wyrażony również za pomocą składni z użyciem nawiasu ostrego (listing 2.19). Dlatego też kod:

```
const imiona: string[] = ["Jan", "Paweł", "Mariusz"];
```

jest odpowiednikiem następującego:

```
const imiona: Array<string> = ["Jan", "Paweł", "Mariusz"];
```

Listing 2.19. Tablica wyrażona za pomocą składni z użyciem nawiasu ostrego

```
let imiona: Array<string> = ["Jan", "Paweł", "Mariusz"];

for (let element of imiona) {
    console.log(element);
}
```

Wykonanie tego kodu da wynik:

```
Jan
Paweł
Mariusz
```

Zadanie 2.5.

Utwórz tablicę zawierającą owoce: `jabłko, arbuz, banan, gruszka`.

- **a.** Zwróć długość tablicy.
- **b.** Zwróć posortowaną tablicę.
- **c.** Dodaj na koniec tablicy nową wartość `ananas`.
- **d.** Usuń pierwszy element tablicy.
- **e.** Usuń ostatni element tablicy.
- **f.** Dodaj jako pierwszy element tablicy nową wartość `ananas`.
- **g.** Odwróć kolejność elementów tablicy.

Zadanie 2.6.

Za pomocą pętli `for` wyświetl zawartość tablicy z zadania 2.5.

Do szybkiego łączenia ze sobą tablic możemy użyć **operatora rozwinięcia** (ang. *spread operator*), który ma postać trzech kropek (...).

Użycie operatora **spread** jest pokazane na listingu 2.20.

Listing 2.20. Operator rozwinięcia

```
let koloryCieple: string[] = ['czerwony', 'pomarańczowy', 'żółty'];
let koloryZimne: string[] = ['zielony', 'niebieski', 'szary'];

let kolory = [...koloryCieple, ...koloryZimne];

console.log(kolory);
```

Tablica `kolory` składa się ze wszystkich elementów tablic `koloryCieple` oraz `koloryZimne`.

```
[
    'czerwony',
    'pomarańczowy',
    'żółty',
    'zielony',
    'niebieski',
    'szary'
]
```

Zadanie 2.7.

Wykorzystaj tablicę z zadania 2.5 i dodatkowo utwórz kolejną, zawierającą warzywa: `marchew`, `burak`, `pietruszka`, `kalafior`.

a. Połącz obie tablice z użyciem metody `.concat()`.
b. Sprawdź, czy tablice można do siebie dodać za pomocą znaku dodawania.
c. Połącz obie tablice z użyciem operatora rozwinięcia.

Krotka

Krotka to tablica o stałej wielkości, której poszczególne elementy mogą być różnych typów. Listing 2.21 ilustruje definicję krotki.

Listing 2.21. Krotka

```
let krotka: [string, number] = ["Jan Kowalski", 40];

console.log(`Nazywam się ${krotka[0]} i mam ${krotka[1]} lat.`);
```

Tablica `krotka` jest zbudowana z dwóch elementów, przy czym pierwszy (o indeksie `0`) jest typu `string`, drugi zaś, zapisany pod indeksem `1`, jest typu `number`.

Próba dodania trzeciego elementu do tak zdefiniowanej krotki zakończy się niepowodzeniem, tak samo jak próba przypisania np. do pierwszego elementu wartości, która nie jest typu `string`. Oba przypadki zostały pokazane na rysunku 2.26.

Wyliczenie

Wyliczenie pozwala nadawać zbiorom wartości bardziej przyjazne nazwy. Wyliczenie definiujemy za pomocą słowa kluczowego `enum`, po którym określamy nazwę wyliczenia. W nawiasie klamrowym znajdują się zaś wartości, oddzielone przecinkami. W literaturze wyliczenie często nazywa się również **enumeracją**. Użycie wyliczenia jest pokazane na listingu 2.22.

```
2 > TS p_2_18.ts > ...
1   {
2
3
4   let krotka : [string, number] = ["Jan Kowalski", 40, "Opole"];
5       let krotka: [string, number]
6       'krotka' is declared but its value is never read. ts(6133)
7
8   }   Type '[string, number, string]' is not assignable to type
        '[string, number]'.
          Source has 3 element(s) but target allows only 2. ts(2322)
        Peek Problem (Alt+F8)   Quick Fix... (Ctrl+.)
```

```
2 > TS p_2_18.ts > [@] krotka
1   {                                 Type 'number' is not assignable to type
2                                     'string'. ts(2322)
3                                     Peek Problem (Alt+F8)   No quick fixes available
4   let krotka : [string, number] = [40];
5
```

Rysunek 2.26. Błędna definicja krotki

Listing 2.22. Definicja wyliczenia

```
enum kolory {
    czerwony,
    zielony,
    niebieski,
    żółty
}

let niebieskiWartosc = kolory.niebieski;
let niebieski = kolory[2];

console.log(`Mój ulubiony kolor to: ${niebieski}, a jego wartość
to ${niebieskiWartosc}.`);
```

W powyższym przykładzie mamy wyliczenie o nazwie `kolory`. Wyliczenie ma cztery wartości: `czerwony`, `zielony`, `niebieski` i `żółty`. Wartości wyliczenia zaczynają się od **zera** i przyrastają co **1**.

Kod przedstawiony na listingu 2.23 generuje nazwę wartości wyliczenia, którą w omawianym przykładzie jest `niebieski`.

```
Mój ulubiony kolor to: niebieski, a jego wartość to 2.
```

Dostęp do nazwy koloru (niebieski) uzyskamy również z wykorzystaniem składni: `kolory[kolory.niebieski]`.

Można samodzielnie zainicjować pierwszą wartość, ale można również określić dowolną. Możemy np. napisać:

Listing 2.23. Definicja wyliczenia

```
enum kolory {
    czerwony = 2,
    zielony,
    niebieski = 5,
    żółty = 10
}

let zolty = kolory[10];
let zielony = kolory[3];

console.log(`Lubię również kolory: ${zolty} oraz ${zielony}.`);
```

Ponieważ pierwszy element wyliczenia rozpoczyna się od wartości `2`, następnemu zostaje przypisana wartość `3`, a dwa kolejne mają wartość wyliczenia określoną przez programistę.

Wynikiem wykonania kodu jest:

```
Lubię również kolory: żółty oraz zielony.
```

Pytania kontrolne

1. W jaki sposób w TypeScripcie definiowana jest tablica?
2. Wymień metody tablicy i wskaż ich zastosowanie. Czy oprócz metod wymienionych w tabeli 2.1 istnieją inne?
3. Czym jest operator rozwinięcia?
4. Czym jest krotka?
5. Wskaż możliwe zastosowania wyliczenia.

2.2.4. Funkcje

Funkcja jest zbiorem instrukcji, które wykonują określone zadanie. Pozwala programiście podzielić kod na mniejsze fragmenty i w razie potrzeby odwoływać się do tego kodu.

Deklaracja funkcji

Deklaracja funkcji składa się ze słowa kluczowego `function` oraz:

- **nazwy funkcji**,
- **listy parametrów** wraz z określeniem ich typu, umieszczonych w nawiasach okrągłych i oddzielonych przecinkami,
- **instrukcji**, które zostaną wykonane po wywołaniu funkcji, umieszczonej w nawiasach klamrowych.

Przyjrzyjmy się najprostszej funkcji przedstawionej na listingu 2.24, której zadaniem jest wypisanie w konsoli przekazanego argumentu.

Listing 2.24. Funkcja wypisująca

```
function powitanie(tresc) {
    console.log(`Witaj, ${tresc}`);
}

powitanie('Cezary');
```

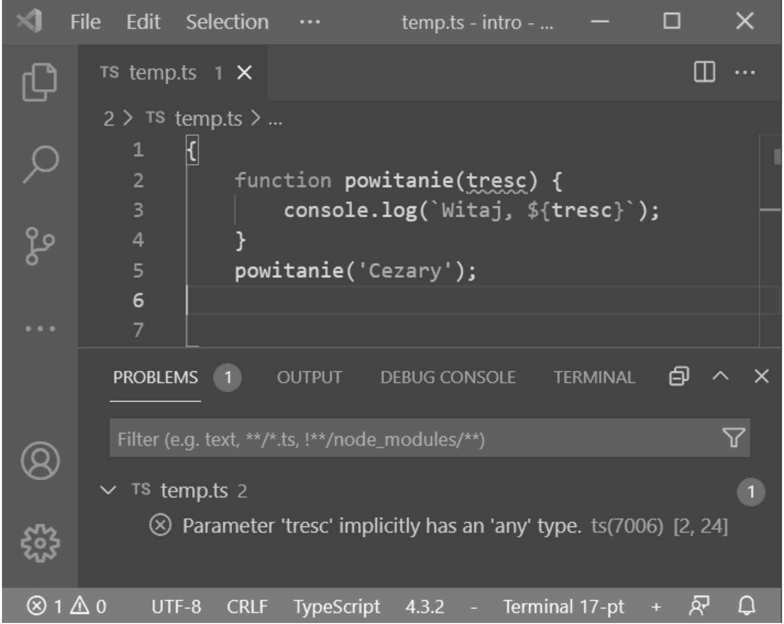
Zadaniem funkcji `powitanie` jest wypisanie podanego argumentu w konsoli. Po wywołaniu funkcji z argumentem `Cezary` otrzymamy komunikat:

```
Witaj Cezary
```

UWAGA

Określenia parametr funkcji oraz argument funkcji są często używane zamiennie, co nie jest poprawne. O parametrach mówimy, gdy funkcję **tworzymy**, o argumentach zaś — gdy ją **wywołujemy**. Na listingu 2.24 funkcja o nazwie `powitanie` ma jeden parametr, `tresc`, a wywołana zostaje w połączeniu z argumentem `Cezary`.

Tak zdefiniowana funkcja zadziała, ale gdy dokładniej przyjrzymy się oknu Visual Studio Code, zobaczymy, że kompilator zgłasza błąd (rysunek 2.27).



Rysunek 2.27. Błąd przy definiowaniu funkcji

Głównym zadaniem funkcji jest **zwrócenie jakiejś wartości**, lecz funkcja `powitanie` tego nie robi — wypisuje jedynie treść komunikatu. Dlatego gdy funkcja nic nie zwraca, należy określić jej typ jako `void`. Typ określamy po dwukropku (tak samo jak w przypadku zmiennej) umieszczonym po zamykającym nawiasie okrągłym. Podkreślenie parametru funkcji to błąd, również związany z typem. Włączony tryb *strict mode* (dbający o jakość kodu) sprawia, że typ `any` nie może zostać przypisany niejawnie, a tak się dzieje w przypadku przekazywanego argumentu `tresc`. Dedukcja typu nie ma tu zastosowania, gdyż TypeScript nie ma danych, które pozwoliłyby mu samodzielnie go określić. Typ parametru definiujemy po dwukropku umieszczonym po jego nazwie.

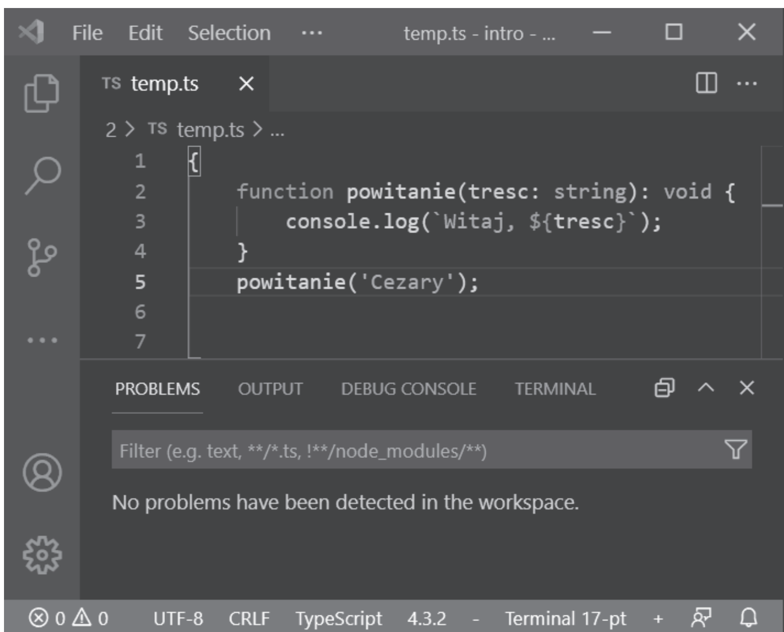
CIEKAWOSTKA

Niejawne przypisywanie typu `any` można wyłączyć. W tym celu należy w pliku *tsconfig.json* usunąć znak komentarza i zmienić wartość parametru `noImplicitAny` z `true` na `false`.

Poprawnie napisany kod jest pokazany na listingu 2.25 oraz rysunku 2.28.

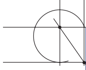
Listing 2.25. Poprawny kod funkcji powitanie

```
function powitanie(tresc: string): void {
    console.log(`Witaj, ${tresc}`);
}

powitanie('Cezary');
```



Rysunek 2.28. Poprawny kod funkcji powitanie

Dodatkowo określenie typu parametru uniemożliwi przekazanie wartości, która nie pasuje do klucza. W naszym przykładzie próba przekazania np. wartości `5`, która jest typu `number`, zakończy się niepowodzeniem.



UWAGA

TypeScript nie pozwala na zdefiniowanie funkcji o już **istniejącej nazwie**, a także nie obsługuje tzw. **przeciążania funkcji**. Mechanizm dostępny w innych językach programowania umożliwia tworzenie wielu funkcji o tej samej nazwie, lecz różnej liczbie parametrów. Funkcja jest rozpoznawana po liczbie przekazywanych argumentów, jak również ich typie.

Pokazana na listingu 2.26 funkcja ma dwa parametry typu `number`, po których otrzymaniu wykonuje operację dodawania. Słowo kluczowe `return` nakazuje zwrócenie wartości tej operacji. Wartość zwracana jest typu `number`.

Listing 2.26. Funkcja suma

```
function suma(a: number, b: number): number {
    return a + b;
}

console.log(`Suma liczb 2 i 7 wynosi ${suma(2, 7)}`)
```

Wynikiem działania kodu jest:

```
Suma liczb 2 i 7 wynosi 9.
```

Zadanie 2.8.

Napisz funkcję, która obliczy i wyświetli w konsoli różnicę, iloczyn i iloraz dwóch liczb podanych przez użytkownika.

Zadanie 2.9.

Napisz funkcję, która wyliczy i wyświetli wartość bezwzględną z zadanej liczby.

Zadanie 2.10.

Napisz funkcję, która po podaniu od użytkownika wartości wzrostu (cm) oraz wagi (kg) obliczy wartość wskaźnika BMI (oblicza się go przez podzielenie masy ciała podanej w kilogramach przez kwadrat wzrostu w metrach). Funkcja w konsoli ma wyświetlić wynik wraz z komentarzem:

- BMI < 18,5: `za mało!`
- BMI > 25: `za dużo!`
- BMI pomiędzy 18,5 a 25: `OK!`

Zadanie 2.11.

Sprawdź, w jaki sposób działa metoda `.filter()`. Masz daną tablicę: `const numery = [2, 4, 7, 11, 14, 19, 21, 100]`. Napisz funkcję, która wypisze w konsoli dwie tablice: pierwszą — z liczbami parzystymi, a drugą — z liczbami nieparzystymi.

Zadanie 2.12.

Napisz funkcję, do której jako argument przekażesz tablicę zawierającą listę osób. Zadaniem funkcji jest wylosowanie osoby. Tablica do wywołania funkcji to: `const osoby = ["Marcin", "Jan", "Beata", "Tadeusz", "Teresa"]`.

TypeScript pozwala również deklarować funkcje za pomocą **wyrażenia**, co zostało przedstawione na listingu 2.27.

Listing 2.27. Funkcja jako wyrażenie

```
const suma = function (a: number, b: number): number {
    return a + b;
}

console.log(`Suma liczb 2 i 7 wynosi ${suma(2, 7)}`)
```

Funkcja `suma` nie zadziała, gdy zamiast dwóch argumentów będziemy próbowali przekazać jeden. Wywołanie funkcji np. za pomocą kodu `suma(4);` spowoduje wyświetlenie komunikatu o błędzie: `Expected 2 arguments, but got 1.`

Próba przekazania większej liczby argumentów, niż zadeklarowano, też zakończy się niepowodzeniem. W omawianym scenariuszu wywołanie funkcji za pomocą kodu `suma(4,5,6)` spowoduje zgłoszenie błędu: `Expected 2 arguments, but got 3.`

Od tych reguł są jednak pewne odstępstwa (listing 2.28).

Listing 2.28. Wartość domyślna parametru funkcji

```
function suma(a: number, b: number = 10): number {
    return a + b;
}

console.log(`Suma liczb wynosi ${suma(3)}`)
```

Wywołanie funkcji przez przekazanie jednego argumentu (wartości `3`) zakończy się sukcesem, ponieważ drugi ma przypisaną wartość **domyślną** `10`. Dlatego wynikiem wykonania funkcji będzie:

```
Suma liczb wynosi 13
```

UWAGA

W momencie wywołania funkcji obowiązuje kolejność przypisywania podanych argumentów funkcji do zadeklarowanych parametrów. To oznacza, że do pierwszego parametru funkcji zostaje przypisana wartość pierwszego argumentu itd.

Listing 2.29 przedstawia kod, w którym drugi z parametrów został zdefiniowany jako **opcjonalny**. Używamy do tego znaku zapytania umieszczonego po nazwie zmiennej.

Listing 2.29. Parametr opcjonalny

```
function powitanie(imie: string, id?: number): string {
    return `Witaj, ${imie}, ${id}`;
console.log(powitanie("Łukasz"));
console.log(powitanie("Łukasz", 44));
}
```

Dlatego wywołanie funkcji z jednym, jak i dwoma przekazanymi argumentami zakończy się sukcesem (rysunek 2.29).

Rysunek 2.29. Użycie parametru opcjonalnego

TypeScript umożliwia również tworzenie funkcji, które mogą przyjąć **dowolną liczbę parametrów** (tzn. nieznaną), co jest zilustrowane na listingu 2.30. Funkcja tego typu jest określana jako **funkcja wariadyczna** (ang. *variadic function*).

Listing 2.30. Funkcja o nieznanej liczbie parametrów

```
function powitanie(...imiona: string[]): string {
    return `Witajcie, ${imiona}`;
}

console.log(powitanie("Cezary", "Tadeusz", "Marcin"));
```

Parametrem tej funkcji jest **tablica** (określana także jako **parametr resztowy**), której nazwa musi być poprzedzona trzema kropkami. Wywołanie funkcji da następujący rezultat:

```
Witajcie, Cezary,Tadeusz,Marcin
```

Ponieważ argumentem funkcji jest tablica, dostępne są metody związane z użyciem tablicy, co przekłada się na wynik działania funkcji. W powyższym przykładzie imiona są oddzielone od siebie przecinkami, ale brakuje spacji. Aby uzyskać pożądany efekt, możemy się posłużyć znaną Ci już metodą `join()`. Wystarczy zmodyfikować drugą linijkę kodu: `` return `Witajcie, ${imiona.join(", ")}`; ``.

Modyfikacja da następujący efekt:

```
Witajcie, Cezary, Tadeusz, Marcin
```

Jeśli funkcja ma kilka parametrów, tablicę **umieszczamy na końcu**, jak zostało pokazane na listingu 2.31.

Listing 2.31. Użycie parametru resztowego

```
function uzytkownik(imie: string, ...dane: string[]): string {
    return `Witaj, ${imie}, pozostałe dane o tobie: ${dane.join(", ")}  `;
}

console.log(user("Cezary", "Opole", "cezary@wp.pl"));
```

Wywołanie funkcji `uzytkownik` z trzema argumentami sprawia, że pierwszy przynależy do parametru `imie`, a dwa pozostałe (opcjonalne) są elementami tablicy.

```
Witaj, Cezary, pozostałe dane o tobie: Opole, cezary@wp.pl
```

Próba wykonania kodu z listingu 2.32 zakończy się niepowodzeniem — kompilator zgłosi błąd: `A rest parameter must be last in a parameter list`. Dzieje się tak, ponieważ parametr resztowy został zdefiniowany jako pierwszy.

Listing 2.32. Błędne użycie parametru resztowego

```
function uzytkownik(...dane: string[], imie: string): string {
    return `Witaj, ${imie}, pozostałe dane o tobie: ${dane.join(", ")}  `;
}

console.log(uzytkownik("Opole", "cezary@wp.pl", "Cezary"));
```

Funkcja anonimowa

W programowaniu **funkcja anonimowa** to funkcja, która nie ma przypisanej nazwy. Funkcje anonimowe są często **argumentami przekazywanymi innym funkcjom**. Najczęściej są używane do operacji, które nie powtarzają się w wielu miejscach, przez co nie chcemy definiować osobnej funkcji.

Przyjrzyjmy się programowi, którego kod jest pokazany na listingu 2.33.

Listing 2.33. Funkcja jako argument

```
function powitanie(imie: string): string {
    return `Witaj, ${imie}`;
}

const bazaImion: string[] = ["Jan", "Paweł", "Mariusz"];
console.log(bazaImion.map(powitanie));
```

Funkcja `powitanie` ma jeden parametr, `imie`, który jest zwracany po jej wywołaniu wraz z napisem `Witaj`. Ta funkcja jest argumentem metody `map()` wykonanej na tablicy `bazaImion`. Takie wywołanie spowoduje wykonanie funkcji na każdym elemencie tablicy (bądź przekazanie każdego elementu tablicy `bazaImion` jako argumentu funkcji `powitanie`).

Wynikiem wywołania kodu jest wyświetlenie:

```
[ 'Witaj, Jan', 'Witaj, Paweł', 'Witaj, Mariusz' ]
```

Ten sam efekt uzyskamy za pomocą kodu przedstawionego na listingu 2.34.

Listing 2.34. Funkcja anonimowa

```
const bazaImion: string[] = ["Jan", "Paweł", "Mariusz"];
    console.log(bazaImion.map(function (imie: string): string { return `Witaj,
${imie}`; }));
```

Funkcja jest definiowana bezpośrednio w metodzie `map()`. Funkcja ta nie ma nazwy, dlatego jest określana jako anonimowa.

Funkcja strzałkowa

Kod z listingu 2.34 może być skrócony do przedstawionego na listingu 2.35.

Listing 2.35. Funkcja strzałkowa

```
const bazaImion: string[] = ["Jan", "Paweł", "Mariusz"];
console.log(bazaImion.map((imie: string): string => `Witaj, ${imie}`));
```

Kod został zapisany z użyciem tzw. **funkcji strzałkowej**, w której zamiast słowa `function` stosowany jest operator `=>`, tzw. **gruba strzałka** (ang. *fat arrow*). Ta funkcja ma tylko jedną instrukcję, dlatego pominięto nawiasy klamrowe i słowo kluczowe `return`.

Ten zapis można jeszcze skrócić. Skoro korzystamy z tablicy, której elementy są typu `string`, niepotrzebna jest definicja typu parametru (zarówno przekazywanego jako argument, jak i zwracanego). Ostatecznie zapis przyjmie postać: ``console.log(bazaImion.map(imie => `Witaj, ${imie}`));``. Ponieważ do funkcji przekazywany jest tylko jeden parametr, zostały również pominięte nawiasy okrągłe (jeśli przekazywane są dwa parametry lub więcej, tego nawiasu **nie można opuścić**).

Użycie funkcji strzałkowej jest pokazane na listingu 2.36, który zawiera skrócony kod z listingu 2.26.

Listing 2.36. Funkcja strzałkowa

```
const suma = (a: number, b: number) => a + b;
console.log(`Suma liczb 2 i 7 wynosi ${suma(2, 7)}`);
```

Następny przykład (listing 2.37) przedstawia użycie metody `forEach()` jako opcji alternatywnej do pętli `for of` użytej w kodzie z listingu 2.17.

Listing 2.37. Użycie metody forEach() — wyświetlenie wszystkich elementów tablicy (tradycyjnie i za pomocą funkcji strzałkowej) z użyciem uprzednio zdefiniowanej funkcji (funkcja nazwana iterate jest argumentem metody forEach()):

```
const imiona: string[] = ["Jan", "Paweł", "Mariusz"];
function iterate(value) {
    console.log(value);
}
imiona.forEach(iterate);

// za pomocą funkcji strzałkowej:
const imiona: string[] = ["Jan", "Paweł", "Mariusz"];
imiona.forEach(value => console.log(value));
```

Wynikiem wywołania obu funkcji będzie:

```
Jan
Paweł
Mariusz
```

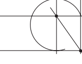
Kolejny przykład użycia funkcji strzałkowej jest pokazany na listingu 2.38.

Listing 2.38. Użycie funkcji strzałkowej — potęgowanie

```
// metoda tradycyjna:
const liczby: number[] = [2, 4, 6];
function potega(a: number) { return a ** 2 };
console.log(liczby.map(potega));

// za pomocą funkcji strzałkowej:
const liczby: number[] = [2, 4, 6];
console.log(liczby.map(value => value ** 2));
```

Wynikiem wywołania każdej z tych funkcji jest wyświetlenie potęg liczb znajdujących się w tablicy.

```
[ 4, 16, 36 ]
```



UWAGA

Wiesz już, jak zadeklarować funkcję `fat arrow`, dlatego od tego momentu w niektórych przykładach będę się posługiwał tym zapisem. Mam też dla Ciebie wyzwanie! Kod z omawianego przykładu spróbuj przekształcić na tradycyjny z użyciem słowa kluczowego `function`, a ten tradycyjny przemianuj tak, aby wykorzystywał strzałkę. Powodzenia!

Zadanie 2.13.

Ponownie wykonaj zadania od 2.8 do 2.12, ale tym razem użyj funkcji strzałkowej.

Zasięg zmiennych

Zmienna zadeklarowana wewnątrz funkcji może zostać użyta jedynie w tej funkcji. To oznacza, **że próba użycia jej poza funkcją zakończy się niepowodzeniem**. Taka zmienna jest nazywana **zmienną lokalną**.

Próbę użycia zmiennych poza ciałem funkcji pokazuje rysunek 2.30. Rezultatem jest komunikat o błędzie: `Cannot find name (…)`. Zmienne `szer` oraz `dl` mogą być użyte jedynie w kodzie funkcji.

```
function poleprostokata(dl:number, szer:number) {
    var pole = dl * szer;
    return pole;
}
dl = 5;
szer = 7;
console.log(poleprostokata(2, 4));
```

PROBLEMS 2
przyklad_2_1.ts
Cannot find name 'dl'. ts(2304) [5, 1]
Cannot find name 'szer'. ts(2304) [6, 1]

Rysunek 2.30. Użycie zmiennych poza funkcją

Zmienna utworzona poza ciałem funkcji ma **charakter globalny** (stąd jej nazwa — **zmienna globalna**). To oznacza, że może zostać użyta w **dowolnym miejscu skryptu**.

Zastosowanie zmiennych globalnych jest pokazane na rysunku 2.31. Zmienne `dl` i `szer` zostały zadeklarowane przed funkcją `wyswietl`. Ponieważ są zmiennymi globalnymi, można ich użyć **wewnątrz funkcji**.

```
let dl = 5;
let szer = 7;

function wyswietl():void {
    console.log(dl*szer)
}

wyswietl();
```

```
D:\Projekty\intro>node przyklad_2_1.js
35

D:\Projekty\intro>
```

Rysunek 2.31. Użycie zmiennych globalnych

Zwrot wielu wartości z funkcji

Zdarzają się sytuacje, w których chcemy, aby funkcja zwróciła nie jedną, ale **dwie, trzy i więcej wartości**. Użycie **tablicy** pozwoli wykonać to zadanie.

Przykład funkcji zwracającej dwie wartości jest przedstawiony na listingu 2.39. Funkcja oblicza dwie wartości: pole siatki prostopadłościanu oraz jego objętość. Wyniki obliczeń zostają przypisane do zmiennych `pole` oraz `objetosc`, a te zostają umieszczone w tablicy `wyniki` i zwrócone. Ponieważ `pole` jest pierwszym elementem tablicy, obliczona wartość zostaje zapisana pod indeksem `0` (pamiętajmy, że indeksowanie tablicy rozpoczynamy od zera, nie od jedynki), a wyliczona objętość — pod indeksem `1`.

Listing 2.39. *Zwrot wielu wartości z funkcji*

```
const prostopadloscian = (wys: number, szer: number, gl: number) => {
    const pole = wys * szer * 2 + wys * gl * 2 + szer * gl * 2;
    const objetosc = wys * szer * gl;
    const wyniki = [pole, objetosc];
    return wyniki;
}
console.log(`Pole siatki prostopadłościanu o wymiarach 5 x 3 x 2 cm wynosi
${prostopadloscian(5, 3, 2)[0]} cm kwadratowych, a jego objętość
to ${prostopadloscian(5, 3, 2)[1]} cm sześciennych`);
```

Wynikiem działania kodu jest:

```
Pole siatki prostopadłościanu o wymiarach 5 x 3 x 2 cm wynosi 62 cm
kwadratowych, a jego objętość to 30 cm sześciennych
```

Pytania kontrolne

1. Wskaż zalety stosowania funkcji.
2. W jaki sposób definiujemy funkcje?
3. Jak zdefiniować wartość domyślną parametru?
4. Czym jest parametr opcjonalny?
5. Jak przekazać do funkcji dowolną liczbę parametrów?
6. Czy funkcja zawsze musi mieć nazwę?
7. Jaką rolę odgrywa „gruba strzałka”?
8. Czy zmienna zadeklarowana w jednej funkcji może zostać użyta przez inną?
9. Ile wartości może zwrócić funkcja?

2.2.5. Praca z obiektami

Obiektem nazywamy zbiór opisujących go zmiennych i funkcji. Niech naszym obiektem będzie pies. Taki obiekt ma cztery łapy, może mieć czarną sierść, wysokość 60 cm i wagę 32 kg. To jego **zmienne**. Potrafi też jeść, szczekać i szybko biegać, czyli wykonuje **funkcje**.

Czym jest obiekt?

W obiekcie zmienne i funkcje przybierają nowe nazwy — zmienna, która jest częścią obiektu, jest jego **właściwością**, funkcja zaś staje się **metodą**. W naszym przykładzie właściwościami obiektu pies są: liczba łap, kolor umaszczenia, wysokość oraz waga, a metodami: jedz, szczekaj i biegaj.

Utworzenie obiektu — notacja literału

Kod przedstawiony na listingu 2.40 opisuje obiekt `hotel`.

Listing 2.40. Obiekt hotel

```
const hotel = {
    nazwa: "Pod topolą",
    standard_pokoi: ["basic", "premium", "vip"],
    liczba_pokoi: 30,
    rezerwacja: 10,
    dostepnosc: function() { return this.liczba_pokoi - this.rezerwacja },
}
console.log(`Hotel ${hotel.nazwa} ma ${hotel.liczba_
pokoi} pokoi w standardzie: ${hotel.standard_pokoi[0]}, ${hotel.standard_
pokoi[1]}, ${hotel.standard_pokoi[2]}, a liczba wolnych pokoi wynosi ${hotel.
dostepnosc()}`);
```

Obiekt zawiera cztery właściwości (ang. *property*) oraz jedną metodę. Tak samo jak zmienne czy funkcje, muszą one mieć nazwę oraz wartość. Nazwa w obiekcie jest określana mianem **klucza** i musi być unikatowa. Do klucza musi zostać przypisana wartość, którą może być ciąg tekstowy, liczba, wartość logiczna, tablica, a także inny obiekt.

Pokazany na listingu 2.40 sposób utworzenia obiektu jest nazywany **notacją literału**. To najłatwiejsza i dlatego najczęściej używana metoda tworzenia obiektów.

Obiekt jest przechowywany w zmiennej `hotel`. Wszystkie pary klucz-wartość są umieszczone pomiędzy nawiasami klamrowymi i oddzielone od siebie przecinkami (z wyjątkiem ostatniego elementu). Klucz od wartości jest oddzielony dwukropkiem.

Aby uzyskać dostęp do właściwości (ale także metody), np. w celu wyświetlenia jej wartości, należy posłużyć się znakiem kropki. Po nazwie obiektu umieszczamy kropkę, a następnie podajemy nazwę klucza.

Alternatywnym sposobem jest użycie **składni z użyciem nawiasu kwadratowego**. Sposób ten został zaprezentowany na listingu 2.41 (definicja obiektu `hotel` wygląda tak samo jak na listingu 2.40).

Listing 2.41. Dostęp do obiektu z zastosowaniem składni z użyciem nawiasu kwadratowego

```
console.log(`Hotel ${hotel['nazwa']}, posiada ${hotel['liczba_pokoi']}
pokoi o standardach: ${hotel['standard_pokoi'][0]}, ${hotel['standard_
pokoi'][1]}, ${hotel['standard_pokoi'][2]}, a liczba wolnych pokoi wynosi
${hotel['dostepnosc']()}`);
```

Którejkolwiek metody użyjemy, rezultat będzie ten sam.

```
Hotel Pod topolą ma 30 pokoi w standardzie: basic, premium, vip, a liczba
wolnych pokoi wynosi 20
```

Słowo kluczowe this

Na listingu 2.40 zostało użyte słowo kluczowe `this`. Jest powszechnie stosowane w funkcjach oraz obiektach. Pozwoliło odwołać się do właściwości obiektu.

Przeanalizujmy prosty przykład. Kod na listingu 2.42 przedstawia obiekt `prostokat`, który zawiera metodę `pole`, obliczającą pole prostokąta. W tym przykładzie słowo kluczowe `this` zostało wywołane w obiekcie `prostokat`, tak więc stanowi odwołanie do niego — użycie zapisu `return this.dl*this.szer` odpowiada `return prostokat.dl*prostokat.szer`.

Listing 2.42. Słowo kluczowe this

```
let prostokat = {
    dl: 12,
    szer: 5,
    pole: function() { return this.dl * this.szer; }
}
console.log(`Pole prostokąta wynosi ${prostokat.pole()} cm kwadratowych`)
```

Na listingu 2.40 mamy sytuację analogiczną do przedstawionej powyżej, z tą różnicą, że zwracana jest liczba wolnych pokoi.

Dzięki użyciu słowa `this` możemy odwoływać się do danego obiektu z **wnętrza jego metod**, a także zastosować pojedynczą funkcję **w kilku obiektach**, jak pokazuje kod z listingu 2.43.

Listing 2.43. Pojedyncza funkcja w kilku obiektach

```
function pokaz(): void {
    console.log(`Procesor: ${this.proc},\nilość pamięci RAM: ${this.RAM}`);
}

const zestaw1 = {
    proc: "AMD 5800X",
    RAM: "32 GB",
    wyswietl: pokaz
}
const zestaw2 = {
    proc: "Intel Core i9–10900K",
    RAM: "16 GB",
    wyswietl: pokaz
}
zestaw1.wyswietl();
zestaw2.wyswietl();
```

Wynikiem działania kodu jest wyświetlenie elementów zestawu komputerowego.

```
Procesor: AMD 5800X,
ilość pamięci RAM: 32 GB
Procesor: Intel Core i9–10900K,
ilość pamięci RAM: 16 GB
```

Zadanie 2.14.

Utwórz dwa obiekty o właściwościach: `nazwa`, `waga`, `cena`. Wypisz w konsoli następujące informacje:

- **a.** nazwę obiektu numer 1,
- **b.** nazwę obiektu numer 2,
- **c.** sumę obu obiektów,
- **d.** droższy obiekt.

Zadanie 2.15.

Napisz funkcję `witaj`, która będzie przyjmować jeden argument — obiekt zawierający dane osoby. Jeżeli przekazany obiekt będzie miał właściwość `imie`, niech funkcja w konsoli wypisze: `Witaj, imie`. Jeśli nie, funkcja ma wypisać tylko: `Witaj`. Przykładowy obiekt do wywołania funkcji to `const osoba = { imie: "Jan", wiek: 25, plec: "m" }`.

Obiekty w tablicy

Powróćmy do przykładu z listingu 2.40. Do przechowania właściwości obiektu została użyta tablica. Właściwość `standard_pokoi` przechowuje trzy wartości: `basic`, `premium` albo `vip`. Dostęp do tych wartości jest realizowany za pomocą kodu rozpoczynającego się od **nazwy obiektu**, następnie po kropce jest umieszczona **nazwa właściwości**, a w nawiasie kwadratowym — **indeks elementu**. Aby np. pobrać wartość `premium`, przechowywaną w tablicy pod indeksem `1`, należy użyć polecenia `hotel.standard_pokoi[1]`.

Nic nie stoi na przeszkodzie, aby to odwrócić i umieścić obiekty w tablicy, co obrazuje listing 2.44.

Listing 2.44. Obiekty w tablicy

```
const osoby = [
    { imie: "Łukasz", wiek: 20 },
    { imie: "Beata", wiek: 33 },
    { imie: "Monika", wiek: 15 }
];
```

Aby wyświetlić daną wartość, rozpoczynamy od nazwy tablicy, po której w nawiasie kwadratowym zostaje umieszczony indeks obiektu. Polecenie kończy kropka połączona z nazwą właściwości przechowującej wartość, czyli aby poznać wiek Beaty, należy użyć polecenia `console.log(osoby[1].wiek)`.

Pętla po obiekcie

Do wyświetlenia wszystkich kluczy i wartości obiektu możemy posłużyć się pętlą `for in` (listing 2.45). Pierwsza pętla wyświetla wartości kluczy obiektu, druga zaś — parę klucz-wartość.

Listing 2.45. Pętla for in

```
const hotel = {
    nazwa: "Pod topolą",
    standard_pokoi: ["basic", "premium", "vip"],
    liczba_pokoi: 30,
    rezerwacja: 10,
    dostepnosc: function () { return this.liczba_pokoi - this.rezerwacja },
}
console.log("Wartości kluczy:")

for (let klucz in hotel) {
    console.log(klucz);
}
console.log("\n----------------------\n")
console.log("Para klucz–wartość:")
for (let klucz in hotel) {
```

```
    console.log("Klucz: ", klucz);
    console.log("Wartość: ", hotel[klucz]);
}
```

Po uruchomieniu kodu zostanie wyświetlona następująca zawartość:

```
Wartości kluczy:
nazwa
standard_pokoi
liczba_pokoi
rezerwacja
dostepnosc

-----------------------

Para klucz—wartość:
Klucz:  nazwa
Wartość:  Pod topolą
Klucz:  standard_pokoi
Wartość:  [ 'basic', 'premium', 'vip' ]
Klucz:  liczba_pokoi
Wartość:  30
Klucz:  rezerwacja
Wartość:  10
Klucz:  dostepnosc
Wartość:  [Function: dostepnosc]
```

Listę kluczy obiektu uzyskamy również za pomocą funkcji `Object.keys(nazwa_obiektu)`.

Typowanie obiektu

TypeScript świetnie radzi sobie z dedukcją typów właściwości i metod obiektu. Aby samemu określić typ, należy po nazwie obiektu umieścić znak dwukropka, a po nim w nawiasie klamrowym podać dla każdego elementu obiektu jego typ (listing 2.46).

Listing 2.46. Typowanie obiektu

```
let hotel: { nazwa: string, standard_pokoi: string[], liczba_pokoi: number,
rezerwacja: number, dostepnosc: () => number; }
    = {
    nazwa: "Pod topolą",
    standard_pokoi: ["basic", "premium", "vip"],
    liczba_pokoi: 30,
    rezerwacja: 10,
    dostepnosc: function () { return this.liczba_pokoi - this.rezerwacja },
}

console.log('Hotel ${hotel.nazwa} ma ${hotel.liczba_pokoi} pokoi w standardzie:
${hotel.standard_pokoi[0]},${hotel.standard_pokoi[1]},${hotel.standard_
pokoi[2]}, a liczba wolnych pokoi wynosi ${hotel.dostepnosc()}');
```

Pytania kontrolne

1. Czym jest obiekt? Podaj przykłady obiektów.
2. W jaki sposób tworzymy obiekt?
3. Czy można obyć się bez `this`?
4. Co rozumiesz przez właściwość obiektu?
5. Do czego używamy metod?

2.2.6. Klasy i interfejsy

Klasa

Klasa to wzór (szablon), na podstawie którego tworzone są obiekty, czyli opisuje zestaw właściwości i metod użytych do budowy obiektu. Utworzony na podstawie klasy obiekt jest nazywany jej **instancją**.

Tworzenie klasy rozpoczynamy od słowa kluczowego `class`, po którym następuje nazwa (przyjęło się, że nazwę klasy rozpoczynamy od dużej litery, choć nie jest to obowiązkowe). W nawiasie klamrowym określamy właściwości i metody (w przykładzie poniżej ich nie ma), które będą przypisane do obiektu.

Na podstawie tak zdefiniowanej klasy możemy utworzyć obiekt — tworzymy zmienną, którą łączymy z nazwą klasy z użyciem słowa kluczowego `new`.

Wartości przypisujemy do właściwości z zastosowaniem notacji z kropką. Utworzenie obiektu za pomocą klasy pokazuje listing 2.47.

Listing 2.47. Zdefiniowanie klasy i utworzenie na jej podstawie obiektu

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;
    kontakt: string[];
}

const Nowak = new Klient();
Nowak.imie = "Tadeusz";
Nowak.nazwisko = "Nowak";
Nowak.wiek = 45;
Nowak.kontakt = ["654342221", "tadnow@gmail.com"];

console.log(`imię i nazwisko klienta: ${Nowak.imie} ${Nowak.nazwisko}
Dane kontaktowe: tel. ${Nowak.kontakt[0]}, email:${Nowak.kontakt[1]}`)
```

Zadanie 2.16.

Utwórz klasę `Samochody`. Klasa powinna mieć właściwości:

```
model: string
marka: string
przebieg: number
wiek: number
```

oraz metodę `szczegoly()`, która wypisze w konsoli powyższe właściwości. Na podstawie tak utworzonej klasy utwórz obiekt.

Przedstawiony powyżej kod jest rozwiązaniem, które wymaga napisania dużej ilości kodu. Znacznie krótszym i łatwiejszym sposobem jest użycie **notacji z konstruktorem**.

Konstruktor to specjalna funkcja umieszczona wewnątrz klasy, wykorzystywana w momencie tworzenia nowego obiektu. Definiujemy ją za pomocą słowa kluczowego `constructor`, po którym w nawiasach okrągłych umieszczamy nazwy właściwości wraz z określeniem typu. Dzięki użyciu słowa kluczowego `this` przekazane argumenty w momencie wywołania klasy stają się jej właściwościami.

Metodę definiuje się podobnie jak funkcję, z tą różnicą, że nie trzeba podawać słowa kluczowego `function`.

Do utworzenia obiektu ponownie wykorzystujemy słowo kluczowe `new` w połączeniu z nazwą klasy. W nawiasach okrągłych należy umieścić wartości wszystkich użytych właściwości (listing 2.48).

Listing 2.48. Definiowanie klasy z wykorzystaniem konstruktora

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;
    kontakt: string[];
    wyswietl() {
        console.log(`Imię i nazwisko klienta: ${this.imie} ${this.nazwisko}
    Dane kontaktowe: tel. ${this.kontakt[0]}, email:${this.kontakt[1]}`)
    }

    constructor(imie: string, nazwisko: string, wiek: number, kontakt:
string[]) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
        this.kontakt = kontakt;
    }
}
const Nowak = new Klient("Tadeusz", "Nowak", 45, ["654342221", "tadnow@gmail.com"]);
Nowak.wyswietl();
```

Wynikiem działania kodu jest wyświetlenie wszystkich właściwości obiektu `Nowak`, który został utworzony z użyciem klasy `Klient`. Za wyświetlenie wartości właściwości odpowiada metoda `wyswietl`.

Kod można jeszcze skrócić dzięki dodaniu w konstruktorze przed parametrem słowa kluczowego `public` (listing 2.49). Parametr ten zostanie automatycznie przypisany do instancji klasy w momencie jej tworzenia. Dodanie słowa `public` pozwala ominąć przypisanie `this`.

Listing 2.49. Użycie słowa kluczowego public

```
class Klient {
    wyswietl() {
        console.log(`Imię i nazwisko klienta: ${this.imie} ${this.nazwisko}
    Dane kontaktowe: tel. ${this.kontakt[0]}, email:${this.kontakt[1]}`)
    }

    constructor(public imie: string, public nazwisko: string, public wiek:
number, public kontakt: string[]) { }
}
let Nowak = new Klient("Tadeusz", "Nowak", 45, ["654342221", "tadnow@gmail.
com"]);
Nowak.wyswietl();
```

UWAGA

Nie tylko modyfikator `public` wykazuje takie działanie, ale również `private` oraz `protected` (przykłady z użyciem modyfikatorów przedstawiono w kolejnym rozdziale).

Zadanie 2.17.

Do klasy z zadania 2.16 zdefiniuj konstruktor. Na jego podstawie utwórz obiekt i wyświetl w konsoli jego właściwości.

Dziedziczenie klas

Klasy mogą po sobie **dziedziczyć** dzięki słowu kluczowemu `extends`.

Omówienie mechanizmu dziedziczenia podzielmy na dwa etapy, a potrzebne klasy utwórzmy w sposób tradycyjny, czyli z wykorzystaniem konstruktora (bez użycia modyfikatora `public`). To pomoże Ci zrozumieć działanie modyfikatorów `private` oraz `protected`.

Listing 2.50 przedstawia klasę `Klient`, po której będzie dziedziczyć inna klasa. Klasa `Klient` ma trzy właściwości oraz dwie metody. Metoda `czyPelnoletni` sprawdza, czy obiekt utworzony na podstawie tej klasy spełnia warunek: wiek co najmniej 18.

Listing 2.50. Klasa Klient

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;

    constructor(imie: string, nazwisko: string, wiek: number) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }

    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}
let Nowak = new Klient("Tadeusz", "Nowak", 44);

console.log(Nowak.czyPelnoletni());
console.log(new Klient("Jan", "Kowalski", 17).czyPelnoletni());
```

Wywołanie kodu zakończy się wypisaniem:

```
true
false
```

`Tadeusz Nowak` ma powyżej 18 lat, natomiast wiek `Jana Kowalskiego` tego warunku nie spełnia, dlatego metoda zwraca `false`. Warto wiedzieć, że TypeScript nie wymaga przypisania z użyciem słowa kluczowego `let`, ale pozwala na bezpośrednie wywołanie metody po utworzeniu obiektu, jak to się dzieje w przypadku `Jana Kowalskiego`.

Na listingu 2.51 do kodu zostaje dodana linijka: `class Dane_klienta extends Klient {}`, która powoduje, że nowo dodana klasa `Dane_klienta` dziedziczy wszystkie właściwości i metody po klasie `Klient`.

Ponownie obiekty `Nowak` i `Kowalski` utworzone na podstawie klasy `Dane_klienta` mogą korzystać z właściwości i metod klasy dziedziczonej. Nic nie stoi na przeszkodzie, aby do obiektu `Nowak` przypisać metodę `czyPelnoletni`, a do obiektu `Kowalski` — metodę `powitanie`, pomimo że metody te nie zostały zdefiniowane w klasie `Dane_klienta`.

Listing 2.51. Użycie extends do utworzenia klasy potomnej

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;

    constructor(imie: string, nazwisko: string, wiek: number) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }

    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}

class Dane_klienta extends Klient { }

let Nowak = new Dane_klienta("Tadeusz", "Nowak", 44);
let Kowalski = new Dane_klienta("Jan", "Kowalski", 17);

console.log(Nowak.czyPelnoletni());
Kowalski.powitanie();
```

Wynikiem wykonania kodu jest wyświetlenie następującej treści:

```
true

Witaj, Jan Kowalski
```

Bardzo często będzie dochodzić do sytuacji, w której pewne właściwości czy metody będziemy chcieli dziedziczyć, ale zarazem dodać własne, przynależne tylko nowo powstałej klasie.

Klasa `Dane_klienta` dziedziczy trzy właściwości (`imie`, `nazwisko`, `wiek`) oraz dwie metody (`powitanie`, `czyPelnoletni`) po klasie `Klient` i równocześnie dodaje nową właściwość: `adres`.

Wykorzystany konstruktor musi uwzględniać właściwości klasy rodzica, jak i właściwości klasy dziedziczącej, dlatego znalazły się w nim wszystkie właściwości obecne w obu klasach. Aby przekazać właściwości z klasy nadrzędnej, należy użyć słowa kluczowego `super` i w nawiasie okrągłym umieścić właściwości (listing 2.52). Wywołanie to musi zostać umieszczone **w pierwszym wierszu konstruktora**.

Listing 2.52. Użycie extends i super do utworzenia klasy potomnej

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;

    constructor(imie: string, nazwisko: string, wiek: number) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }

    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}

class DaneKlienta extends Klient {
    adres: string;

    constructor(imie: string, nazwisko: string, wiek: number, adres: string) {
        super(imie, nazwisko, wiek);
        this.adres = adres;
    }
}

let Nowak = new DaneKlienta("Tadeusz", "Nowak", 44, "Opole, Portowa 17");
let Kowalski = new DaneKlienta("Jan", "Kowalski", 17, "Szczecin, Mickiewicza
19");

for (let wartosc in Nowak) {
    console.log(Nowak[wartosc]);
};
Kowalski.powitanie();
```

Wykonanie kodu spowoduje wyświetlenie wszystkich wartości przypisanych do właściwości obiektu `Nowak` wraz z metodami oraz wykonanie metody `powitanie` na obiekcie `Kowalski`.

```
Tadeusz
Nowak
44
Opole, Portowa 17
[Function: Dane_klienta]
[Function (anonymous)]
[Function (anonymous)]
Witaj, Jan Kowalski
```

Zadanie 2.18.

Wykorzystaj klasę `Samochody` z zadania 2.17. Utwórz nową klasę `kolorSamochodu`, która będzie dziedziczyć właściwości i metody z klasy `Samochody` oraz będzie mieć dodatkową właściwość `kolor: string`. Na podstawie nowej klasy utwórz obiekt i wyświetl w konsoli jego właściwości.

> **WSKAZÓWKA**
>
> Wraz ze zmianą specyfikacji języka JavaScript (ECMAScript 8/ ES2017) dostęp do pól obiektów jest możliwy z użyciem metod: `Object.values`, `Object.keys` oraz `Object.entries`, wiąże się to jednak ze zmianą domyślnych ustawień edytora.

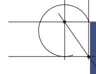
Klasa dziedzicząca ma możliwość nadpisania odziedziczonych metod. To oznacza, że klasa `DaneKlienta` może nadpisać np. metodę `powitanie`, która została jej przekazana z klasy `Klient`. Mechanizm ten jest zobrazowany na listingu 2.53.

Nowa wersja metody `powitanie` została zdefiniowana w klasie dziedziczącej, czyli `Dane_klienta`.

Listing 2.53. Nadpisanie metody

```
class Klient {
    imie: string;
    nazwisko: string;
    wiek: number;

    constructor(imie: string, nazwisko: string, wiek: number) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }

    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}

class DaneKlienta extends Klient {
    adres: string;

    constructor(imie: string, nazwisko: string, wiek: number, adres: string) {
        super(imie, nazwisko, wiek);
        this.adres = adres;
    }
    powitanie() { console.log(`Dzień dobry, ${this.imie} ${this.nazwisko}`); }
}

let Kowalski = new DaneKlienta("Jan", "Kowalski", 17, "Szczecin, Mickiewicza 19");
Kowalski.powitanie();
```

Wywołanie metody `powitanie` spowoduje wyświetlenie jej nowej wersji. Zamiast napisu `Witaj, Jan Kowalski` zostanie wyświetlony `Dzień dobry, Jan Kowalski`.



CIEKAWOSTKA

Można sprawić, aby wartości pól (nie metody) nie zostały nadpisane w procesie dziedziczenia — takie pole trzeba oznaczyć za pomocą znaku `#`. Pola oznaczone w ten sposób są unikatowe w obrębie swojej klasy.

Wartości właściwości można modyfikować, co nie zawsze jest pożądane. Aby temu zapobiec, można użyć **modyfikatorów dostępu**. Umieszczenie w klasie `Klient` przed polami `imie` i `nazwisko` modyfikatora `private` spowoduje zablokowanie dostępu do danego pola z zewnątrz, również dla klasy dziedziczącej (świadczy o tym podkreślenie właściwości `imie` i `nazwisko` w metodzie `powitanie` wewnątrz klasy `DaneKlienta`). Można z niego korzystać tylko wewnątrz klasy. Bezpośrednie odwołanie, np. `Kowalski.imie`, spowoduje błąd, ale już `Nowak.powitanie();` — nie. Dzieje się tak, ponieważ metoda `powitanie` jest obecna wewnątrz klasy `Klient`, na podstawie której powstały obiekty `Kowalski` oraz `Nowak` (rysunek 2.32).

```
class Klient {
    private imie: string;
    private nazwisko: string;
    wiek: number;
    constructor(imie: string, nazwisko: string, wiek: number) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }

    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}
class DaneKlienta extends Klient {
    adres: string;
    constructor(imie: string, nazwisko: string, wiek: number, adres: string) {
        super(imie, nazwisko, wiek);
        this.adres = adres;
    }
    powitanie() { console.log(`Dzień dobry, ${this.imie} ${this.nazwisko}`); }
}

let Nowak = new Klient("Tadeusz", "Nowak", 44)
let Kowalski = new Klient("Jan", "Kowalski", 17);

Nowak.powitanie();
Kowalski.imie();
```

Rysunek 2.32. Użycie modyfikatora dostępu private

Zastosowanie modyfikatora dostępu `protected` sprawi, że pole stanie się **prywatne** w obrębie danej klasy oraz klas, które po niej dziedziczą — nie wystąpi błąd w metodzie `powitanie` w klasie `DaneKlienta`. Nadal jednak jest zabroniony dostęp z zewnątrz (rysunek 2.33).

```
class Klient {
    protected imie: string;
    protected nazwisko: string;
    wiek: number;
    constructor(imie: string, nazwisko: string, wiek: number) {
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }

    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}
class DaneKlienta extends Klient {
    adres: string;
    constructor(imie: string, nazwisko: string, wiek: number, adres: string) {
        super(imie, nazwisko, wiek);
        this.adres = adres;
    }
    powitanie() { console.log(`Dzień dobry, ${this.imie} ${this.nazwisko}`); }
}

let Nowak = new Klient("Tadeusz", "Nowak", 44)
let Kowalski = new Klient("Jan", "Kowalski", 17);

Nowak.powitanie();
Kowalski.imie();
```

Rysunek 2.33. Użycie modyfikatora dostępu protected

Tych parametrów można również użyć w konstruktorze, dzięki czemu, podobnie jak w przypadku modyfikatora `public`, cały kod skróci się do przedstawionego na listingu 2.54.

Listing 2.54. Użycie modyfikatorów w konstruktorze (I)

```
class Klient {
    constructor(protected imie: string, protected nazwisko: string, public
wiek: number) { }
    powitanie() { console.log(`Witaj, ${this.imie} ${this.nazwisko}`); }
    czyPelnoletni() {
        if (this.wiek >= 18) return true;
        else return false;
    }
}
```

```
class DaneKlienta extends Klient {
    constructor(imie: string, nazwisko: string, wiek: number, public adres: 
string) {
        super(imie, nazwisko, wiek);
    }
    powitanie() { console.log(`Dzień dobry, ${this.imie} ${this.nazwisko}`); }
}

let Nowak = new DaneKlienta("Tadeusz", "Nowak", 44, "Opole, Portowa 17");
let Kowalski = new DaneKlienta("Jan", "Kowalski", 17, "Szczecin, Mickiewicza 
19");

Nowak.powitanie();
console.log(Kowalski.czyPelnoletni())
```

Przedstawiony kod zwróci następujące wartości:

```
Dzień dobry, Tadeusz Nowak
false
```

Pola wraz z metodami w klasach mogą być oznaczone jako `readonly`. Użycie takiego zapisu spowoduje, że **przypisanie do nich wartości poza konstruktorem będzie niemożliwe** (listing 2.55).

Listing 2.55. Użycie modyfikatorów w konstruktorze (II)

```
class uzytkownik {
    imie: string;
    constructor(imie: string = "Tomek") {
        this.imie = imie;
    }
    aktualizujImie() {
        this.imie = "Adam";
    }
}

let Admin = new uzytkownik;
console.log(Admin.imie);
Admin.aktualizujImie();
console.log(Admin.imie);
```

Kod przedstawiony na powyższym listingu skompiluje się, a wynikiem jego wykonania będzie wyświetlenie wartości:

```
Tomek
Adam
```

Początkowe imię `Tomek` zostaje zastąpione przez `Adam`.

Dodanie parametru `readonly` spowoduje błąd: `Cannot assign to 'imie' because it is a read-only property` (rysunek 2.34).

```
class Uzytkownik {
    readonly      readonly imie: string;
    construct     (property) Uzytkownik.imie: any
        this.     Cannot assign to 'imie' because it is a read-only property. ts(2540)
    }
    aktualizu     View Problem (Alt+F8)   No quick fixes available
        this.imie = "Adam";
    }
}
let Admin = new Uzytkownik;
console.log(Admin.imie);
Admin.aktualizujImie();
console.log(Admin.imie);
```

Rysunek 2.34. Użycie parametru readonly

UWAGA

Zastosowanie parametru `readonly` nie wyklucza użycia modyfikatorów dostępu — można utworzyć pole, które będzie np. typu `readonly protected`.

Interfejsy

Zadaniem **interfejsu** jest opisanie kształtu np. obiektu przez określenie zbioru jego właściwości i metod.

Utworzenie interfejsu rozpoczynamy od słowa kluczowego `interface`, po którym następuje nazwa. W nawiasie klamrowym umieszczamy nazwy pól wraz z typem. Użyta nazwa pola i jego typ będą odzwierciedlały kształt obiektu lub klasy, do której interfejs będzie przypisany.

Na listingu 2.56 interfejs o nazwie `InterfejsKlient` za pomocą znaku dwukropka zostaje powiązany z obiektem `klient`. Dzięki powiązaniu obiekt **musi spełnić wszystkie wymagania interfejsu, tj. zdefiniowane nazwy i typy pól interfejsu muszą mieć odzwierciedlenie w obiekcie.**

Listing 2.56. Utworzenie interfejsu

```
interface InterfejsKlient {
    imie: string;
    nazwisko: string;
    wiek: number;
}

let klient: InterfejsKlient = {
    imie: "Tomasz",
    nazwisko: "Tryla",
    wiek: 35,
};
```

```
console.log(klient.imie);
```

Niewłaściwe nazwa pola (rysunek 2.35) lub typ spowodują błąd.

Rysunek 2.35. Błąd w nazwie pola

Zadanie 2.19.

Zaprojektuj interfejs do klasy `Samochody` z zadania 2.16. Utworzony interfejs połącz z klasą.

Pola w interfejsach mogą być **opcjonalne** lub **tylko do odczytu**. Służą do tego celu parametry `?` i `readonly`.

Aby zdefiniować pole opcjonalne, podobnie jak w przypadku argumentów funkcji należy zastosować znak zapytania. Użycie ? po nazwie pola ustawi je jako pole opcjonalne (listing 2.57). Utworzenie obiektu bez definiowania go jest możliwe. Wartość do właściwości obiektu możemy przypisać w dowolnej chwili.

Listing 2.57. Pole opcjonalne

```
interface InterfejsKlient {
    imie: string;
    nazwisko: string;
    wiek: number;
    email?: string
}

let klient: InterfejsKlient = {
    imie: "Tomasz",
    nazwisko: "Tryla",
    wiek: 35,
};

klient.email = "tomtryl@gmail.com";
console.log(klient.email);
```

Użycie parametru `readonly` **uniemożliwi** zmianę danej własności (pole `imie`) po utworzeniu obiektu (rysunek 2.36).

Rysunek 2.36. Użycie parametru readonly

Interfejsy i obiekty można również stosować w połączeniu z funkcjami.

Interfejs `InFaktura` zostaje użyty do określenia kształtu obiektu, a następnie obiekt ten staje się argumentem funkcji `faktura` (funkcja zadeklarowana tradycyjnie) oraz `innaFaktura` (funkcja strzałkowa) (listing 2.58).

Listing 2.58. Użycie interfejsu

```
interface InFaktura {
    doZaplaty: number;
    od: string;
}

const kwota: InFaktura = {
    doZaplaty: 230,
    od: "Temp"
};

function faktura(a: InFaktura) {
    return `Twoja faktura od ${a.od} opiewa na kwotę ${a.doZaplaty}.`;
}

console.log(faktura(kwota));
// funkcja strzałkowa
let innaFaktura = (a: InFaktura) => console.log(`Twoja faktura od ${a.od}
opiewa na kwotę ${a.doZaplaty}.`);
innaFaktura(kwota);
```

Do zdefiniowania typu parametrów funkcji możemy się posłużyć słowem kluczowym `type`, po którym określamy nazwę tworzonego typu oraz podajemy, jakiego typu są parametry i jaki typ funkcja zwraca (listing 2.59).

Funkcja `suma` do działania wymaga dwóch parametrów, `x` i `y` — oba są typu `number`, podobnie jak zwracany wynik.

Funkcja `czyPrawda` również wymaga dwóch parametrów, przy czym pierwszy jest typu `string`, drugi zaś — `number`. Zwracany typ to `boolean`.

Listing 2.59. Definicja typu parametrów funkcji

```
type mojTyp = (x: number, y: number) => number
let suma: mojTyp = (x, y) => x + y;

type test = (name: string, age: number) => boolean;
const czyPrawda: test = (name, age) => name.length > 5 || age > 18;

console.log(suma(35, 23));
console.log(czyPrawda("Magdalena", 10));
console.log(czyPrawda("Adam", 30));
console.log(czyPrawda("Ewa", 12));
```

Wynikiem działania kodu jest wyświetlenie w konsoli następującej zawartości:

```
58
true
true
false
```

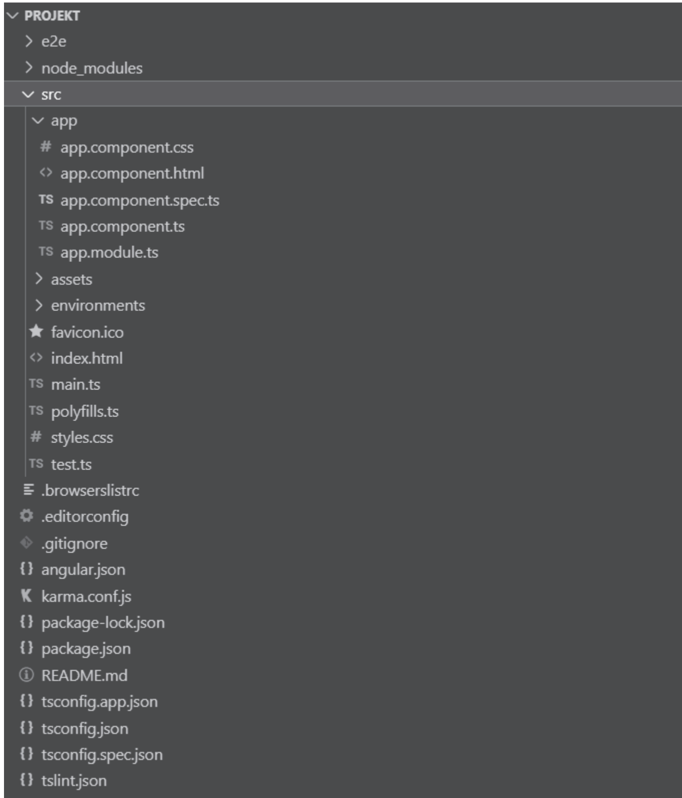
`58` to wynik dodawania, pierwsza wartość `true` jest zwracana dlatego, że długość argumentu `name` to więcej niż `5` znaków, druga wartość `true` jest zwracana, gdyż argument `age` jest większy niż `18`, z kolei wartość `false` bierze się stąd, że żaden z przekazanych argumentów nie spełnił warunku — długość pierwszego wynosi mniej niż `5` znaków, a drugi jest mniejszy od `18` (oba warunki są połączone operatorem || (`or`), a każdy z nich zwraca `false`, więc finalnie całe wyrażenie też zwraca `false`).

Pytania kontrolne

1. Czym jest klasa? Po co tworzy się klasy?
2. Na czym polega mechanizm dziedziczenia klas?
3. Jak działają modyfikatory klasy?
4. Jaką rolę odgrywa interfejs?
5. Która metoda jest lepsza: tradycyjne typowanie obiektów czy użycie interfejsu?

2.3. Angular — pierwsze kroki

Po wygenerowaniu projektu struktura plików aplikacji jest następująca (rysunek 2.37):



Rysunek 2.37. Struktura plików aplikacji Angular

Wszystkie elementy użyte do zbudowania aplikacji Angular (szablony, style, obrazy, usługi) i niezbędne do jej działania znajdują się w katalogu *src*. Oto najważniejsze z nich:

- *app/app.component.(ts, html, css, spec.ts)* — są to cztery pliki o wspólnej nazwie *app.component* z rozszerzeniami podanymi w nawiasie. Pierwszy (*app.component.ts*) zawiera definicje głównego komponentu aplikacji, drugi jest szablonem HTML, trzeci arkuszem stylów CSS, a ostatni jest wykorzystywany do testów. Wraz z rozwojem aplikacji pojawią się tu pliki zagnieżdżonych komponentów.
- *app/app.module.ts* — ten plik określa konfigurację aplikacji.
- *assets/** — katalog z plikami wykorzystywanymi przez aplikację (obrazy, wideo, dźwięk).
- *index.html* — główna strona HTML aplikacji. Angular automatycznie dodaje do tego pliku wszystkie pliki *.js* i *.css* użyte podczas jej budowania.
- *node_modules/* — katalog utworzony przez Node.js. Zawiera wszystkie wymagane moduły używane przez aplikację.

- *angular.json* — konfiguracja dla Angular CLI.
- *package.json* — konfiguracja npm z listą pakietów użytych w aplikacji.
- *tsconfig.json* — konfiguracja kompilatora języka TypeScript.

Jednym z najważniejszych plików jest *app.component.ts* w katalogu *app*. Zawiera on definicję **komponentu** `AppComponent`, który jest „korzeniem” całej aplikacji.

Komponenty frameworka Angular to podstawowy budulec tworzonej aplikacji. Zarządzają szablonem i dostarczają do niego dane.

W pliku *app.component.ts* możemy wyróżnić trzy sekcje. Pierwsza zawiera polecenie `import` i odpowiada za wczytanie modułu `@angular/core`, który zarządza pracą komponentów. Druga, zaczynająca się od znaku `@`, to przykład tzw. **dekoratora**, dostarczającego informacji konfiguracyjnych za pomocą właściwości, którymi są: `selector`, `templateUrl` oraz `styleUrls`. Właściwość `selector`, której wartość odnajdziemy również w pliku *index.html* (umieszczoną w znacznikach <>), wskaże frameworkowi, że w tym miejscu za treść odpowiada komponent. To tutaj Angular wstrzyknie zawartość **szablonu komponentu**. Właściwości `templateUrl` oraz `styleUrls` wskazują frameworkowi Angular, gdzie znajduje się szablon powiązany z komponentem oraz formatujący go arkusz stylów.

Ostatnia sekcja to definicja klasy, która posłużyła do utworzenia komponentu (listing 2.60).

Listing 2.60. Domyślna zawartość pliku app.component.ts

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

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

Zawartość wyświetlanego **szablonu** znajduje się w pliku *./app.component.html*, na co wskazuje znacznik `templateUrl`. Szablon zawiera informacje wyświetlane przez aplikację. Do budowy widoku używa się składni języka HTML oraz dodatków frameworka Angular. W polu `title` znajduje się nazwa projektu (rysunek 2.38).

```
src > app > TS app.component.ts > AppComponent
1   import { Component } from '@angular/core';
2
3   @Component({
4     selector: 'app-root',
5     //template:,
6     templateUrl: './app.component.html',
7     //styleUrls: ['./app.component.css']
8   })
9   export class AppComponent {
10   title = 'app';
11  }
12
```

Rysunek 2.38. Zawartość pliku app.component.ts

2.3.1. Jednokierunkowe techniki wiązania danych

Interpolacja (string interpolation)

Utwórzmy nowy plik HTML o nazwie *moj_plik.html* i zmieńmy domyślną ścieżkę do pliku szablonu tak, aby wskazywała na niego.

W pliku pomiędzy znacznikami `<h1></h1>` został umieszczony tekst `Witaj`. Aby go wyświetlić, należy zmienić domyślną ścieżkę do szablonu na `templateUrl: './moj_plik.html'`, Po zapisaniu pliku następuje uaktualnienie okna przeglądarki (rysunek 2.39).

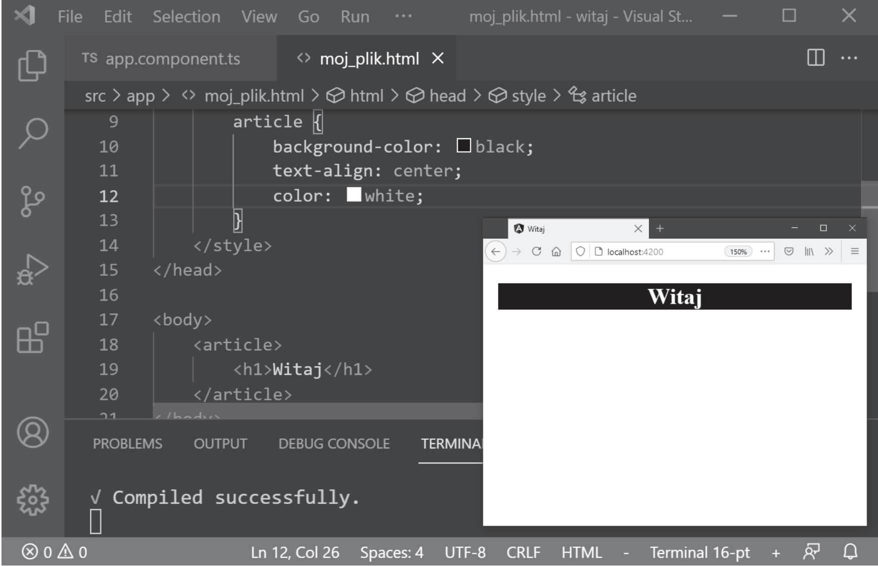
Rysunek 2.39. Zawartość pliku moj_plik.html

Plik ten oprócz znaczników języka HTML może zawierać style CSS.

Nasz napis zostaje umieszczony pomiędzy znacznikami `<article></article>`, a zawartość jest formatowana za pomocą arkusza stylów, który zmienia kolor tła i tekstu oraz go wyśrodkowuje (rysunek 2.40).

Styl można zdefiniować w dowolnym pliku CSS. Aby to zademonstrować, w katalogu *app* utworzono plik o nazwie *style.css*. Podpięcie stylu, podobnie jak pliku HTML, następuje w pliku *app.component.ts* — po usunięciu znacznika komentarza z linijki zawierającej `styleUrls:` zmieniamy domyślną ścieżkę na: *./style.css* (rysunek 2.41).

Szablon może się znajdować także w pliku *app.component.ts*. W tym celu należy usunąć znacznik komentarza umieszczony przed `template` i pomiędzy znakami ` `` `, `""` lub `''` zdefiniować zawartość.



Rysunek 2.40. Zawartość pliku moj_plik.html po zmianie — dodaniu stylu CSS

```
import { Component } from '@angul...

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

```
article {
    background-color: blue;
    text-align: center;
    color: white;
}
```

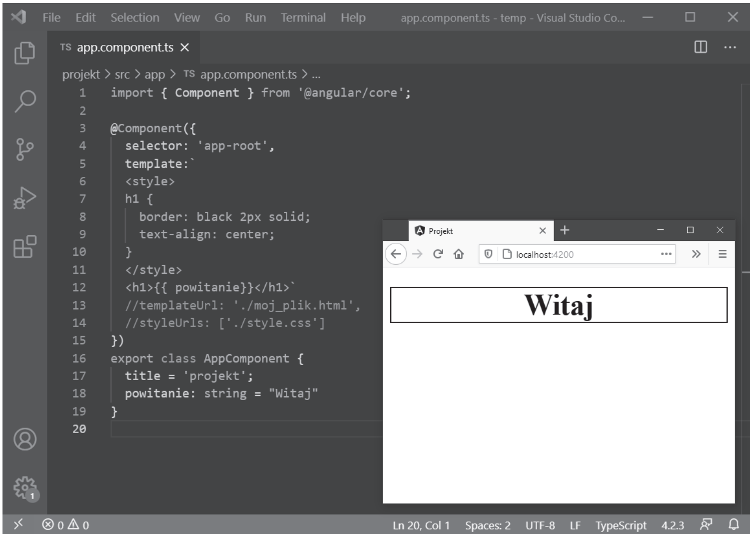
Rysunek 2.41. Definicja stylu CSS zapisanego we własnym pliku

WSKAZÓWKA

Pierwsze rozwiązanie, czyli umieszczenie zawartości szablonu pomiędzy znakami ``, jest najwygodniejsze, gdyż pozwala tworzyć wielowierszowe konstrukcje.

Dodajmy do naszego komponentu (jego definicja znajduje się w klasie AppComponent) pole powitanie, a jego wartość ustalmy na Witaj. Wartość zdefiniowanego w ten sposób pola można przekazać do szablonu — w tym celu należy umieścić **nazwę pola pomiędzy podwójnymi nawiasami klamrowymi ({{}})**. Mechanizm ten nazywa się **interpolacją** lub **string interpolation**.

Pole to zostało umieszczone pomiędzy znacznikami <h1></h1>, a wygląd ustalono za pomocą stylu (obramowanie koloru czarnego o rozmiarze 2 px w postaci linii ciągłej i z wyśrodkowanym tekstem), którego definicja znajduje się w tym samym pliku (rysunek 2.42).



Rysunek 2.42. Umieszczenie wartości pola w szablonie — interpolacja

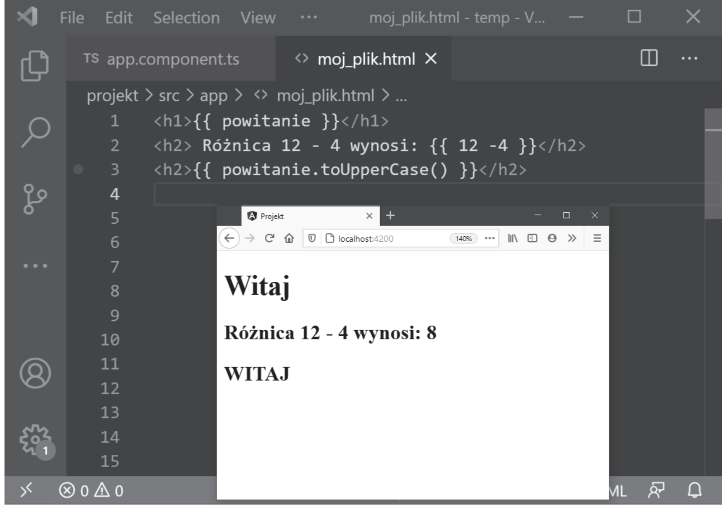
Oczywiście tego typu wstrzyknięcie zawartości komponentu zadziała także wtedy, gdy szablon znajduje się w zewnętrznym pliku HTML.

Wewnątrz podwójnych nawiasów klamrowych oprócz nazw pól komponentów można również umieścić **wyrażenia** (różnica liczb 12 i 4), a także wywoływać **metody** (wartość przypisana do pola powitanie jest wyświetlana dużymi literami w wyniku użycia metody toUpperCase) (rysunek 2.43).

Wiązanie właściwości (property binding)

Jak już wiesz, elementy w szablonie można umieszczać przez ich bezpośrednie definiowanie, a także z wykorzystaniem interpolacji. Jest jeszcze jeden sposób: **wiązanie właściwości** (ang. *property binding*).

Użycie wszystkich metod jest pokazane na listingu 2.61.



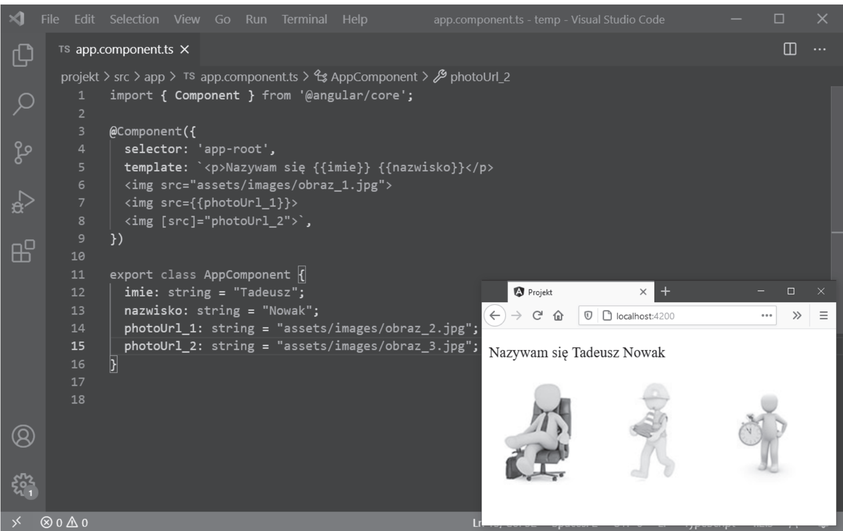
Rysunek 2.43. Zastosowanie interpolacji

Listing 2.61. Sposoby umieszczenia danych w szablonie

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

@Component({
  selector: 'app-root',
  template: `<p>Nazywam się {{imie}} {{nazwisko}}</p>
    <img src="assets/images/obraz_1.jpg">
    <img src={{photoUrl_1}}>
    <img [src]="photoUrl_2">`,
})

export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  photoUrl_1: string = "assets/images/obraz_2.jpg";
  photoUrl_2: string = "assets/images/obraz_3.jpg";
}
```

Użycie zapisu `<img [src]="photoUrl_2">` mówi frameworkowi Angular, że na elemencie `img` istnieje właściwość `src` (zapisujemy ją pomiędzy nawiasami kwadratowymi), której wartość powinna zostać pobrana z pola `photoUrl_2`.

Wynikiem wywołania kodu jest umieszczenie w widoku napisu `Nazywam się Tadeusz Nowak` oraz trzech zdjęć z katalogu *./assets/images* (rysunek 2.44).



Rysunek 2.44. Sposoby umieszczania danych w szablonie

Tych mechanizmów będziemy używali w zależności od typu przekazywanych wartości. **Interpolacja nie nadaje się do przekazywania argumentów np. do funkcji, które nie są typu `string`.**

Przykład z listingu 2.62 pokazuje zastosowanie metody wiązania właściwości.

Listing 2.62. Przykłady użycia mechanizmu wiązania właściwości

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

@Component({
  selector: 'app-root',
  template: `
    <!--przykład pierwszy-->
    <p [style.background-color]="kolor" [style.text-align]="jak"> Nazywam się
{{imie}} {{nazwisko}}</p>
    <!--przykład drugi-->
    <img src="assets/images/obraz_1.jpg" [class.zdjecie]="aktywna">
  `,
  styles: ['.zdjecie {border: 2px solid #000; display: block; margin: auto;}']
})

export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
}
```

Pierwszy przykład pokazuje, jak za pomocą wiązania właściwości zmienić styl elementu. W tym celu zdefiniowano dwie właściwości: `kolor` o wartości `yellow` oraz `jak` o wartości `center`. Właściwości te zostały użyte do zmiany wyglądu akapitu, w którym znajduje się napis `Nazywam się Tadeusz Nowak` — kolor tła zmienił się na żółty, a tekst jest teraz wycentrowany. Zapis `<p [style.background-color]="kolor" [style.text-align]="jak">` powoduje zmianę stylu.

Drugi przykład dodaje nową klasę, czego efektem jest zmiana wyglądu zdjęcia. Do znacznika `img` zostaje dodany kod `[class.zdjecie]="aktywna"`, który inicjuje klasę (przekazywana jest wartość `true`). Definicja stylu klasy powinna być umieszczona w selektorze `styles`. Pole to uaktywnia lokalny arkusz stylu, który jest definiowany bezpośrednio w pliku komponentu. Użyte znaczniki CSS powodują dodanie czarnego obramowania zdjęcia o szerokości 2 px oraz jego wyśrodkowanie.

Wiązanie zdarzeń (event binding)

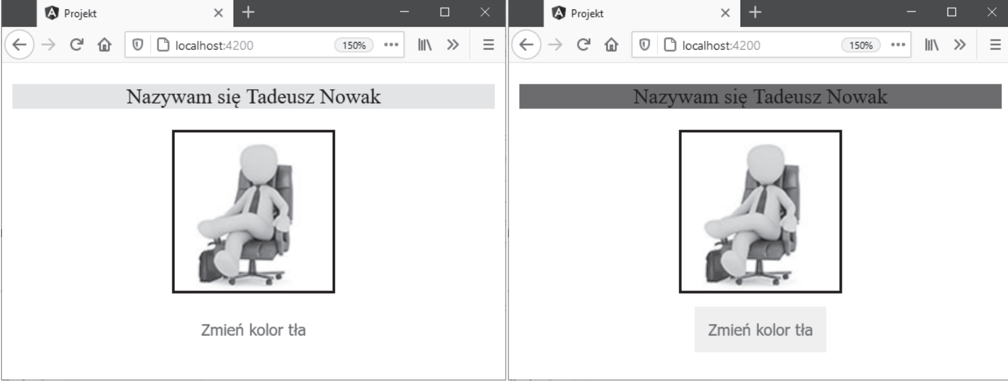
W Angularze **wiązanie zdarzeń** (ang. *event binding*) jest wykorzystywane do obsługi zdarzeń. Zaliczamy do nich np. kliknięcie przycisku myszy, ruch wskaźnika myszy i zmianę wartości tekstowej. Taka interakcja powoduje wywołanie w komponencie określonej metody.

Przykład zdarzenia jest pokazany na listingu 2.63. Za pomocą znacznika `<button>` zostaje dodany przycisk, którego kliknięcie powoduje (typ zdarzenia umieszczamy w nawiasie okrągłym) wykonanie metody `zmienKolor()` — zmianę koloru tła pod napisem. Wraz z przyciskiem zostają zdefiniowane dwie klasy: pierwsza określa jego wygląd, a druga — zachowanie po nakierowaniu na niego wskaźnika myszy (pseudoklasa `:hover`). Metoda `zmienKolor()` zostaje zdefiniowana w komponencie — odpowiada ona za zmianę koloru tła akapitu (rysunek 2.45).

Listing 2.63. Wiązanie zdarzeń

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

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{imie}} {{nazwisko}}</p>
    <img src="assets/images/obraz_1.jpg" [class.zdjecie]="aktywna">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>`,
  styles: [`
    .zdjecie {border: 2px solid #000; display: block; margin: auto;}
    .przycisk {border: none; color: green; background-color: inherit; padding:
10px; font-size: 12px; cursor: pointer; display: block; margin: 10px auto;}
    .przycisk:hover {background: #eee;}
  `]
})
```

```
export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```



Rysunek 2.45. Użycie wiązania zdarzeń

Zastosowany w metodzie `zmienKolor()` zapis `this.kolor = this.kolor === "yellow" ? "green" : "yellow";` to tzw. **wyrażenie warunkowe**. Oznacza ono: jeżeli warunek jest prawdziwy, przypisz do `this.kolor` wartość `green`, w przeciwnym razie użyj `yellow`. Należy je traktować jako skróconą wersję instrukcji warunkowej `if`.

Podane przykłady to jednokierunkowe techniki wiązania danych, wykorzystywane do umieszczania w szablonie (widoku) danych pochodzących z kodu TypeScript.

UWAGA

Zazwyczaj w definicji zdarzenia podaje się nazwę metody, która ma zostać użyta po zaistnieniu zdarzenia, ale nic nie stoi na przeszkodzie, aby zamiast nazwy metody umieścić tu jej definicję. Oznacza to, że kod `this.kolor = this.kolor === "yellow" ? "green" : "yellow";` może się znajdować bezpośrednio w szablonie. Jednak to, co dozwolone, nie zawsze jest zalecane — podczas analizy kodu (bądź gdy trzeba do niego wrócić po jakimś czasie) pierwszy wariant wydaje się lepszym podejściem.

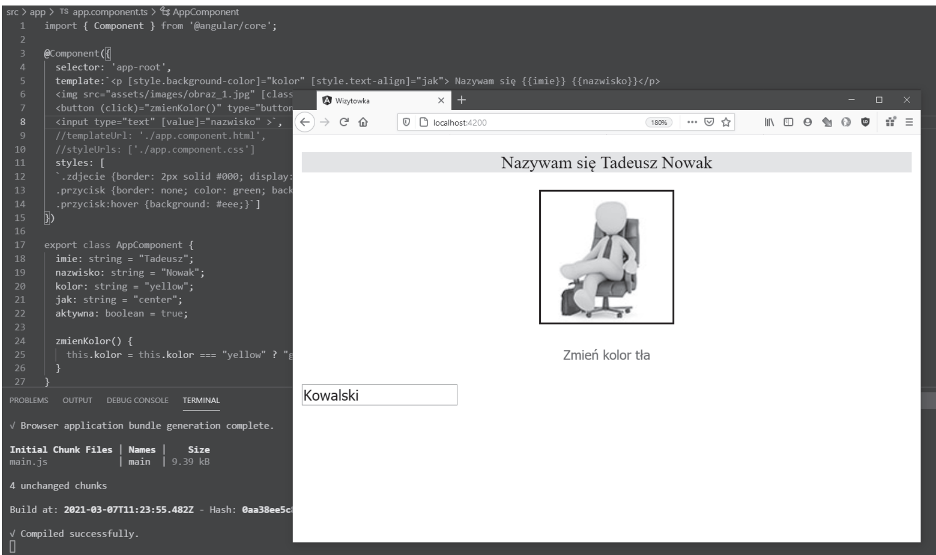
Zadanie 2.20.

Przygotuj trzy pliki: prosty szablon, arkusz stylów, który go formatuje, i swoje zdjęcie. Za pomocą wybranych technik wiązań jednokierunkowych stwórz prostą wizytówkę.

2.3.2. Wiązanie dwukierunkowe (two-way binding)

Wiązanie jednokierunkowe nie pozwala na odzwierciedlenie zmian szablonu w kodzie TypeScript. Dlatego też framework Angular został wyposażony w tzw. **mechanizm wiązania dwukierunkowego** (ang. *two-way binding*), którego zadaniem jest przekazywanie danych pochodzących z komponentu do widoku i na odwrót. Synchronizacja ta następuje automatycznie.

Na przykład dodanie do szablonu linijki `<input type="text" [value]="nazwisko">` spowoduje wyświetlenie pola formularza z wartością `Nowak`, lecz jakakolwiek jego zmiana nie doprowadzi do aktualizacji właściwości `nazwisko` (rysunek 2.46).

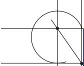


Rysunek 2.46. Dodanie pola formularza

Aby to zmienić, należy utworzyć **wiązanie dwukierunkowe** według wzoru:

```
[(ngModel)] = "właściwość zdefiniowana w komponencie"
```

Najpierw używamy słowa `ngModel`, umieszczonego pomiędzy nawiasami okrągłymi i kwadratowymi, a następnie, po znaku `=`, wpisujemy nazwę właściwości zdefiniowanej w komponencie. Ponieważ wiązanie dwukierunkowe jest **połączeniem wiązania właściwości oraz wiązania zdarzeń**, jego składnia zawiera parę nawiasów kwadratowych oraz okrągłych (listing 2.64).



WSKAZÓWKA

Aby zapamiętać kolejność stosowania nawiasów, wyobraź sobie banany w skrzynce.

Listing 2.64. Użycie wiązania dwustronnego — plik app.component.ts

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

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{imie}} {{nazwisko}}</p>
    <img src="assets/images/obraz_1.jpg" [class.zdjecie]="aktywna">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <input type="text" [(ngModel)]="nazwisko" >`,
  styles: [`
    .zdjecie {border: 2px solid #000; display: block; margin: auto;}
    .przycisk {border: none; color: green; background-color: inherit; padding:
10px; font-size: 12px; cursor: pointer; display: block; margin: 10px auto;}
    .przycisk:hover {background: #eee;}
  `]
})

export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Zapisanie projektu wywoła błąd kompilacji. Angular nie umie rozpoznać, czym jest `ngModel` (rysunek 2.47).

Rysunek 2.47. Błąd — nieznana właściwość ngModel

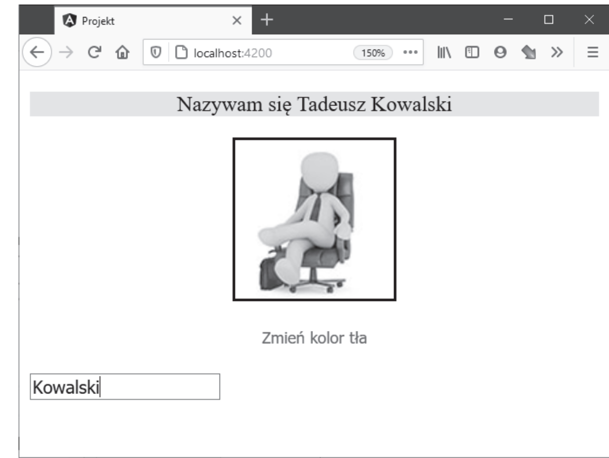
Użyty moduł odpowiada za obsługę formularzy (powrócimy do niego w podrozdziale 2.7) i w domyślnej konfiguracji nie jest importowany, tak więc aby móc skorzystać z tego typu wiązań, musimy **włączyć dyrektywę ngModel**. Jest ona zależna od pakietu `FormsModule`, którego definicja musi się znaleźć w pliku modułu głównego *app.module.ts* (listing 2.65).

W pliku *app.module.ts* importujemy identyfikator `FormsModule` z biblioteki *angular/forms*, a następnie w tablicy `imports` dopisujemy `FormsModule`.

Listing 2.65. Zawartość pliku app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Po zapisaniu zmian i skompilowaniu kodu wiązanie dwustronne działa. Zmiana zawartości pola `input` powoduje aktualizację właściwości `nazwisko` (rysunek 2.48).



Rysunek 2.48. Wiązanie dwustronne

Ponieważ rozwijanego kodu użyjemy w następnym rozdziale, wprowadźmy w nim pewne modyfikacje.

Dwa przedstawione poniżej listingi, 2.66 oraz 2.67, nie zmieniają w żaden sposób funkcjonalności przykładu, lecz jedynie go porządkują — część właściwości zostaje zgrupowana w jeden obiekt, którego kształt jest określany przez interfejs zdefiniowany w pliku *interfejs.ts.*

Listing 2.66. Zawartość pliku app.component.ts

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

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <img src={{osoba.zdjecie}} [class.zdjecie]="aktywna">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <input type="text" [(ngModel)]="osoba.nazwisko">`,
  styleUrls: ['./app.component.css']
 })

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Listing 2.67. Zawartość pliku interfejs.ts

```
export interface Osoba {
    imie: string;
    nazwisko: string;
    zdjecie: string;
}
```

W pliku *interfejs.ts* został utworzony interfejs o nazwie `Osoba` (zgodnie z opisem zamieszczonym w punkcie 2.2.6). Plik ten ma trzy pola: `imie`, `nazwisko` oraz `zdjecie`, wszystkie typu `string`. Ponieważ pola tego interfejsu mają być dostępne w pliku *app.component.ts*, przed jego definicją zostało umieszczone słowo `export`.

Definicja interfejsu `Osoba` zostanie wykorzystana w pliku *app.component.ts*, dlatego interfejs musi zostać do niego zaimportowany. Aby to zrobić, należy dodać w kodzie linijkę `import { Osoba } from './interfejs';`. W nawiasach klamrowych określamy nazwy importowanych elementów, a po słowie `from` — plik, z którego pochodzą. Oba pliki znajdują się w tym samym katalogu, dlatego nazwa pliku rozpoczyna się od znaków `./`, po których następuje nazwa pliku (opuszczamy rozszerzenie).

Zaimportowany interfejs `Osoba` posłużył do utworzenia obiektu `osoba`. Ponieważ interfejs określa **kształt obiektu**, wszystkie pola obecne w interfejsie muszą mieć odzwierciedlenie w obiekcie, do którego interfejs został przypisany. Ze względu na to, że już nie odwołujemy się do zmiennej zadeklarowanej w komponencie, lecz do obiektu, wszystkie wywołania należy poprzedzić **nazwą obiektu.** Dlatego np. musiał się zmienić zapis — z `imie` na `osoba.imie` itd.

Jeśli nie wykonamy importu, podczas kompilacji nastąpi błąd. Visual Studio Code wspomaga nas w tym zadaniu — wystarczy, że klikniemy element, o którym wiemy, że pochodzi z innego pliku, i wybierzemy skrót *Ctrl+I*, a wtedy edytor podpowie, w jakim pliku znajduje się jego definicja (rysunek 2.49). Innym sposobem jest użycie **szybkiego rozwiązania**, tzw. *Quick Fix*, dostępnego w opisie błędu. Kliknięcie go wprowadzi w nagłówku pliku stosowne instrukcje.

Dodatkowo definicja lokalnego stylu została przeniesiona do pliku *./app.component.css*.

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


@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.te
  <img src={{osoba.zdjecie}} [class.zdjecie]="aktywna">
  <button (click)="zmienKolor()" type="button" [class.przy
  <input type="text" [(ngModel)]="osoba.nazwisko">`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
  Osoba src/app/interfejs
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }
```

Rysunek 2.49. Użycie skrótu Ctrl+I

Pytania kontrolne

1. W jaki sposób przekazać dane do szablonu?
2. Jakie są ograniczenia jednokierunkowych technik wiązania danych?
3. Czy wiązania dwukierunkowe pozwalają pokonać te ograniczenia?

2.4. Dyrektywy

Dyrektywy wbudowane we framework Angular umożliwiają dostęp do struktury dokumentu. Z wykorzystaniem dyrektyw możemy **zmieniać zarówno wygląd, jak i zachowanie danego elementu**. Angular zapewnia wiele wbudowanych dyrektyw (rozpoczynają się one od prefiksu `ng`), ale także pozwala tworzyć własne.

Dyrektywy w Angularze dzielimy na trzy kategorie, w zależności od ich zachowania:

- **komponenty** (ang. *Component Directives*),
- **dyrektywy strukturalne** (ang. *Structural Directives*),
- **dyrektywy atrybutowe** (ang. *Attribute Directives*).

Definiowanie komponentu rozpocząłeś (nieświadomie) w poprzednim rozdziale; w następnym powrócimy do tego tematu.

2.4.1. Dyrektywy strukturalne

Dyrektywy strukturalne zmieniają strukturę drzewa dokumentów przez dodawanie, usuwanie lub modyfikowanie jego elementów, a co za tym idzie, mają **wpływ na strukturę kodu HTML oraz jego układ**.

Dyrektywa ta zostaje przypisana do konkretnego elementu i łatwo ją rozpoznać, ponieważ poprzedza się ją znakiem gwiazdki (`*`).

Dyrektywa ngIf

Dyrektywa `ngIf` jest używana do **dodawania lub usuwania elementów HTML**. Jeżeli wartość to **false**, element jest usuwany z drzewa dokumentów, a jeśli **true** — jest w nim uwzględniany.

UWAGA

Dyrektywa `ngIf` nie ukrywa elementu, ale nie uwzględnia go w drzewie dokumentów.

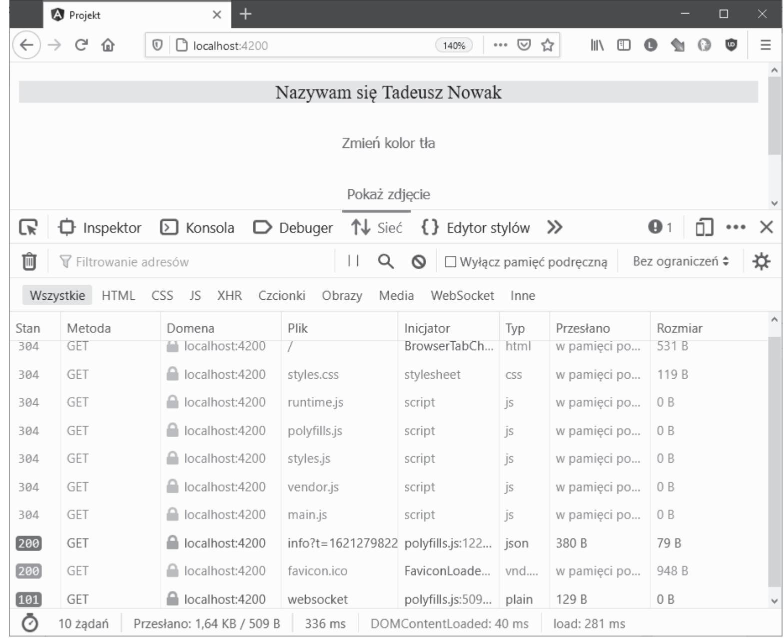
Abyś zobaczył, jak działa dyrektywa `ngIf`, zmodyfikujmy listing 2.67 i ukryjmy zdjęcie. Będzie można je wyświetlić przez wciśnięcie przycisku (listing 2.68).

Został dodany nowy przycisk z etykietą `Pokaż zdjęcie`. Kliknięcie go uruchamia zdarzenie — zmianę wartości zmiennej `pokazZdjecie` z `false` na `true` bądź odwrotnie. Domyślna wartość zmiennej `pokazZdjecie` jest ustalona na `false`. Dyrektywa `ngIf` zostaje zdefiniowana w znaczniku <img> i powiązana z `pokazZdjecie` (deklarację rozpoczynamy od gwiazdki). Dodatkowo został dodany akapit, który wyświetla bieżącą wartość zmiennej `pokazZdjecie`. Po skompilowaniu kodu zdjęcie nie zostaje załadowane.

Listing 2.68. Dyrektywa ngIf

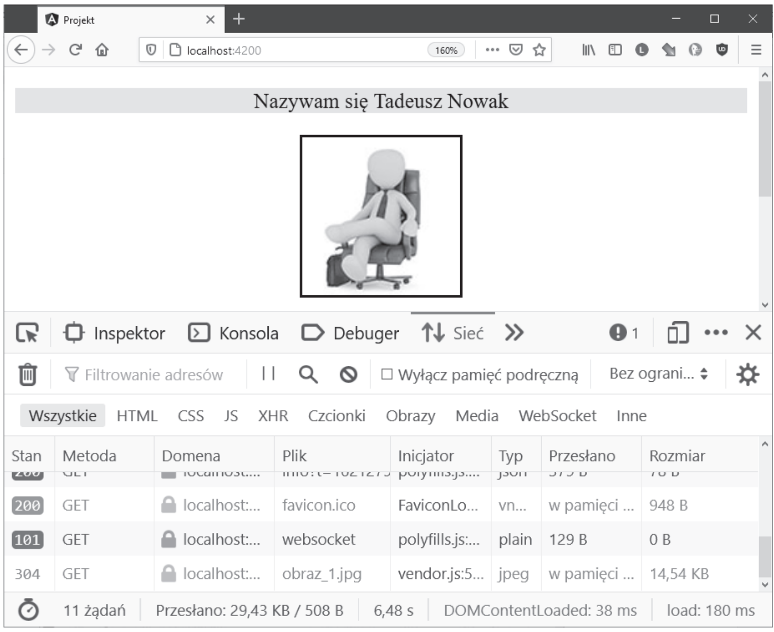
```
import { Component } from '@angular/core';
import { Osoba } from './interfejs';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <button (click)="pokazZdjecie=!pokazZdjecie" type="button" [class.
przycisk]="aktywna">Pokaż zdjęcie </button>
    <p> Obecny status właściwości 'pokazZdjecie' = {{pokazZdjecie}} </p>`,
    styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Aby zweryfikować, czy obraz **nie jest uwzględniany** w drzewie dokumentu, należy uruchomić narzędzia deweloperskie przeglądarki (klawiszem *F12*) i wybrać zakładkę *Sieć*. Jak można się przekonać, na liście żądań *GET* nie ma tego, które nakazuje pobranie obrazka (rysunek 2.50).



Rysunek 2.50. Sposób działania dyrektywy ngIf

Żądanie wczytania zdjęcia (metoda *GET*) pojawia się w chwili kliknięcia przycisku *Pokaż zdjęcie* (rysunek 2.51).



Rysunek 2.51. Sposób działania dyrektywy ngIf

Taki sposób działania dyrektywy powoduje, że jej umiejętne wykorzystanie znacznie wpływa na szybkość działania aplikacji — **jej poszczególne elementy są wczytywane, w miarę jak występują określone zdarzenia**.

Możliwość ukrywania i wyświetlania zdjęcia możemy zapewnić również za pomocą pola typu **checkbox**. Zaznaczenie pola (przekazanie wartości `true`) skutkuje pojawieniem się zdjęcia, natomiast jego brak (przekazanie wartości `false`) — ukryciem. Zmodyfikowany program jest pokazany na listingu 2.69.

Do powiązania pola `checkbox` ze zmienną `pokazZdjecie` została użyta dyrektywa `ngModel`.

Listing 2.69. Użycie dyrektywy ngIf w połączeniu z polem typu checkbox

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

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna" *ngIf="pokazZdjecie"
>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <button (click)="pokazZdjecie=!pokazZdjecie" type="button" [class.
przycisk]="aktywna">Pokaż zdjęcie </button>
    <label><input type="checkbox" [(ngModel)]=pokazZdjecie>Pokaż zdjęcie </
label>
    <p> Obecny status właściwości 'pokazZdjecie' = {{pokazZdjecie}} </p>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Dyrektywa `ngIf` może zostać połączona z klauzulą `else` — określa, co ma się wydarzyć w przypadku, gdy stan będzie miał wartość `false`. Jest to zobrazowane w przykładzie poniżej.

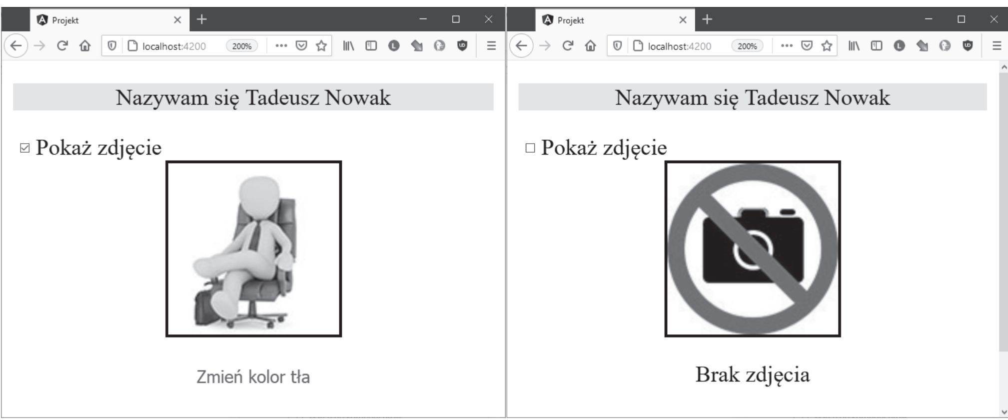
Zapis `*ngIf="pokazZdjecie else brakZdjecia"` oznacza, że jeśli warunek nie będzie spełniony, ma zostać wykonany kod zawarty w **zmiennej szablonowej** `brakZdjecia`. Zmienna ta została zdefiniowana wraz ze znacznikami `<ng-template></ng-template>`, pomiędzy którymi jest zawarta instrukcja — wyświetlenie zdjęcia wraz z podpisem `Brak zdjęcia`. Znaczniki te należy traktować jako **kontener** z kodem, który zostanie wykonany, jeśli nastąpi zdefiniowane zdarzenie. **Gdyby go nie było, kod zostałby zinterpretowany i wyświetlony natychmiast.** Nazwa zmiennej szablonowej musi być poprzedzona znakiem *hash* (`#`). Do określenia ścieżki zdjęcia użyto właściwości `bezZdjecia`. Właściwość znajduje się w obiekcie `osoba`, a jak pamiętasz, obiekt ten został utworzony za pomocą interfejsu, tak więc definicja właściwości również musi się znaleźć w opisie interfejsu. Dodajemy zatem linijkę: `bezZdjecia: string;`. Do pliku *app.component.css* zostaje dodana nowa klasa, `.tekst_srodek`, która wyrównuje tekst. Zmiany w kodzie pokazuje listing 2.70.

Listing 2.70. Dyrektywa ngIf w połączeniu z else

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

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <label><input type="checkbox" [(ngModel)]="pokazZdjecie">Pokaż zdjęcie </
label>
    <div id="ramka">
        <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <ng-template #brakZdjecia>
        <img src="{{osoba.bezZdjecia}}" class="zdjecie">
        <p class="tekst_srodek">Brak zdjęcia</p>
    </ng-template>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg",
    bezZdjecia: "assets/images/nofoto.jpg"
  }
```

```
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Wynik działania kodu jest pokazany na rysunku 2.52.



Rysunek 2.52. Użycie dyrektywy ngIf w połączeniu z else. Po lewej — pole checkbox zaznaczone, po prawej — zaznaczenie pola checkbox usunięte

Zadanie 2.21.

Przygotuj krótki opis miejscowości, w której mieszkasz, oraz dołącz do niego dwa zdjęcia. Opis i zdjęcia wykorzystaj do stworzenia szablonu. Dodatkowo pod opisem umieść dwa przyciski i z użyciem dyrektywy `ngIf` spraw, aby kliknięcie każdego z nich powodowało wyświetlenie innego zdjęcia.

Dyrektywa ngFor

Dyrektywa **`*ngFor`** pozwala na powtarzanie części szablonu HTML dla iterowanego obiektu.

Podstawowa składnia jest następująca:

```
*ngFor="let element of kolekcja;"
```

Przykład użycia tej dyrektywy jest pokazany na listingu 2.71.

Obiekt `osoba` został wzbogacony o tabelę zawierającą adres. Pod pierwszym indeksem jest zapisana `ulica`, pod drugim `kod pocztowy`, pod trzecim zaś `miasto`. Aby można było umieścić te dane w pliku *app.component.ts*, należy zmodyfikować definicję interfejsu, który znajduje

się w pliku *interfejs.ts*, tzn. dodać linijkę: `adres: string[];` (dane będą przechowywane w tablicy). Za wyświetlenie tych informacji odpowiada dyrektywa `ngFor`, zadeklarowana w znaczniku `<li>`. Nakazuje ona pobranie wartości z tablicy `osoba.adres` i przypisanie jej do zmiennej `value`. Wartość zmiennej jest następnie wypisywana za pomocą interpolacji. Elementy tabeli zawierające adres są wyświetlane jako lista punktowana.

Listing 2.71. Użycie dyrektywy ngFor

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

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <label><input type="checkbox" [(ngModel)]="pokazZdjecie">Pokaż zdjęcie </
label>
    <div id="ramka">
        <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <div id="kontakt" class="tekst">
        <p>Kontakt: </p>
        <ul>
            <li *ngFor="let value of osoba.adres"> {{value}} </li>
        </ul>
      </div>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <ng-template #brakZdjecia>
        <img src="{{osoba.bezZdjecia}}" class="zdjecie">
        <p class="tekst_srodek">Brak zdjęcia</p>
    </ng-template>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg",
    bezZdjecia: "assets/images/nofoto.jpg",
    adres: [
      "Piękna 16",
      "40-003",
      "Katowice"
    ]
  }
```

```
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

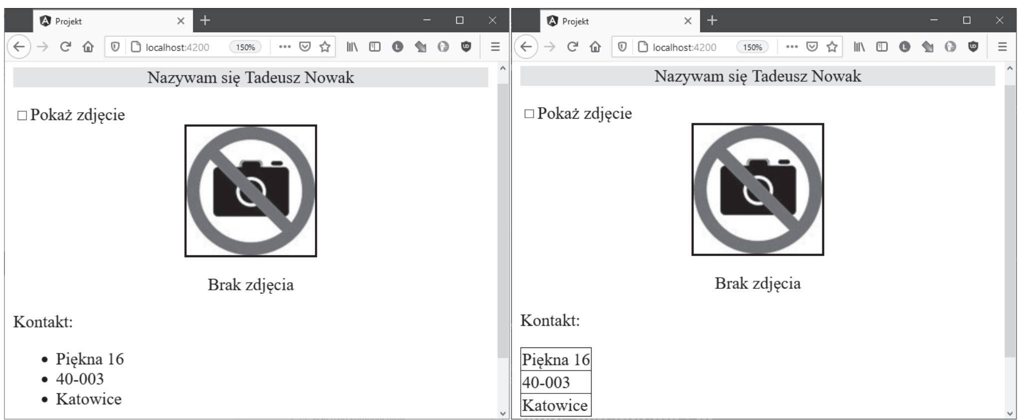
Zamiana kodu objętego znacznikiem `div` o identyfikatorze `kontakt` na:

```
<table>
    <tr *ngFor="let item of osoba.adres">
        <td>{{item}}</td>
    </tr>
</table>
```

spowoduje wyświetlenie adresu w tabeli. Oczywiście za wyświetlenie obramowania tabeli odpowiada styl CSS.

```
table, td {
    border: 1px solid black;
    border-collapse: collapse;
}
```

Efekt zastosowania obu wersji kodu jest pokazany na rysunku 2.53.



Rysunek 2.53. Użycie dyrektywy ngFor. Po lewej — lista wypunktowana, po prawej — tabela

Działaniem dyrektywy ngFor możemy sterować za pomocą zmiennych pokazanych w tabeli 2.2.

Tabela 2.2. Zmienne dyrektywy ngFor

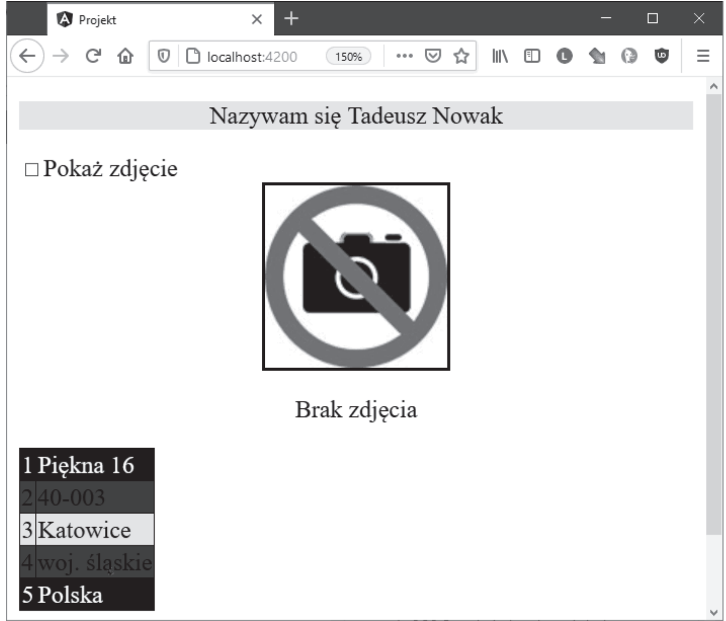
Nazwa zmiennej	Opis
index	Wartość typu number określająca położenie bieżącego obiektu
odd	Wartość logiczna; true oznacza, że położenie bieżącego obiektu jest określone liczbą nieparzystą
even	Wartość logiczna; true oznacza, że położenie bieżącego obiektu jest określone liczbą parzystą
first	Wartość logiczna; true oznacza, że bieżący obiekt jest pierwszym
last	Wartość logiczna; true oznacza, że bieżący obiekt jest ostatnim

Użycie wszystkich zmiennych dyrektywy jest przedstawione w kodzie poniżej (między znacznikami div o identyfikatorze kontakt). Został utworzony szereg zmiennych: zmienna i to index, zmienna parzysta przyjmuje wartość true, gdy w źródle danych położenie zostało określone liczbą parzystą (w omawianym scenariuszu jest to parzysty wiersz tabeli), a zmienne pierwszy i ostatni określają, odpowiednio, pierwszy i ostatni element tabeli. Zależnie od wartości zmiennych do wiersza tabeli zostaje przypisana odpowiednia klasa (listing 2.72).

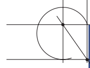
Listing 2.72. Zastosowanie zmiennych dyrektywy ngFor

```
<div id="kontakt" class="tekst">
    <table>
        <tr *ngFor="let item of osoba.adres; let i = index; let parzysta =
even; let pierwszy = first; let ostatni = last"
            [class.parzysta]="parzysta" [class.nieparzysta]="!parzysta" [class.
wyroznienie]="pierwszy || ostatni">
            <td>{{i+1}}</td>
            <td>{{item}}</td>
        </tr>
    </table>
</div>
```

Efektem jest zastosowanie różnego formatowania wierszy tabeli w zależności od tego, czy dany wiersz jest parzysty (kolor żółty — .parzysta { background-color: yellow; }), czy nieparzysty (kolor niebieski — .nieparzysta { background-color: blue; }), ale także od tego, czy jest wierszem pierwszym, czy ostatnim (efekt negatywu — .wyroznienie {background-color: black; color: white; }). Zmienna i (zwiększana o 1) numeruje kolejne wiersze tabeli (rysunek 2.54).



Rysunek 2.54. Użycie zmiennych dyrektywy ngFor



UWAGA

Pamiętaj, że indeksowanie tabeli rozpoczynamy od 0, dlatego wyświetlony numer wiersza nie odpowiada jego faktycznemu indeksowi. Można by sądzić, że wiersz wyświetlający nazwę miasta powinien być koloru niebieskiego, ponieważ jak wskazuje wyświetlona wartość, jest on nieparzysty. W rzeczywistości wartość `Katowice` jest przechowywana w tabeli pod indeksem `2`.

Dyrektywa ngSwitch

W Angularze **ngSwitch** jest dyrektywą strukturalną, która jest używana do dodawania/usuwania elementów w drzewie dokumentów. Sposobem działania przypomina instrukcję `switch` dostępną w innych językach programowania, takich jak C++ i PHP. Przyłączanie odbywa się na podstawie zdefiniowanego wyrażenia i dopasowania warunków.

Ogólna składnia dyrektywy jest następująca:

```
<znacznik [ngSwitch]="wyrażenie">
    <znacznik_wewnętrzny *ngSwitchCase="wartość_1">instrukcja_1</ znacznik_
wewnętrzny>
    <znacznik_wewnętrzny *ngSwitchCase="wartość_2">instrukcja_2</znacznik_
wewnętrzny >
     .
     .
```

```
    <znacznik_wewnętrzny *ngSwitchCase="wartość_n">instrukcja_n</znacznik_
wewnętrzny>
    <znacznik_wewnętrzny *ngSwitchDefault>instrukcja_domyślna</znacznik_
wewnętrzny>
</znacznik>
```

Kod z listingu 2.73 przedstawia sposób użycia dyrektywy `ngSwitch`. Jej zadaniem jest wyświetlenie znaku obok nazwiska w zależności od płci: kobieta — znak Wenus, mężczyzna — znak Marsa, płeć nieokreślona — gwiazdka. Dyrektywa `ngSwitch` została połączona z wyliczeniem zdefiniowanym w pliku *interfejs.ts* (musimy pamiętać o wyeksportowaniu go i zaimportowaniu do pliku, w którym znajduje się szablon). Wyliczenie wraz z modyfikacją interfejsu `Osoba` jest pokazane na listingu 2.74. Sprawdzane wyrażenie umieszczono pomiędzy znacznikami `<ng-container></ng-container>`. Stosujemy je, gdy do szablonu nie chcemy wprowadzać żadnych dodatkowych obiektów języka HTML. Ponieważ wyliczenie (zobacz punkt 2.2.3) zdefiniowanym nazwom przypisuje wartość, wykorzystano to do sterowania dyrektywą `ngSwitch`. Za pomocą instrukcji `*ngSwitchCase` określamy, co ma się stać, gdy warunek zostanie spełniony. Wartość `1` spowoduje wyświetlenie piktogramu *wenus.png*, a wartość `2` — *mars.png*. Domyślnie numerowanie nazw wyliczenia rozpoczyna się od `0`, dlatego w jego definicji przy pierwszym elemencie dopisano `=1`. Ponowna zmiana definicji interfejsu wymusiła zmianę kształtu obiektu `osoba`. Obiekt poszerzono o właściwość `type`, która określa (lub nie) płeć.

Listing 2.73. Użycie dyrektywy ngSwitch

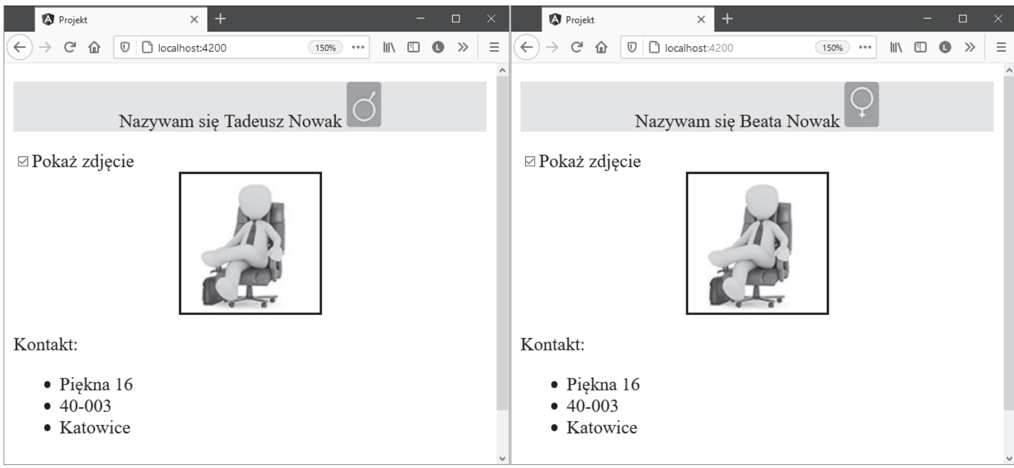
```
import { Component } from '@angular/core';
import { Osoba, Plec } from './interfejs';

@Component({
  selector: 'app-root',
  template: `
    <p [style.background-color]="kolor" [style.text-align]="jak"> Nazywam się
{{osoba.imie}} {{osoba.nazwisko}}
        <ng-container [ngSwitch]="osoba.type">
            <img src="assets/images/wenus.png" *ngSwitchCase=1>
            <img src="assets/images/mars.png" *ngSwitchCase=2>
            <img src="assets/images/unnamed.png" *ngSwitchDefault>
        </ng-container>
    </p>
    <label><input type="checkbox" [(ngModel)]="pokazZdjecie">Pokaż zdjęcie </
label>
    <div id="ramka">
        <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <div id="kontakt" class="tekst">
        <p>Kontakt: </p>
        <ul>
            <li *ngFor="let value of osoba.adres"> {{value}} </li>
        </ul>
```

```
    </div>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <ng-template #brakZdjecia>
        <img src="{{osoba.bezZdjecia}}" class="zdjecie">
        <p class="tekst_srodek">Brak zdjęcia</p>
    </ng-template>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg",
    bezZdjecia: "assets/images/nofoto.jpg",
    adres: [
      "Piękna 16",
      "40-003",
      "Katowice"
    ],
    type: Plec.niezdefiniowana
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Listing 2.74. Wyliczenie (enum) Plec

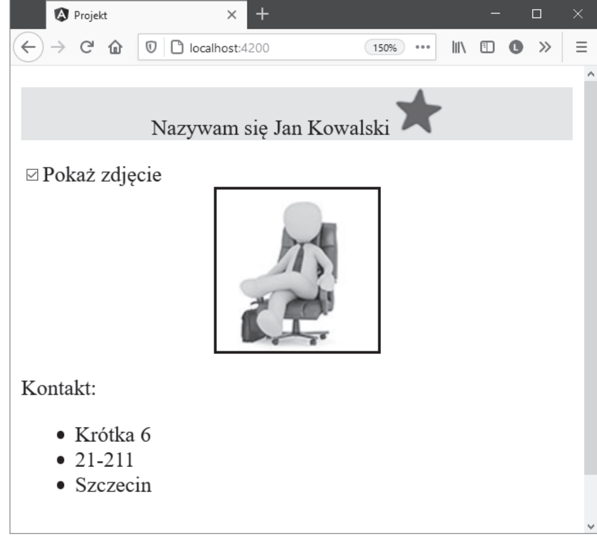
```
export interface Osoba {
    imie: string;
    nazwisko: string;
    zdjecie: string;
    bezZdjecia: string;
    adres: string[];
    type: Plec;
}

export enum Plec {
    kobieta = 1,
    mezczyzna,
    niezdefiniowana
}
```

Rysunek 2.55 pokazuje, jak zmienia się okno przeglądarki w zależności od definicji płci.



Rysunek 2.55. Działanie dyrektywy ngSwitch — dopasowanie. Po lewej — mężczyzna, po prawej — kobieta

W Angularze dozwolone jest posługiwanie się instrukcją `ngSwitchDefault`. Taki domyślny element zostanie wyświetlony, jeśli **nie zostanie znalezione żadne dopasowanie — wartość domyślna**. W naszym przykładzie, gdy właściwość `type` przyjmie wartość `Plec.niezdefiniowana`, zostanie wyświetlona gwiazdka (w wyliczeniu pole `niezdefiniowana` znajduje się na trzeciej pozycji, dlatego jego wartość to `3`, co oznacza brak dopasowania z instrukcją `ngSwitchCase` (rysunek 2.56).



Rysunek 2.56. Działanie dyrektywy ngSwitch — brak dopasowania (wartość domyślna)

WSKAZÓWKA

Aby wraz z instrukcją `ngSwitchCase` zamiast wartości numerycznych móc zastosować bardziej czytelny zapis, należy w komponencie dodać nowe pole `Plec`, które następnie trzeba powiązać z wyliczeniem. Wpis `Plec=Plec;` pozwoli zastąpić wartości liczbowe. Wartość `1` będzie można zamienić na `Plec.kobieta`, a wartość `2` — na `Plec.mezczyzna`.

2.4.2. Dyrektywy atrybutowe

Dyrektywy tego typu są używane do **zmiany wyglądu i zachowania elementów drzewa dokumentów**.

Dyrektywa ngStyle

Dyrektywa `ngStyle` służy do ustawienia stylu formatowanego elementu.

Listing 2.75 przedstawia kod, który zmienia rozmiar tekstu nagłówka `h1` oraz kolor tła po każdym wciśnięciu przycisku `Zmień styl` (wywołanie zdarzenia `click()`).

Listing 2.75. Dyrektywa ngStyle — zawartość pliku app.component.html

```
<button type="button" (click)="zmiana()">Zmień styl</button>
<h1 [ngStyle]="{'font-size.%': wybor*20+100,
'color': 'white',
'background-color': kolory[wybor] }"> Treść nagłówka</h1>
```

Oczywiście aby zdarzenie mogło zaistnieć, jego definicja musi zostać zapisana w pliku *app.component.ts*.

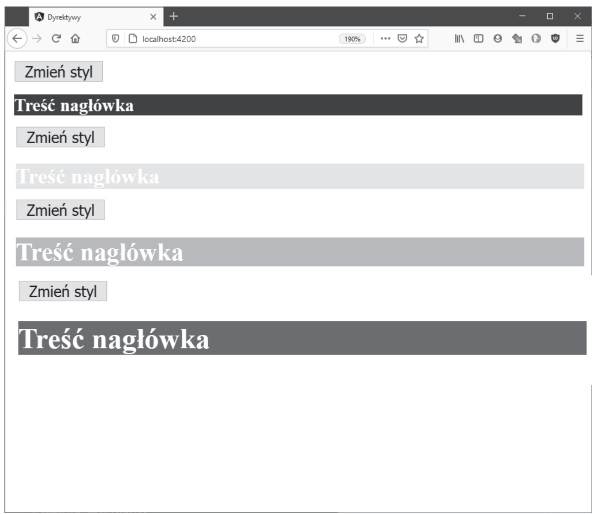
Kolor tła jest pobierany z tablicy `kolory`, a za jego wybór odpowiada zmienna `wybor`. Zmienna ta wpływa również na rozmiar wyświetlanej czcionki (listing 2.76).

Listing 2.76. Dyrektywa ngStyle — zawartość pliku app.component.ts

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

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

  kolory: string[] = ["blue", "yellow", "orange", "red"];
  wybor: number = 0;
```

```
  zmiana(): void {
    this.wybor++;
    if (this.wybor >= 4) this.wybor = 0;
  }
}
```

Wynik działania kodu jest pokazany na rysunku 2.57 — każdorazowe wciśnięcie przycisku powoduje zmianę selektorów `font-size` i `background-color` klasy.



Rysunek 2.57. Działanie dyrektywy ngStyle

Dyrektywa ngClass

Dyrektywa **`ngClass`** odpowiada za ustawienie klasy elementu.

Aby poznać sposób jej działania, prześledźmy poniższy przykład. Rozpoczynamy od zdefiniowania pliku stylu.

Plik zawiera definicję trzech stylów: `a`, `b` oraz `c` (listing 2.77).

Listing 2.77. Definicja pliku stylów CSS

```
.a {
    color: blue;
}
.b {
    font-size: 20px;
    text-decoration: underline;
}
.c {
    background-color: black;
}
```

W pliku *app.component.ts* zdefiniowano tablicę `osoby`.

```
osoby = ['Jan Kowalski', 'Tadeusz Nowak', 'Beata Tryla']
```

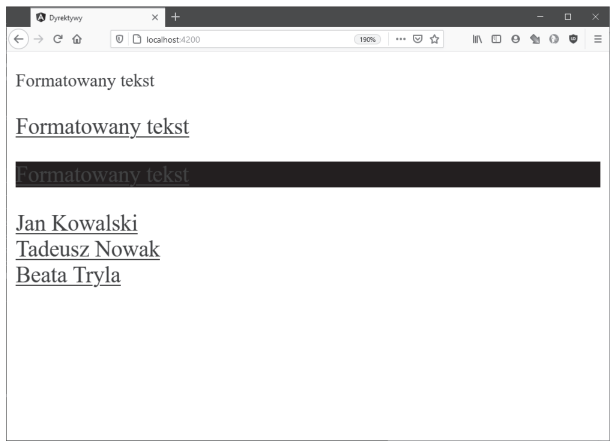
Za wyświetlenie sformatowanego tekstu odpowiada kod znajdujący się w pliku HTML (listing 2.78). Poszczególnym akapitom zostają przypisane klasy.

Przedstawiona w poprzednim rozdziale dyrektywa `ngFor` dodatkowo wyświetla i formatuje wszystkie elementy tablicy `osoby`.

Listing 2.78. Działanie dyrektywy ngClass

```
<p [ngClass]="'a'">Formatowany tekst</p>
<p [ngClass]="'a b'">Formatowany tekst</p>
<p [ngClass]="'a b c'">Formatowany tekst</p>

<div *ngFor="let osoba of osoby" [ngClass]="'a b'">{{osoba}}</div>
```

Wynik działania kodu jest pokazany na rysunku 2.58.



Rysunek 2.58. Działanie dyrektywy ngClass

Pytania kontrolne

1. Jaką rolę odgrywa dyrektywa?
2. Wskaż różnice i podobieństwa między omówionymi dyrektywami strukturalnymi a ich pierwowzorami: funkcją warunkową, pętlą oraz przełącznikiem. Czy różnią się w sposobie działania?
3. Podaj sposoby wykorzystania dyrektyw atrybutowych.

2.5. Komponenty

2.5.1. Wstęp do komponentów

Komponenty w Angularze służą do budowania aplikacji — są jej składowymi. Budowę aplikacji w tym frameworku rozpoczynamy od pojedynczego komponentu; kolejne tworzą **drzewo komponentów**.

Przyjrzyjmy się przykładowi z listingu 2.79 (ma on dużo cech wspólnych z poprzednim) i dalszym — 2.80 oraz 2.81. Będą one wstępem do komponentów.

Listing 2.79. Samochód marzeń — plik app.component.ts

```
import { Component } from '@angular/core';
import { Rodzaj, Samochod } from './interfejs_samochod';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  samochody: Samochod[] =
    [
      {
        marka: "McLaren",
        model: "720S",
        zdjecie: "assets/images/mclaren.jpg",
        dane: [
          "do setki 3,5 s",
          "cena  0,5 mln euro",
        ],
        type: Rodzaj.sportowy
      },
      {
        marka: "Hummer",
        model: "H5",
        zdjecie: "assets/images/hummer.jpg",
        dane: [
          "do setki 18 s",
          "cena 250 000 euro",
        ],
        type: Rodzaj.terenowy
      },
      {
```

```
        marka: "Mercedes",
        model: "Vario Mobil Signature",
        zdjecie: "assets/images/vario.jpg",
        dane: [
          "do setki 32 s",
          "cena 0,8 mln euro",
        ],
        type: Rodzaj.kamper
      }
    ]

  samochod: Samochod = this.samochody[0];
  kolorb: string = "white";
  kolort: string = "black";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolorb = this.kolorb === "white" ? "black" : "white";
    this.kolort = this.kolort === "black" ? "white" : "black";
  }

  Rodzaj = Rodzaj;

  constructor() {
  }
}
```

Listing 2.80. Samochód marzeń — plik app.component.html

```
<div id="contener">
    <div id="baner" [style.background-color]="kolorb" [style.
color]="kolort">Wybierz samochód marzeń:
        <select [(ngModel)]=samochod>
            <option [ngValue]="null">Brak wyboru</option>
            <option *ngFor="let value of samochody" [ngValue]="value">{{ value.
marka }} {{value.model}}</option>
        </select>
        <button (click)="zmienKolor()" type="button" [class.przycisk]="aktywna"
[style.color]="kolort">Zmień kolor
            tła</button>
    </div>
    <div *ngIf="samochod">
        <div id="panel_lewy">
            <label class="check"><input type="checkbox"
[(ngModel)]=pokazZdjecie>Pokaż zdjęcie </label>
```

```
        </div>
        <div id="panel_srodkowy">
            <p class="akapit">{{samochod.marka}} {{samochod.model}}
                <ng-template #brakZdjecia>
                    <p class=center>Brak zdjęcia</p>
                </ng-template>
            </p>
            <br>
            <img class=center src={{samochod.zdjecie}} [class.
zdjecie]="aktywna" *ngIf="pokazZdjecie else brakZdjecia">
        </div>
        <div id="panel_prawy">
            <ng-container [ngSwitch]="samochod.type">
                <img src="assets/images/sport.png" *ngSwitchCase=Rodzaj.
sportowy>
                <img src="assets/images/suv.png" *ngSwitchCase=Rodzaj.terenowy>
                <img src="assets/images/camper.png" *ngSwitchCase=Rodzaj.
kamper>
            </ng-container>
            <p> Dane: </p>
            <ul>
                <li *ngFor="let item of samochod.dane"> {{item}}
            </ul>
        </div>
    </div>
</div>
```

Listing 2.81. Samochód marzeń — plik interfejs_samochod.ts

```
export interface Samochod {
    marka: string;
    model: string;
    zdjecie: string;
    dane: string[];
    type: Rodzaj;
}
export enum Rodzaj {
    sportowy = 1,
    terenowy,
    kamper
}
```

Na listingu 2.79 zdefiniowano tablicę obiektów `samochody` (podobnie jak na listingu 2.44), którą połączono z interfejsem `Samochod` przedstawionym na listingu 2.81. Każdy z obiektów zawiera pola: `marka`, `model`, `zdjecie`, `dane` (tablica z dwiema wartościami) oraz `type` (wyliczenie o nazwie `Rodzaj`). W pliku tym znajdują się również właściwości, które sterują działaniem funkcji i dyrektyw.

Plik szablonu to osobny plik o nazwie *app.component.html*. Jego zawartość jest pokazana na listingu 2.80.

W polu `<div>` o identyfikatorze `baner` znajduje się dyrektywa `ngFor`, która pobiera wartości z tablicy `samochody` i przedstawia je w postaci rozwijanej listy (znacznik `<option>`). Do listy opcji dodano także pole `Brak wyboru` oraz metodę `zmienKolor()`, która odpowiada za zmianę tła banera.

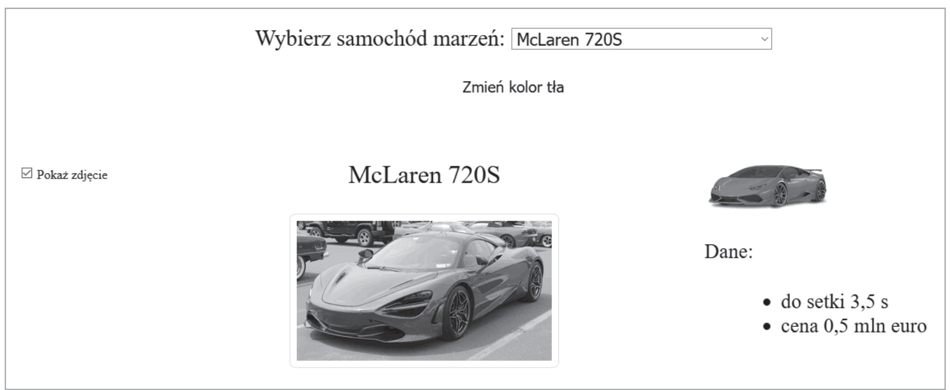
Ten obszar stron przedstawiają dwa rysunki, 2.59 i 2.60. Pierwszy pokazuje aktywne pole *Brak wyboru* po zmianie domyślnego tła.

Rysunek 2.59. Zmiana koloru tła banera

Drugi pokazuje dostępne opcje.

Rysunek 2.60. Rozwijana lista z wyborem samochodów

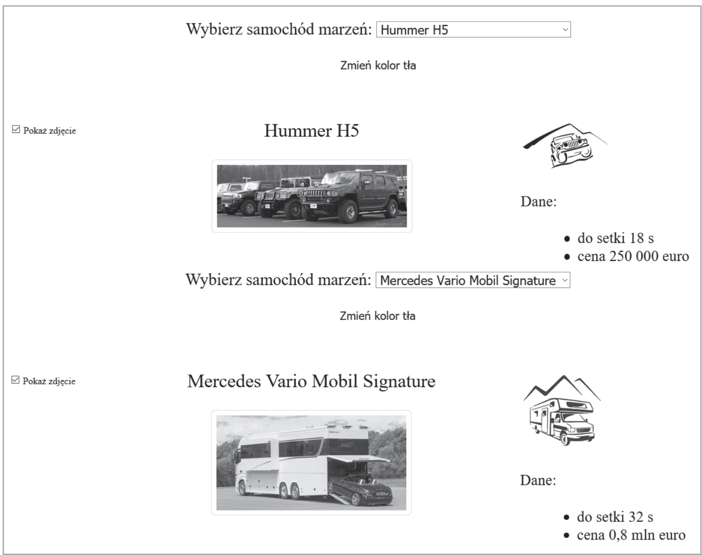
Wybór opcji spowoduje wyświetlenie poniżej (rysunek 2.61) dodatkowych szczegółów w postaci **krótkiego opisu** (przyspieszenie oraz cena), **zdjęcia** (włączenie odbywa się po zaznaczeniu pola checkbox *Pokaż zdjęcie*).



Rysunek 2.61. Wybór opcji

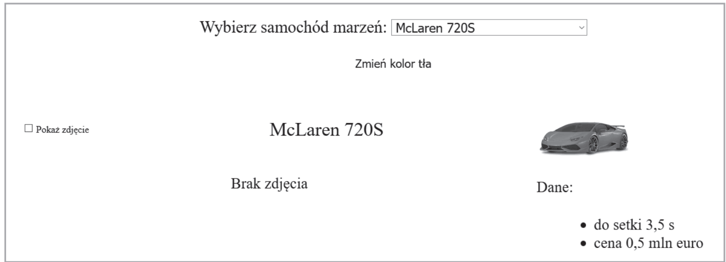
W zależności od zdefiniowanej kategorii samochodu nad polem *Dane* pojawia się powiązane z kategorią **zdjęcie**.

Wszystkie pozostałe wybory są pokazane na rysunku 2.62.



Rysunek 2.62. Zmiana zawartości widoku w zależności od wybranej opcji

W sekcji `div` o identyfikatorze `panel_lewy` znajduje się pole `checkbox`. Dzięki użyciu dyrektywy `ngIf` w momencie zaznaczenia go zostaje przekazana wartość `true` i wyświetlane jest zdjęcie wybranego samochodu w `panel_srodkowy`. Nieaktywne pole wywołuje instrukcję `else`, która jest połączona ze zmienną szablonową `brakZdjecia` — zostaje wyświetlony napis *Brak zdjęcia* (rysunek 2.63).



Rysunek 2.63. Działanie pola checkbox

Nad sekcją `div` o identyfikatorze `panel_lewy` znajduje się jeszcze jeden nienazwany `div` z jedną instrukcją, `*ngIf="samochod"`. Dyrektywa powoduje niewczytanie strony w momencie wybrania opcji *Brak wyboru*.

Prawy panel wyświetla dane o wybranym samochodzie oraz miniaturowe zdjęcie. Wyborem miniaturki steruje dyrektywa `ngSwitch`.

Zanim jednak wykonamy dalsze czynności, musisz się dowiedzieć, jak zbudowany jest komponent i jak można przekazywać wartości w ramach różnych komponentów.

2.5.2. Budowa komponentu i zagnieżdżanie

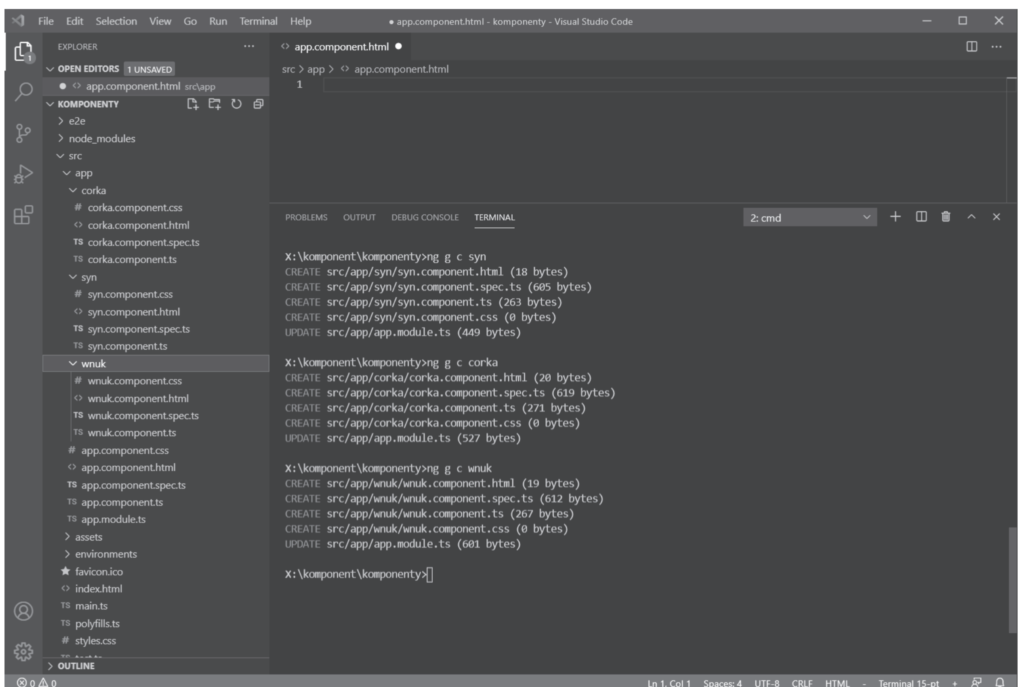
Każdy komponent ma klasę, plik HTML i plik stylów. Aby wygenerować komponent, należy się posłużyć poleceniem **ng generate component <nazwa_komponentu>**. Dozwolona jest skrócona forma: **ng g c <nazwa_komponentu>**.

Spróbujmy utworzyć i połączyć ze sobą cztery komponenty o nazwach:

- komponent główny aplikacji (rodzic) — app-root,
  - » komponent syn,
  - » komponent corka,
    - – komponent wnuk.

Komponenty o nazwie syn i corka znajdują się na jednym poziomie i są umiejscowione bezpośrednio w komponencie rodzica, natomiast komponent wnuk jest zagnieżdżony w komponencie corka.

Rysunek 2.64 przedstawia proces generowania komponentów. Jak można zaobserwować, dla każdego z nich zostają wygenerowane cztery pliki: *<nazwa_komponentu>.component.html* to plik szablonu, *<nazwa_komponentu>.component.css* to arkusz stylów, *<nazwa_komponentu>.component.ts* to plik TypeScript, w którym znajdują się zmienne oraz metody, a plik *<nazwa_komponentu>.component.spec.ts* odpowiada za testy.



Rysunek 2.64. Wygenerowanie komponentów

Aby można było używać wygenerowanych komponentów, należy je zainicjować w module aplikacji (rysunek 2.65). Odpowiednie wpisy są umieszczone w pliku *app.module.ts* (podczas tworzenia komponentu plik ten zostanie przez Angular zaktualizowany, co widać na rysunku 2.64, na którym ostatnią linijką jest komunikat `UPDATE src/app/app.module.ts`).

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'

import { AppComponent } from './app.component';
import { SynComponent } from './syn/syn.component';
import { CorkaComponent } from './corka/corka.component';
import { WnukComponent } from './wnuk/wnuk.component';


@NgModule({
  declarations: [
    AppComponent,
    SynComponent,
    CorkaComponent,
    WnukComponent,
```

Rysunek 2.65. Zawartość pliku app.module.ts

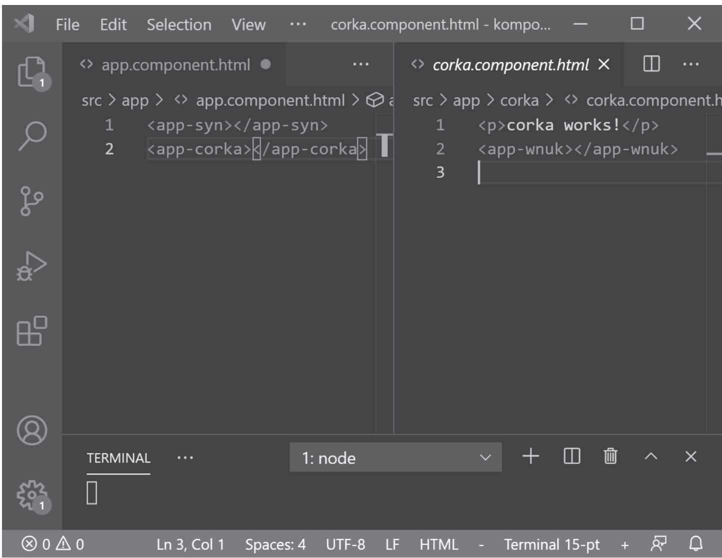
Utworzenie nowego projektu powoduje wygenerowanie komponentu — na samym początku tworzona aplikacja jest złożona z pojedynczego komponentu, będącego rodzicem całej struktury (korzeniem). Każdy nowo powstały komponent jest połączony z korzeniem bezpośrednio lub poprzez inny, pośredni komponent.

Aby rozpocząć zagnieżdżanie komponentu, w pierwszej kolejności należy poznać jego nazwę. Odnajdziesz ją w pliku *<nazwa_komponentu>.component.ts* w polu o nazwie `selector` (rysunek 2.66).

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

@Component({
  selector: 'app-syn',
  templateUrl: './syn.component.html',
  styleUrls: ['./syn.component.css']
})
export class SynComponent implements OnInit {
  @Input()  zmiennaImport!: string;

  constructor() { }

  ngOnInit(): void {
  }

}
```

Rysunek 2.66. Nazwa komponentu syn — plik syn.component.ts.

Gdy już dysponujemy nazwą, musimy otworzyć plik rodzica *app.component.html* i umieścić w nim tag HTML, który jest nazwą zagnieżdżanego komponentu (lewy panel na rysunku 2.67). W opisywanym przykładzie komponent `wnuk` jest zagnieżdżony w komponencie `corka`, dlatego jego tag znajduje się w pliku HTML komponentu `corka` (prawy panel).



Rysunek 2.67. Zagnieżdżenie komponentów

Ponieważ domyślnie w każdym pliku HTML nowo utworzonego komponentu znajduje się akapit z jego nazwą i napisem *works!*, efektem zagnieżdżenia jest wyświetlenie strony pokazanej na rysunku 2.68.

Rysunek 2.68. Efekt zagnieżdżenia komponentów

2.5.3. Przesyłanie danych pomiędzy komponentami

Z komponentu nadrzędnego do podrzędnego (@Input)

Pomiędzy wygenerowanymi komponentami możemy przesyłać informacje. Przyjrzyjmy się, jak przesłać wartości zmiennych zdefiniowane w komponencie nadrzędnym (rodzicu) do komponentu podrzędnego (np. `syn`).

Rozpoczynamy od utworzenia w pliku *app.component.ts* zmiennych, które będą przekazywane do elementu podrzędnego. Została utworzona tablica o nazwie `zmiennaEksport`, która ma dwa pola (listing 2.82).

Listing 2.82. Eksportowanie zmiennych z komponentu

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'projekt';
  zmiennaEksport: string[] = ['17:00', 'pływalnią'];
}
```

Następnie w pliku *syn.component.ts* należy zastosować dekorator `@Input` i zapisać nazwę zmiennej (na listingu 2.83 użyto nazwy `zmiennaImport`), która ma przechowywać wartość przesłaną z elementu nadrzędnego. Aby móc użyć dekoratora `@Input`, trzeba zaimportować komponent `Input` z `@angular/core`.

Listing 2.83. Importowanie zmiennych do komponentu

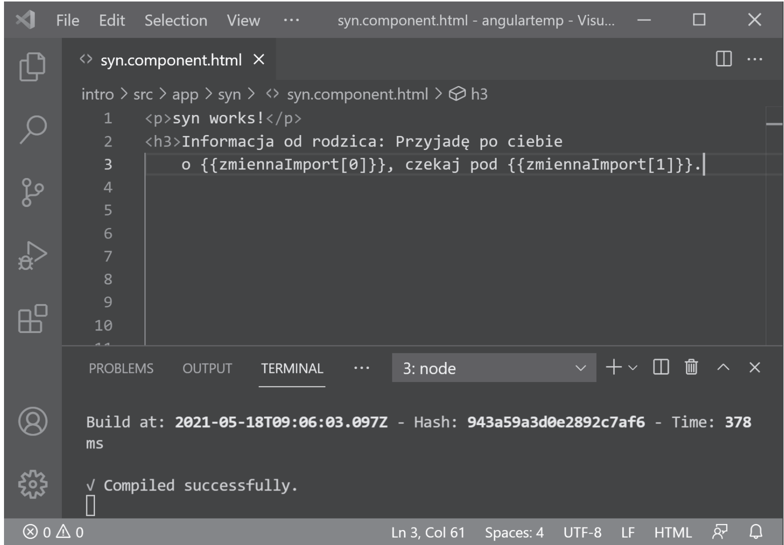
```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-syn',
  templateUrl: './syn.component.html',
  styleUrls: ['./syn.component.css']
})
export class SynComponent implements OnInit {
  @Input() zmiennaImport!: string[];

  constructor() { }

  ngOnInit(): void {
  }
}
```

Ostatnią czynnością jest modyfikacja pliku HTML, w którym zagnieździliśmy komponent (*app.component.html*) w wywołującym go znaczniku. Określamy, do jakiej zmiennej ma być przypisana eksportowana zmienna. Do zmiennej importowanej, `zmiennaImport`, zostanie przypisana wartość eksportowana, `zmiennaEksport` (listing 2.84).

Listing 2.84. Definicja zmiennych

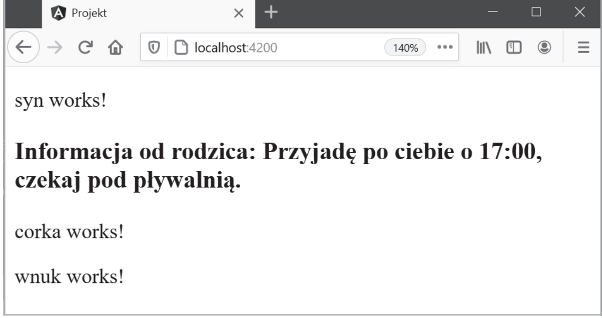
```
<app-syn [zmiennaImport]="zmiennaEksport"></app-syn>
<app-corka></app-corka>
```

Od tej pory w pliku HTML komponentu podrzędnego (*syn.component.html*) można się do zmiennej (`zmiennaImport`) odwoływać np. z użyciem interpolacji (rysunek 2.69).



Rysunek 2.69. Przekazanie wartości zmiennej z komponentu nadrzędnego do podrzędnego

Efektem przekazania jest wyświetlenie napisu: *Informacja od rodzica: Przyjadę po ciebie o 17:00, czekaj pod pływalnią.* (rysunek 2.70).



Rysunek 2.70. Efekt przekazania wartości zmiennej

Z komponentu podrzędnego do nadrzędnego (@Output)

Na powiadomienie wysłane od rodzica (komponent nadrzędny) odpowie `syn` (komponent podrzędny).

Wysyłanie informacji w drugą stronę jest bardziej skomplikowane, a to ze względu na konieczność utworzenia tzw. **event emittera**, który inicjujemy przez zaimportowanie komponentów `@Output` oraz `EventEmitter` z `@angular/core` w pliku *.ts* komponentu podrzędnego. W drugim kroku łączymy z dekoratorem `@Output` nowy obiekt `EventEmitter` (obiekt będzie przesyłał dane typu `string`) o nazwie `odpowiedz`. Aby dana zmienna/wartość mogła zostać wyeksportowana, musimy użyć metody `emit()`. Jako jej argument podajemy to, co chcemy wysłać (listing 2.85).

Listing 2.85. Przekazanie zmiennej z komponentu podrzędnego do nadrzędnego — plik syn.component.ts

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

@Component({
  selector: 'app-syn',
  templateUrl: './syn.component.html',
  styleUrls: ['./syn.component.css']
})
export class SynComponent implements OnInit {
  @Input() zmiennaImport!: string[];

  @Output() odpowiedz = new EventEmitter<string>();

  constructor() { }

  ngOnInit(): void { }

  wyslanie() {
    this.odpowiedz.emit('Będę czekał');
  }
}
```

Ponieważ wyeksportowanie zmiennej jest skutkiem zdarzenia, w pliku *syn.component.html* zostaje umieszczony przycisk *Wyślij odpowiedź do rodzica*, którego wciśnięcie wywoła zdarzenie `click` połączone z funkcją `wyslanie`, a tym samym wyemitowanie stringu `Będę czekał` (listing 2.86).

Listing 2.86. Wyeksportowanie wartości — plik syn.component.html

```
<p>syn works!</p>
<h3>Informacja od rodzica: Przyjadę po ciebie o {{zmiennaImport[0]}},
    czekaj pod {{zmiennaImport[1]}}.</h3>
<button (click)="wyslanie()" type="button" >Wyślij odpowiedź do rodzica</
button>
```

Aby przekazanie napisu do komponentu znajdującego się w hierarchii wyżej doszło do skutku, podobnie jak w przypadku zastosowania `@Input` należy zmodyfikować plik HTML, w którym jest użyty tag komponentu. Pole `odpowiedz` łączymy z metodą obsługującą zdarzenie o nazwie `onOdpowiedz`. Jako argument metody zostaje podana zmienna `$event`, która zawiera przekazywaną wartość (listing 2.87).

Listing 2.87. Wyeksportowanie wartości — plik app.component.html

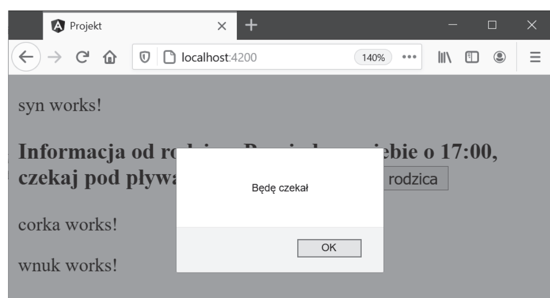
```
<app-syn [zmiennaImport]="zmiennaEksport" (odpowiedz)="onOdpowiedz($event)"></
app-syn>
<app-corka></app-corka>
```

Ostatnią czynnością jest obsłużenie metody `onOdpowiedz`. Jej definicja zostaje zapisana w pliku *app.component.ts* rodzica. Metodzie `onOdpowiedz` przekazywany jest jeden argument (pochodzący od `$event`) typu `string` o nazwie `otrzymane`. Ten finalnie zostaje połączony z wywołaniem funkcji `alert()` (listing 2.88).

Listing 2.88. Wyeksportowanie wartości — plik app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'projekt';
  zmiennaEksport: string[] = ['17:00', 'pływalnią'];

  onOdpowiedz(otrzymane: string) {
    alert(otrzymane);
  }
}
```

Końcowym efektem jest wyświetlenie po wciśnięciu przycisku *Wyślij odpowiedź do rodzica* napisu przekazanego z komponentu podrzędnego w oknie typu *alert*, którego wywołanie znajduje się w komponencie nadrzędnym (rysunek 2.71).



Rysunek 2.71. Efekt przekazania wartości zmiennej z komponentu podrzędnego do nadrzędnego

Uzbrojeni w nową wiedzę, możemy powrócić do przykładu zaprezentowanego na początku rozdziału. Naszym zadaniem będzie podzielenie kodu aplikacji tak, aby znalazł się on w **dwóch odrębnych komponentach**, oraz **zapewnienie komunikacji pomiędzy nimi**.

Rozpoczynamy od wygenerowania dwóch komponentów o nazwach: *glowny* i *szczegol*. W komponencie *glowny* umieszczony zostanie banner (rozwijana lista samochodów i przycisk zmieniający tło — listing 2.89), a w komponencie *szczegol* — informacje o samochodzie (przycisk *Pokaż zdjęcie*, zdjęcie, opis i miniatura — listing 2.90).

Komponent *glowny* jest zagnieżdżony w głównym drzewie komponentów, a komponent *szczegol* — w komponencie *glowny*.

Listing 2.89. Zawartość pliku glowny.component.html

```
<div id="baner" [style.background-color]="kolorb" [style.
color]="kolort">Wybierz samochód marzeń:
    <select [(ngModel)]="samochod">
        <option [ngValue]="null">Brak wyboru</option>
        <option *ngFor="let value of samochody" [ngValue]="value">{{ value.
marka }} {{value.model}}</option>
    </select>
    <button (click)="zmienKolor()" type="button" [class.przycisk]="aktywna"
[style.color]="kolort">Zmień kolor
        tła</button>
</div>
<app-szczegol [samochod]="samochod"></app-szczegol>
```

Listing 2.90. Zawartość pliku szczegol.component.html

```
<div *ngIf="samochod">
    <div id="panel_lewy">
        <label class="check"><input type="checkbox"
[(ngModel)]=pokazZdjecie>Pokaż zdjęcie </label>
    </div>
    <div id="panel_srodkowy">
        <p class="akapit">{{samochod.marka}} {{samochod.model}}
            <ng-template #brakZdjecia>
                <p class="center">Brak zdjęcia</p>
            </ng-template>
        </p>
        <br>
        <img class="center" src="{{samochod.zdjecie}}" [class.
zdjecie]="aktywna" *ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <div id="panel_prawy">
        <ng-container [ngSwitch]="samochod.type">
            <img src="assets/images/sport.png" *ngSwitchCase=Rodzaj.sportowy>
            <img src="assets/images/suv.png" *ngSwitchCase=Rodzaj.terenowy>
            <img src="assets/images/camper.png" *ngSwitchCase=Rodzaj.kamper>
```

```
        </ng-container>
        <p> Dane: </p>
        <ul>
            <li *ngFor="let item of samochod.dane"> {{item}}
        </ul>
    </div>
</div>
```

Tak samo należy podzielić plik *app.component.ts*. Właściwości i metody **muszą zostać przypisane do komponentów, które z nich korzystają** (listingi 2.91 i 2.92).

Listing 2.91. Zawartość pliku glowny.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Rodzaj, Samochod } from '../interfejs_samochod';

@Component({
  selector: 'app-glowny',
  templateUrl: './glowny.component.html',
  styleUrls: ['./glowny.component.css']
})
export class GlownyComponent implements OnInit {
  samochody: Samochod[] =
    [
      {
        marka: "McLaren",
        model: "720S",
        zdjecie: "assets/images/mclaren.jpg",
        dane: [
          "do setki 3,5 s",
          "cena  0,5 mln euro",
        ],
        type: Rodzaj.sportowy
      },
      {
        marka: "Hummer",
        model: "H5",
        zdjecie: "assets/images/hummer.jpg",
        dane: [
          "do setki 18 s",
          "cena 250 000 euro",
        ],
        type: Rodzaj.terenowy
      },
      {
        marka: "Mercedes",
        model: "Vario Mobil Signature",
        zdjecie: "assets/images/vario.jpg",
        dane: [
```

```
          "do setki 32 s",
          "cena 0,8 mln euro",
        ],
        type: Rodzaj.kamper
      }
    ]

  samochod: Samochod = this.samochody[0];
  kolorb: string = "white";
  kolort: string = "black";
  aktywna: boolean = true;

  Rodzaj = Rodzaj;

  zmienKolor() {
    this.kolorb = this.kolorb === "white" ? "black" : "white";
    this.kolort = this.kolort === "black" ? "white" : "black";
  }

  constructor() { }

  ngOnInit(): void {
  }
}
```

Listing 2.92. Zawartość pliku szczegol.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { Samochod, Rodzaj } from '../interfejs_samochod';

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

  @Input() samochod!: Samochod;

  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  Rodzaj = Rodzaj;

  constructor() { }

  ngOnInit(): void {
  }
}
```

Aby nasza aplikacja zaczęła działać, komponent *glowny* musi poinformować komponent *szczegol*, jaki samochód został wybrany. Dlatego na listingu 2.89 ostatnia linijka (`<app-szczegol [samochod]="samochod"></app-szczegol>`) łączy oba komponenty ze sobą. Dodatkowo następuje przekazanie wartości zmiennej `samochod` (tego, co zostało wybrane z listy) do zmiennej o takiej samej nazwie. Ponieważ zmienna `samochod` jest importowana, jej użycie wymaga zastosowania deklaracji za pomocą dekoratora `@Input` (listing 2.92).

Plik komponentu rodzica, czyli *app.component.ts*, pozostaje bez zmian, a w pliku HTML komponentu (*app.component.html*) musi się znaleźć kod, który łączy komponent rodzica z komponentem *glowny*, dlatego dodajemy `<app-glowny></app-glowny>`.

Po wykonaniu podziału aplikacja działa tak, jakby jej kod znajdował się w pojedynczym komponencie, podczas gdy w rzeczywistości jest on podzielony na dwa połączone ze sobą komponenty.

Zadanie 2.22.

Na podstawie zadania 2.21 i omówionego przykładu „Samochód marzeń” stwórz aplikację pokazującą walory/ciekawostki miejscowości, w której mieszkasz — przedstaw trzy wybrane. Aplikację stwórz jako pojedynczy komponent.

Zadanie 2.23.

Działającą aplikację z zadania 2.22 podziel na dwa komponenty.

2.5.4. Kompozycja i cykl życia komponentu

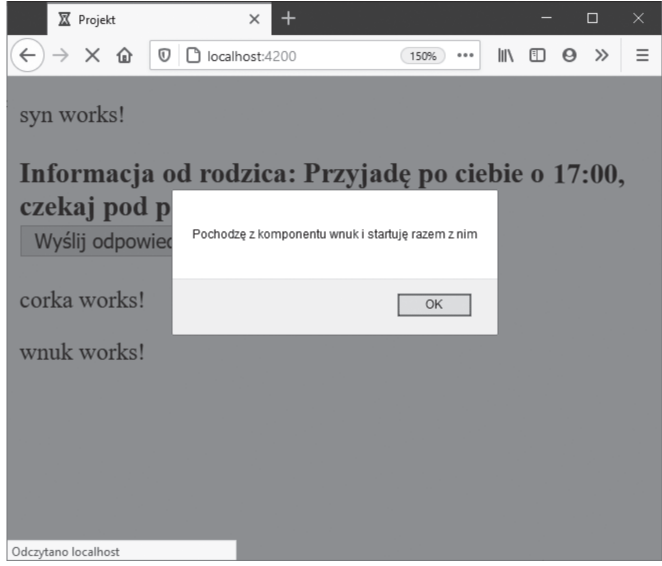
W nowo utworzonym komponencie znajduje się metoda `ngOnInit()`, która jest wywoływana podczas inicjalizacji komponentu. Aby zweryfikować działanie metody, dodajmy instrukcję, która wywoła metodę `alert`. Instrukcja deklaruje stałą o nazwie `wiadomosc`, której wartość zostaje przekazana do metody `alert` (listing 2.93) i wyświetlona w oknie dialogowym (rysunek 2.72).

Listing 2.93. Inicjalizacja zmiennej wraz z uruchomieniem komponentu — plik wnuk.component.ts

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

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

  constructor() { }
```

```
  ngOnInit(): void {
    const wiadomosc = "Pochodzę z komponentu wnuk i startuję razem z nim";
    alert(wiadomosc);
  }
}
```

Po uruchomieniu aplikacji możemy zobaczyć treść przekazanej wiadomości.



Rysunek 2.72. Inicjalizacja zmiennej wraz z uruchomieniem komponentu

Tak jak możemy inicjować zasoby, tak **możemy te zasoby zwalniać**.

W pliku *corka.component.ts* dodano komunikat, który zostanie użyty, gdy komponent zostanie odłączony od drzewa dokumentów.

```
ngOnDestroy(): void {
    console.log('Zostałem zniszczony');
}
```

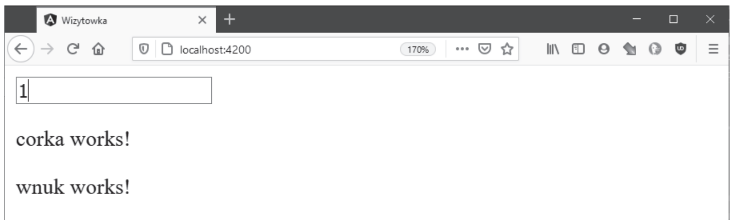
Przyłączenie i odłączenie komponentu realizuje dyrektywa `ngIf`. W tym celu w komponencie rodzica w pliku HTML (*app.component.html*) umieszczono kod:

```
<input [(ngModel)]="formularz" />
<app-corka [name]="formularz" *ngIf="formularz.length>0"></app-corka>
```

Należy również uaktualnić główny plik *app.component.ts* o deklarację zmiennej `formularz`:

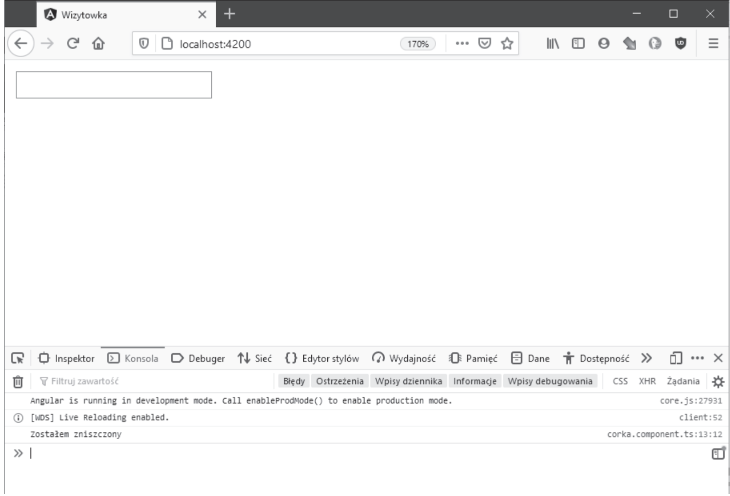
```
formularz: string = "";
```

Istnienie komponentu *corka* (a także *wnuk*, ponieważ ten został zagnieżdżony w *corka*) jest zależne od tego, czy w polu tekstowym zostanie wpisany jakikolwiek znak — jeśli tak, pojawią się akapity *corka works!* oraz *wnuk works!* (rysunek 2.73).



Rysunek 2.73. Podłączenie komponentów corka i wnuk do drzewa dokumentów

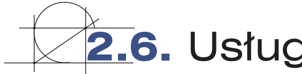
Brak znaku sprawia, że dyrektywa `ngIf` zwraca `false`, co w konsekwencji powoduje odłączenie komponentu `corka`. Ponieważ w pliku komponentu zdefiniowaliśmy metodę `ngOnDestroy`, w konsoli pojawia się komunikat `Zostałem zniszczony` (rysunek 2.74).



Rysunek 2.74. Odłączenie komponentu

Pytania kontrolne

1. Czym jest komponent? Wskaż jego przeznaczenie.
2. Czy są ograniczenia w liczbie użytych komponentów?
3. Po co się stosuje zagnieżdżanie komponentów?
4. W jaki sposób można przesyłać dane pomiędzy komponentami?
5. W jakim celu odłącza się komponent od drzewa dokumentów?



2.6. Usługi

2.6.1. Poznajemy usługę

Usługa to obiekt zapewniający pewną funkcjonalność. To oznacza, że jeśli mamy stały blok kodu wykonujący określone zadanie, a to jest wywoływane w różnych miejscach, możemy je zdefiniować jako usługę. Dzięki temu kod nie jest powielany.

2.6.2. Implementacja usługi

Podobnie jak w przypadku komponentów, rozpoczynamy od wygenerowania usługi. Aby utworzyć usługę, należy wydać polecenie **`ng generate service <nazwa_usługi>`**. Można także użyć skrótu **`n g s <nazwa_usługi>`**. Po wydaniu polecenia do projektu zostaną dodane dwa nowe pliki: *<nazwa_usługi>.service.spec.ts* oraz *<nazwa_usługi>.service.ts* (rysunek 2.75).

Nasza nowo utworzona usługa będzie się nazywać *Kalkulator*, a jej zadaniem będzie dodanie do siebie przekazanych wartości.

```
X:\intro\projekt>ng generate service Kalkulator
CREATE src/app/kalkulator.service.spec.ts (377 bytes)
CREATE src/app/kalkulator.service.ts (139 bytes)

X:\intro\projekt>
```

Rysunek 2.75. Utworzenie usługi

Aby móc odwołać się do usługi z poziomu innych komponentów, należy jej nazwę umieścić w tabeli *providers* (nazwę usługi poznasz po przejściu do pliku *<nazwa_usługi>.service.ts*) oraz zaimportować ją za pomocą znanego Ci już polecenia `import`. Wszystkie te czynności przeprowadzamy po otwarciu pliku *app.module.ts* (rysunek 2.76).

Powiązanie usługi z konstruktorem sprawi, że będziemy mogli się do niej odwoływać (listing 2.94).

Listing 2.94. Zawartość pliku app.component.ts

```
import { Component } from '@angular/core';
import { KalkulatorService } from './kalkulator.service';

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

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { KalkulatorService } from './kalkulator.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [KalkulatorService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Rysunek 2.76. Definicja usługi w pliku app.module.ts

Po połączeniu komponentu z usługą należy zdefiniować metodę usługi, czyli to, co ma ona wykonać. Zostaje zdefiniowana metoda `dodaj` (listing 2.95). Dzięki użyciu parametru resztowego (zobacz listing 2.31) możemy do niej przekazać dowolną liczbę argumentów.

Listing 2.95. Zawartość pliku kalkulator.service.ts

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

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

  constructor() { }

  dodaj(...liczby: number[]): number {
    let wynik = 0;
    for (let wartosci of liczby) {
      wynik += wartosci;
    }
    return wynik;
  }
}
```

Po wykonaniu wszystkich przypisań możemy sprawdzić, czy usługa działa. W tym celu w pliku **.ts* komponentu zdefiniowano właściwość `dodawanie` typu `number`, a następnie umieszczono odwołanie do usługi wraz z argumentami, które zostaną do siebie dodane (listing 2.96).

***Listing 2.96.** Użycie metody — app.component.ts*

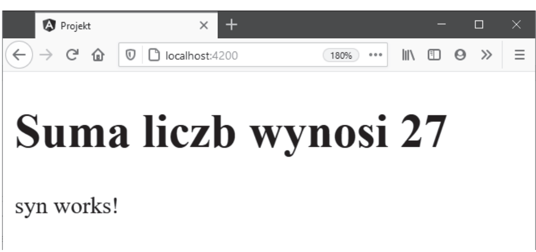
```
import { Component } from '@angular/core';
import { KalkulatorService } from './kalkulator.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [KalkulatorService]
})
export class AppComponent {
  title = 'projekt';
  dodawanie: number = 0;
  constructor(kalkulator: KalkulatorService) {
    this.dodawanie = kalkulator.dodaj(3, 6, 7, 9, 2);
  }
}
```

Ostatnią czynnością jest wyświetlenie wyniku (listing 2.97).

***Listing 2.97.** Wyświetlenie wyniku w szablonie*

```
<h1>
  Suma liczb wynosi {{dodawanie}}
</h1>
```

Efekt wprowadzonych zmian jest pokazany na rysunku 2.77.



Rysunek 2.77. Działanie usługi KalkulatorService

Na początku zostało powiedziane, że z usługi może korzystać wiele komponentów — sprawdźmy, czy faktycznie tak jest.

Wystarczy, że w pliku kolejnego komponentu, np. *syn.komponent.ts*, wykonamy odwołanie do usługi, aby zwróciła pożądany wynik (listing 2.98).

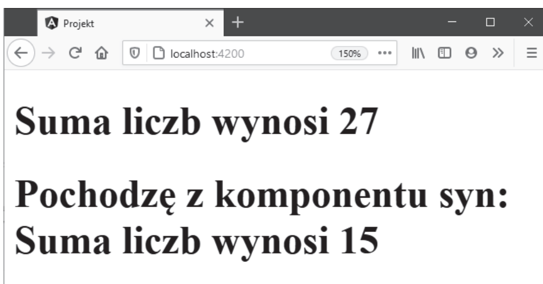
Listing 2.98. Implementacja usługi KalkulatorService w kolejnym komponencie

```
import { Component, OnInit } from '@angular/core';
import { KalkulatorService } from '../kalkulator.service';

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

  dodawanie: number = 0;
  constructor(kalkulator: KalkulatorService) {
    this.dodawanie = kalkulator.dodaj(1, 2, 3, 4, 5);
  }

  ngOnInit(): void {
  }
}
```

Oczywiście należy również uaktualnić plik szablonu komponentu (*syn.component.html*).

```
<h1>
    Pochodzę z komponentu syn: Suma liczb wynosi {{dodawanie}}
</h1>
```

Rezultat wykonania odwołania w pliku HTML jest pokazany na rysunku 2.78.



Rysunek 2.78. Użycie usługi KalkulatorService w kolejnym komponencie

Zadanie 2.24.

Rozszerz funkcjonalność przedstawionej usługi *KalkulatorService* o kolejne działania: odejmowanie, mnożenie, dzielenie.

Pytania kontrolne

1. Do czego można wykorzystać usługę?
2. Usługa kontra funkcja — wskaż podobieństwa i różnice.
3. Kto może korzystać z usługi?

2.7. Zdarzenia i formularze

Angular zapewnia także obsługę i walidację formularzy.

2.7.1. Przygotowanie formularza

Aby móc rozpocząć pracę z formularzami, należy dodać *FormsModule*. Import modułu przeprowadzamy w pliku *app.module.ts* (rysunek 2.79).

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { FormsModule} from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Rysunek 2.79. Dodanie FormsModule

Następnym krokiem jest utworzenie formularza. Do jego budowy wykorzystujemy standardowe znaczniki języka HTML.

Kod formularza jest pokazany na listingu 2.99. Został umieszczony w szablonie widoku (plik *app.component.html*). Tworząc go, należy dopilnować, aby jego pola zostały nazwane, w przeciwnym razie wystąpią błędy kompilacji.

Listing 2.99. Kod HTML formularza

```
<form>
  <div class="formularz">
    <h1>Zamówienie:</h1>
    <label>
      <span>Imię i nazwisko:</span>
      <input type="text" class="wpis" id="imieinazwisko" name="imieinazwisko">
```

```
      </label>
      <label>
        <span>Email:</span>
        <input type="text" class="wpis" id="email" name="email">
      </label>
      <label>
        <span>Produkt:</span>
        <select id="produkt" class="produkt" name="produkt">
          <option value="pralka">pralka</option>
          <option value="telewizor">telewizor</option>
          <option value="lodowka">lodówka</option>
          <option value="laptop">laptop</option>
        </select>
      </label>
      <label>
        <span>Ilość:</span>
        <input type="number" class="wpis" id="ilosc" name="ilosc">
      </label>
      <label>
        <span>Wiadomość:</span>
        <textarea class="wiadomosc" id="wiadomosc" name="wiadomosc"></textarea>
        <button type="submit" class="przycisk">Zamów</button>
      </label>
    </div>
</form>
```

Formularz został sformatowany za pomocą arkusza stylów, pokazanego na listingu 2.100 (plik *app.component.css*).

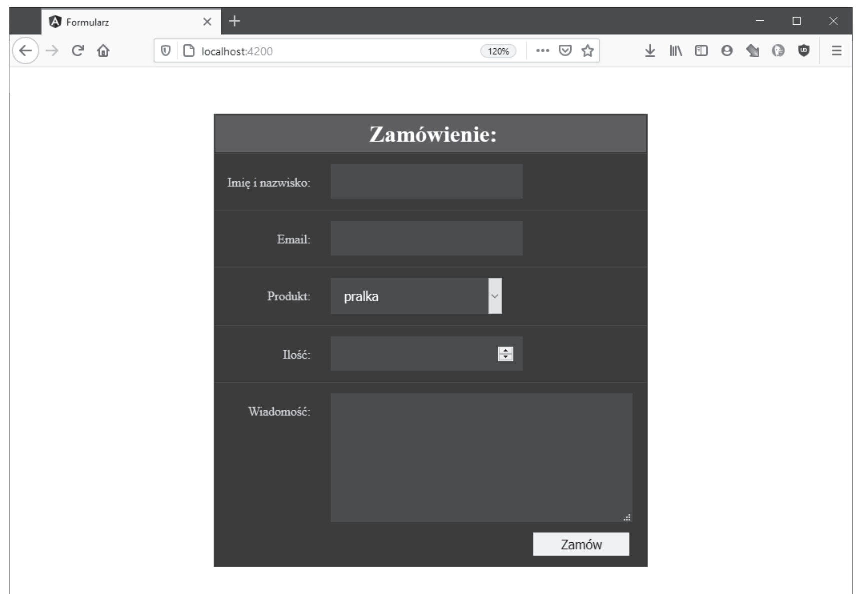
Listing 2.100. Arkusz stylów formularza

```
* {
    margin: 0;
    padding: 0;
}

body {
    font: 100% normal Arial, Helvetica, sans-serif;
}

form, input, select, textarea {
    margin: 0;
    padding: 0;
    color: #ffffff;
}
```

```
div.formularz h1 {
    text-align: center;
    color: #ffffff;
    font-size: 24px;
    padding: 5px 0 5px 5px;
    border: 1px solid #161712;
}

div.formularz {
    margin: 0 auto;
    width: 450px;
    background: #3d3939;
    position: relative;
    top: 40px;
    border: 1px solid #262626;
}

div.formularz .wpis {
    padding: 10px 10px;
    width: 180px;
    background: #262626;
    border: 1px double #171717
}

div.formularz .produkt {
    padding: 10px 10px;
    width: 180px;
    background: #262626;
    border: 1px double #171717
}

div.formularz .wiadomosc {
    padding: 7px 7px;
    width: 300px;
    height: 120px;
    background: #262626;
    border: 1px double #171717;
    overflow: hidden;
}

div.formularz .przycisk {
    margin: 0 0 10px 0;
    padding: 4px 7px;
    border: 0px;
```

```
    position: relative;
    top: 10px;
    left: 332px;
    width: 100px;
}

div.formularz label {
    width: 100%;
    display: block;
    background: #1b1616;
    border-top: 1px solid #262626;
    border-bottom: 1px solid #161712;
    padding: 10px 0 10px 0;
}

div.formularz label span {
    display: block;
    color: #d0dbdf;
    font-size: 13px;
    float: left;
    width: 100px;
    text-align: right;
    padding: 12px 20px 0 0;
}
```

Wygląd i układ pól formularza jest pokazany na rysunku 2.80.



Rysunek 2.80. Wygląd formularza

2.7.2. Implementacja formularza

Za zarządzanie formularzem odpowiada **dyrektywa ngForm**. Jej zastosowanie pozwoli uzyskać dostęp do wartości wpisanych do pól formularza. Aby zainicjować dyrektywę, przypisujemy ją do zmiennej szablonowej o dowolnej nazwie. Z użyciem tej nazwy będziemy się odwoływać do formularza. Linijka ze znacznikiem `<form>` przyjmuje postać `<form #formularz="ngForm">`.

Aby móc zacząć rejestrować pola formularza, inicjujemy je za pomocą **dyrektywy ngModel**. Przypisanie wartości spowoduje wyświetlenie jej w polu formularza.

Aby sprawdzić działanie formularza, definiujemy zdarzenie `click` na przycisku `Zamów`. Jego kliknięcie spowoduje wypisanie tekstu `Wszystko działa` w konsoli przeglądarki.

Kod po zmianach jest pokazany na listingu 2.101.

Listing 2.101. Implementacja formularza

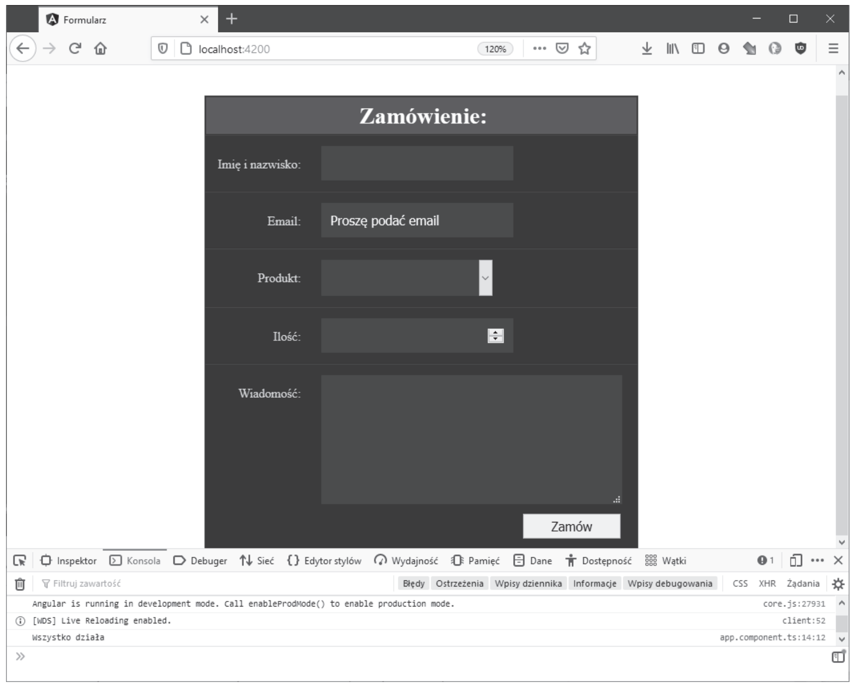
```
<form #formularz="ngForm">
  <div class="formularz">
    <h1>Zamówienie:</h1>
    <label>
      <span>Imię i nazwisko:</span>
      <input type="text" class="wpis" id="imieinazwisko" ngModel
name="imieinazwisko">
    </label>
    <label>
      <span>Email:</span>
      <input type="text" class="wpis" id="email" ngModel="Proszę podać email"
name="email">
    </label>
    <label>
      <span>Produkt:</span>
      <select id="produkt" class="produkt" ngModel name="produkt">
        <option value="pralka">pralka</option>
        <option value="telewizor">telewizor</option>
        <option value="lodowka">lodówka</option>
        <option value="laptop">laptop</option>
      </select>
    </label>
    <label>
      <span>Ilość:</span>
      <input type="number" class="wpis" id="ilosc" ngModel name="ilosc">
    </label>
    <label>
      <span>Wiadomość:</span>
```

```
      <textarea class="wiadomosc" id="wiadomosc" ngModel name="wiadomosc"></
textarea>
      <button type="submit" class="przycisk" (click)="onSubmit()">Zamów</
button>
    </label>
  </div>
</form>
```

Aby formularz działał, w pliku *app.component.ts*. musi zostać zdefiniowana metoda `onSubmit`. Należy dodać kod:

```
 onSubmit() {
    console.log("Wszystko działa");
  }
```

Po tych wszystkich zmianach w polu *Email* zostaje wypisana informacja *Proszę podać email*, a w konsoli, po kliknięciu przycisku *Zamów* — zdefiniowany tekst. Formularz działa poprawnie (rysunek 2.81).



Rysunek 2.81. Implementacja formularza

Aby rozpocząć przechwytywanie wartości wpisanych w polach formularza, należy **skorzystać z wiązania zdarzeń**, które definiujemy w polu `input` formularza i pliku *app.component.ts* komponentu.

Kod formularza jest pokazany na listingu 2.102, a zawartość pliku *app.component.ts* — na listingu 2.103.

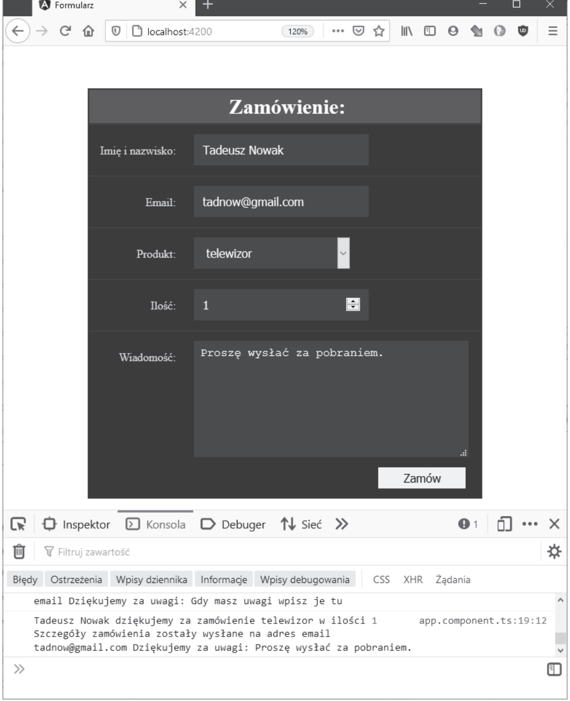
Listing 2.102. Rejestrowanie wartości wpisanych w polu formularza

```
<form #formularz="ngForm">
  <div class="formularz">
    <h1>Zamówienie:</h1>
    <label>
      <span>Imię i nazwisko:</span>
      <input type="text" class="wpis" id="imieinazwisko"
[(ngModel)]="imieinazwisko" name="imieinazwisko">
    </label>
    <label>
      <span>Email:</span>
      <input type="text" class="wpis" id="email" [(ngModel)]="email"
name="email">
    </label>
    <label>
      <span>Produkt:</span>
      <select id="produkt" class="produkt" [(ngModel)]="produkt"
name="produkt">
        <option value="pralka">pralka</option>
        <option value="telewizor">telewizor</option>
        <option value="lodowka">lodówka</option>
        <option value="laptop">laptop</option>
      </select>
    </label>
    <label>
      <span>Ilość:</span>
      <input type="number" class="wpis" id="ilosc" [(ngModel)]="ilosc"
name="ilosc">
    </label>
    <label>
      <span>Wiadomość:</span>
      <textarea class="wiadomosc" id="wiadomosc" [(ngModel)]="wiadomosc"
name="wiadomosc"></textarea>
      <button type="submit" class="przycisk" (click)="onSubmit()">Zamów</
button>
    </label>
  </div>
</form>
```

Listing 2.103. Obsługa formularza — plik app.component.ts

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

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

  imieinazwisko: string = "";
  email: string = "Proszę podać email";
  produkt: string = "";
  ilosc: number = 0;
  wiadomosc: string = "Jeśli masz uwagi, wpisz je tu";

  onSubmit() {
    console.log(this.imieinazwisko, 'dziękujemy za zamówienie', this.produkt,
'w ilości', this.ilosc, 'Szczegóły zamówienia zostały wysłane na adres email',
this.email, 'Dziękujemy za uwagi:', this.wiadomosc);
  }
}
```

Po uzupełnieniu wszystkich pól formularza ich wartość zostanie wyświetlona w konsoli (rysunek 2.82).

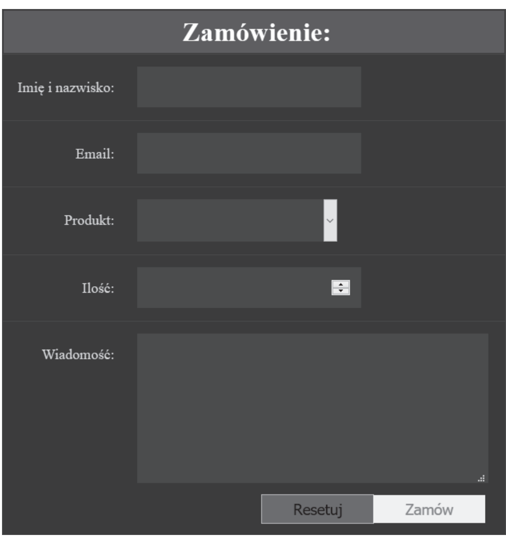


Rysunek 2.82. Wyświetlenie zawartości pól formularza

Aby do formularza dodać funkcjonalność resetowania, należy skorzystać z metody `resetForm()`. W tym celu wstawiamy nowy przycisk i przypisujemy do niego metodę (nie zapominamy o aktualizacji arkusza stylów — właściwość `left` klasy `div.formularz .przycisk` zmieniamy z `332px` np. na `220px`). Przykładowy kod realizujący to zadanie mógłby wyglądać tak:

```
<button type="button" class="przycisk" style="background-color: red;"
(click)="formularz.resetForm()">Resetuj</button>
```

Uaktualnienie kodu spowoduje dodanie czerwonego przycisku *Resetuj*, którego kliknięcie poskutkuje wyczyszczeniem wszystkich pól formularza (rysunek 2.83).



Rysunek 2.83. Resetowanie pól formularza

2.7.3. Walidacja formularzy

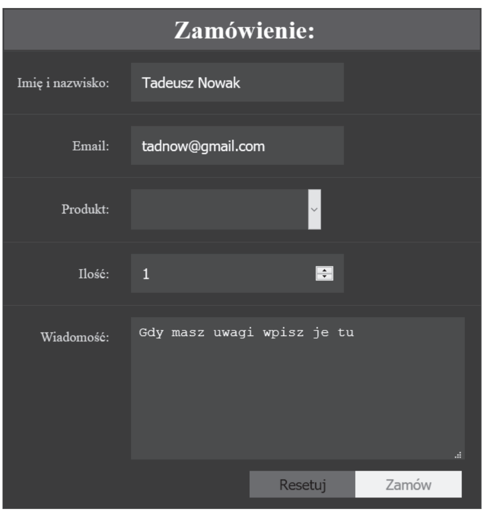
Formularze tworzone w Angularze możemy walidować. To oznacza, że mamy możliwość sprawdzenia zawartości pola formularza. Kod formularza weryfikujący zawartość pól jest pokazany na listingu 2.104.

Listing 2.104. Walidacja formularzy

```
<form #formularz="ngForm">
  <div class="formularz">
    <h1>Zamówienie:</h1>
    <label>
      <span>Imię i nazwisko:</span>
      <input type="text" class="wpis" id="imieinazwisko"
[(ngModel)]="imieinazwisko" name="imieinazwisko" required
#imieinazwiskoForm="ngModel">
      <div class="alert" *ngIf="imieinazwiskoForm.invalid && imieinazwiskoForm.
touched">Pole imię i nazwisko jest wymagane</div>
```

```
    </label>
    <label>
      <span>Email:</span>
      <input type="text" class="wpis" id="email" [(ngModel)]="email" 
name="email" required pattern=".+@.+" #emailForm="ngModel">
      <div class="alert" *ngIf="emailForm.invalid && emailForm.touched"> </div>
      <div class="alert" *ngIf="emailForm.touched && emailForm.errors?.
pattern">To nie jest email</div>
    </label>
    <label>
      <span>Produkt:</span>
      <select id="produkt" class="produkt" [(ngModel)]="produkt" name="produkt" 
required>
        <option value="pralka">pralka</option>
        <option value="telewizor">telewizor</option>
        <option value="lodowka">lodówka</option>
        <option value="laptop">laptop</option>
      </select>
    </label>
    <label>
      <span>Ilość:</span>
      <input type="number" class="wpis" id="ilosc" name="ilosc" 
[(ngModel)]="ilosc" min="1" max="5">
    </label>
    <label>
      <span>Wiadomość:</span>
      <textarea class="wiadomosc" id="wiadomosc" [(ngModel)]="wiadomosc" 
name="wiadomosc"></textarea>
      <button type="button" class="przycisk" style="background-color: red;" 
(click)="formularz.resetForm()">Resetuj</button>
      <button type="submit" class="przycisk" [disabled]="!formularz.form.
valid">Zamów</button>
    </label>
  </div>
</form>
```

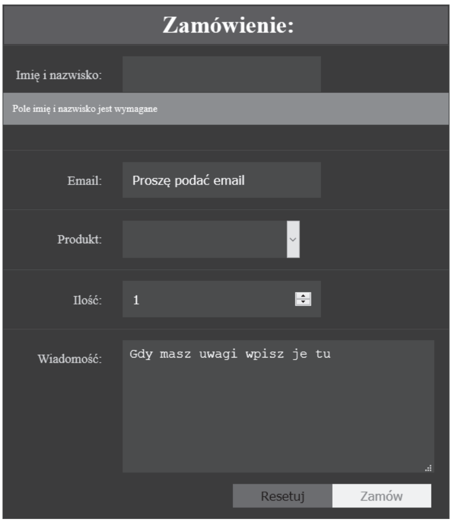
Użycie słowa kluczowego `required` powoduje, że dany element formularza jest wymagany. Nie będzie można go przesłać, jeśli pole nie zostanie uzupełnione. Dba o to kod, który znajduje się w definicji przycisku *Zamów*. Jest on nieaktywny, dopóki wymagane pola (`imieinazwisko`, `email` oraz `produkt`) są puste (rysunek 2.84).



Rysunek 2.84. Wysłanie formularza jest niemożliwe, ponieważ wymagane pole Produkt nie zostało określone

Pole `ilosc` zawiera dwa walidatory: `min` oraz `max`, których użycie powoduje, że liczba produktów została ograniczona do przedziału od 1 do 5.

Zastosowanie zmiennych szablonowych `#imieinazwiskoForm` oraz `#emailForm` wraz z przypisaniem ich do `ngModel` pozwala wyświetlić dodatkowe pola ostrzegawcze. Te dodatkowe okna są wyświetlane w momencie wybrania pola i opuszczenia go. Tę funkcjonalność zapewniają klasy `invalid` oraz `touched`. Ich działaniem steruje dyrektywa `ngIf`. Komunikaty zostały wystylizowane za pomocą klasy `.alert {font-size: 10px; padding: 10px; background-color: #f44336; color: white; margin-bottom: 15px;}` (rysunek 2.85).



Rysunek 2.85. Dodatkowe alerty

Wszystkie dostępne stany pól formularza są pokazane w tabeli 2.3.

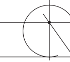
Tabela 2.3. Klasy określające stan weryfikacji danych formularza HTML

Klasa	Opis
`valid`	Formularz został poprawnie wypełniony
`invalid`	Formularz nie został poprawnie wypełniony
`submitted`	Formularz został zatwierdzony
`touched`	Pole formularza zostało wybrane przez kliknięcie lub klawiszem *TAB*
`untouched`	Pole formularza nie zostało wybrane
`dirty`	W polu formularza było coś wpisane (mogło zostać później wykasowane)
`pristine`	W polu formularza nic nie było wpisane (jest dziewicze)

Dostępne są jeszcze stany:

- `status` — status danego pola (czy dane pole — nie cały formularz — jest typu `invalid`, czy `valid`),
- `disabled` — czy dane pole jest typu `disabled` (zwraca `true`/`false`),
- `errors` — indeks wskazujący błąd, przez który pole przyjmuje stan `invalid`,
- `value` — przechowuje wartość pola `input` wpisaną przez użytkownika.

W polu `email` dodatkowo zostało zawarte wyrażenie `pattern=".+@.+"`, które sprawdza, czy ustalony wzorzec pasuje do adresu email. Sprawdzane jest wystąpienie znaku `@` oraz to, czy przed tym znakiem i po nim występuje tekst.



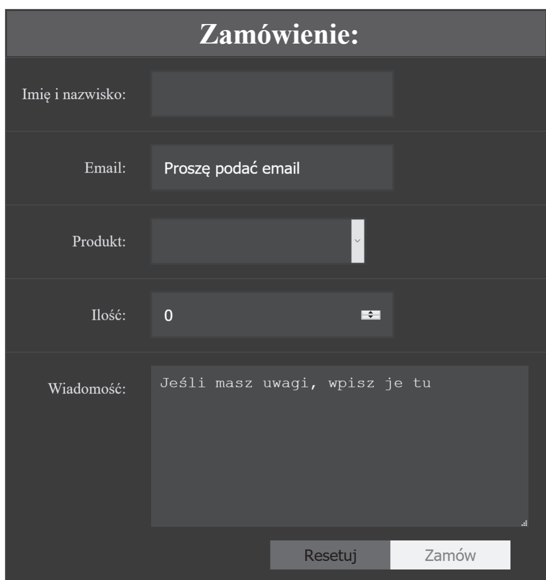
CIEKAWOSTKA

Wyrażenie, które w 99,99% gwarantuje walidację adresu email, ma postać (na podstawie RFC 5322 — *https://tools.ietf.org/html/rfc5322*):

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\
x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\
x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-
9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\
x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

```
choć o wiele łatwiej jest użyć formy: <input type="email"
placeholder="Proszę podać email" />.
```

Ponieważ zamieszczone w tabeli stany pola formularza są klasami, można im nadać style za pomocą arkusza stylów. Dodanie np. stylu `.ng-invalid {border: 2px solid blue};` (stan poprzedzamy ciągiem `ng-`) spowoduje, że wszystkie wymagane pola będą obramowane niebieską ramką (rysunek 2.86).



Rysunek 2.86. Użycie klasy .ng-invalid

Zadanie 2.25.

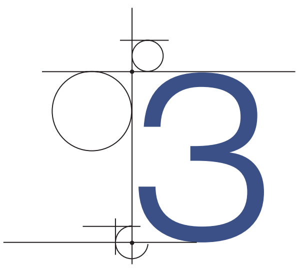
Stwórz ankietę składającą się z trzech pól:

- *Imię i nazwisko* — pole typu `text`,
- *W skali od 1 (najniżej) do 5 (najwyżej) oceń jakość obsługi klienta* — lista rozwijana z polami od *1* do *5*,
- *Status zawodowy: uczeń/student, pracujący, bezrobotny, na rencie/emeryturze* — pola typu `radio`.

Poniżej umieść przycisk *Zatwierdź*. Wysłanie ankiety ma być możliwe po wpisaniu wartości we wszystkich polach. Wynik wyświetl w konsoli.

Pytania kontrolne

1. Jak zaimplementować formularz?
2. W jakim celu wykonuje się walidację formularza?
3. W jaki sposób można zweryfikować informacje wprowadzone w polach formularza?



3 Środowisko uruchomieniowe — platforma Node.js

Node.js to środowisko uruchomieniowe dla programów napisanych w JavaScripcie, które pozwala je uruchamiać poza przeglądarką.

3.1. Platforma Node.js

3.1.1. Czym jest Node.js?

Platforma Node.js jest oparta na **silniku V8**, stworzonym przez firmę Google i używanym w przeglądarce Google Chrome. Dodatkowo w skład platformy weszła biblioteka **libuv**, która zapewnia **asynchroniczność**.

Do stworzenia Node.js użyto języka C++. Zrobiono tak z dwóch powodów. Po pierwsze, silnik V8, który jest jednym z filarów narzędzia, również został napisany z wykorzystaniem tego języka. Po drugie, Node.js zyskał w ten sposób możliwość rozbudowania swoich funkcji o zewnętrzne rozszerzenia i biblioteki.

CIEKAWOSTKA

Kod źródłowy silnika V8 jest dostępny na licencji open source, co oznacza, że twórca zezwala na jego bezpłatne rozpowszechnianie i na wprowadzanie w nim zmian. Taki sposób licencjonowania pozwolił wykorzystać go w Node.js.

Zadaniem silnika jest kompilacja i wykonanie kodu napisanego w JavaScripcie *just in time* — natychmiast, w chwili wywołania. Implementuje on specyfikację ECMAScript (zasady i wytyczne, którymi musi się kierować język programowania) oraz zarządza pamięcią. Sam silnik

jest jednowątkowy i działa w sposób **synchroniczny**. Synchroniczność oznacza, że kod jest wykonywany po kolei (linijka po linijce). Wadą tego rozwiązania jest **blokowanie** się programu — aby przejść do następnego kroku, musi wykonać bieżący. Dlatego operacje takie jak połączenie z serwerem powodują zatrzymanie wykonywania kodu. Użycie rozwiązań zaszytych w Node.js sprawia, że operacje te są wykonywane **asynchronicznie** (działania, które wymagają czasu, nie powodują zatrzymania wykonywania kodu) — są przekazywane z silnika V8 do Node.js.

WSKAZÓWKA

Aby sprawdzić wersję użytego silnika, należy wydać polecenie *node -p process.versions.v8* lub *npm version*.

Instalacja Node.js jest omówiona w punkcie 2.1.1. Warto wiedzieć, że wraz z Node.js jest dostarczany **REPL**, czyli interaktywny terminal, dostępny po wpisaniu w wierszu poleceń (CLI) lub konsoli PowerShell polecenia `node`. Nazwa REPL pochodzi od pierwszych liter słów *read*, *eval*, *print* oraz *loop*.

Działanie terminala jest pokazane na rysunku 3.1. Po wpisaniu `node` zostaje wyświetlona wersja narzędzia wraz ze znakiem zachęty >. Aby napisać kod programu zajmującego więcej niż jedną linijkę, należy przejść do trybu edytora poleceniem `.editor`. Rysunek przedstawia funkcję `wyswietl` z jednym parametrem, `tekst`. Zadaniem funkcji jest wyświetlenie w konsoli przekazanego argumentu `tekst`. Aby wyjść z trybu edytora, należy wcisnąć kombinację klawiszy *Ctrl+D* (użycie *Ctrl+C* przerywa działanie edytora — definicje nie będą pamiętane).

```
Wiersz polecenia - node

C:\Users\luk>node
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> .editor
// Entering editor mode (Ctrl+D to finish, Ctrl+C to cancel)
let wyswietl = (tekst) => {
console.log(tekst)
}

undefined
> wyswietl("Witaj")
Witaj
undefined
> const pokaz = "Welcome"
undefined
> console.log(pokaz)
Welcome
undefined
>
```

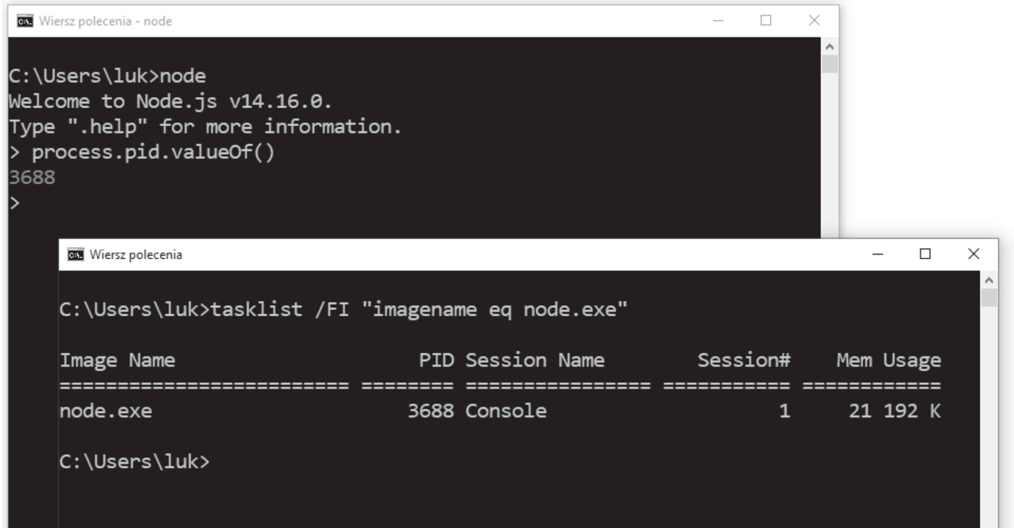
Rysunek 3.1. REPL

W następnym kroku do funkcji `wyswietl` zostaje przekazany argument `Witaj`, co powoduje jego wyświetlenie. Jednowierszowe definicje nie wymagają przejścia do trybu edytora, można je wpisywać bezpośrednio po znaku zachęty — zdefiniowana stała `pokaz` jest wyświetlana bezpośrednio w konsoli.

Cały wpisywany kod jest dostępny dopóty, dopóki narzędzie jest uruchomione. Aby zapisać bieżącą sesję do pliku, należy posłużyć się poleceniem `.save <nazwa_pliku>`. Aby wczytać plik, należy wydać polecenie `.load <nazwa_pliku>`.

Wciśnięcie dwa razy klawisza *Tab* spowoduje wyświetlenie listy funkcji i obiektów dostępnych z poziomu narzędzia REPL. Jednym z najważniejszych obiektów jest **`global`**, który jest nadrzędny dla wszystkich pozostałych, dlatego do metod i właściwości tego obiektu możemy się odwołać z poziomu każdego modułu. To podobnie jak w przypadku obiektu `window` w przeglądarkach. W środowisku Node.js odwołanie do obiektu `window` spowoduje błąd, ponieważ taki obiekt nie istnieje. Wpisanie `global.` i wciśnięcie dwa razy klawisza *Tab* spowoduje wyświetlenie wszystkich właściwości i metod obiektu (tym sposobem sprawdzisz także, co mają do zaoferowania pozostałe obiekty).

Drugim obiektem wartym poznania jest **`process`** — udostępnia on szereg metod i właściwości, które pozwolą uzyskać informacje o procesie oraz go kontrolować (w dalszej części podręcznika `process` wystąpi w kontekście modułu — nie jest to błąd, ponieważ obiekt `process` jest zwracany przez moduł `process`). Wpisanie np. `process.pid.valueOf()` spowoduje wyświetlenie numeru PID (ang. *Process Identifier*) procesu *node.exe*. Weryfikację PID procesu można przeprowadzić bezpośrednio w systemie Windows po wydaniu polecenia `tasklist /FI "imagename eq node.exe"`. Jak można zobaczyć na rysunku 3.2, oba numery są takie same.

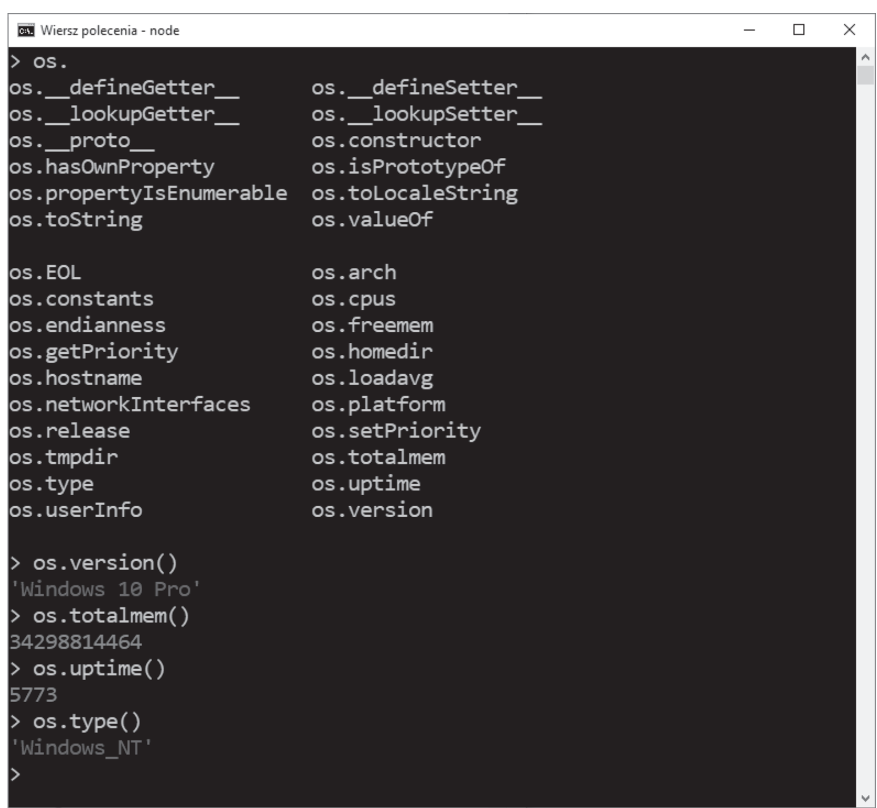


Rysunek 3.2. Obiekt process

UWAGA

PID procesu *node.exe* będzie różny od PID wiersza poleceń, za którego pośrednictwem proces został uruchomiony — to dwa oddzielne procesy. Do poznania PID wiersza poleceń można użyć polecenia *tasklist*, pokazanego powyżej, ale również `wmic process where name='cmd.exe' get ProcessId`. Aby poznać procesy, które zostały uruchomione za pośrednictwem terminala (ang. *child process*), należy użyć polecenia `wmic process where (ParentProcessId=`**`<numer PID procesu>`**`) get Caption,ProcessId`.

REPL pozwala również odwołać się do modułów podstawowych zainstalowanych wraz z Node.js. Jednym z takich modułów jest *os*, udostępniający informacje o systemie, w którym został zainstalowany Node.js. Użycie kilku metod tego modułu jest pokazane na rysunku 3.3.

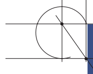


Rysunek 3.3. Metody modułu os

3.1.2. Menedżer pakietów npm

Narzędzie Node to tylko część całego środowiska, platformę tworzy również infrastruktura modułów, czyli zbiór pakietów (gotowych rozwiązań) dla Node. Za pobranie i instalację paczek odpowiedzialne jest narzędzie **npm** (ang. *node package manager*). W bazie znajduje się

ogromna liczba pakietów stworzonych przez rzeszę programistów skupionych wokół projektu. Dzięki temu możemy korzystać z bazy gotowych i sprawdzonych rozwiązań.



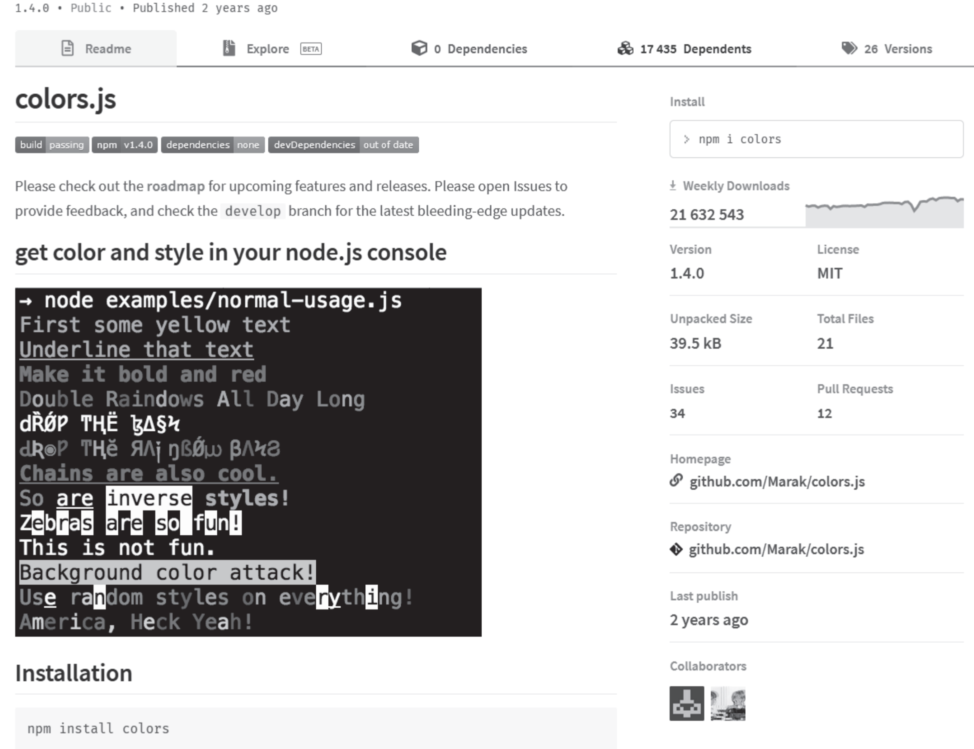
CIEKAWOSTKA

Opcją alternatywną dla menedżera npm jest narzędzie **yarn**, dostępne na stronie *https://yarnpkg.com/*.

Liczba pakietów do pobrania przekracza milion. Siłą rzeczy funkcjonalność wielu z nich się pokrywa, a problemem jest wybór tego, który oferuje najwięcej. Ułatwia to **wyszukiwarka** dostępna pod adresem *https://www.npmjs.com*. Korzystając z niej, najlepiej jest kierować się popularnością pakietu, częstotliwością jego aktualizacji oraz jakością dołączonej dokumentacji.

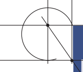
Sprawdźmy, co ma do zaoferowania przykładowy pakiet o nazwie *colors*, który pozwala kolorować tekst wyświetlany w konsoli.

Po wpisaniu w wyszukiwarce nazwy pakietu zostaniemy przeniesieni do strony zawierającej jego opis (rysunek 3.4).



Rysunek 3.4. Wyszukiwarka pakietów, informacje o pakiecie colors

Strona dostarcza informacji o aktualnej wersji pakietu, czasie jego opublikowania, licencji, liczbie pobrań (wraz z przedstawiającym ją wykresem) oraz liczbie i rozmiarze plików. Więcej szczegółów uzyskasz po przejściu na stronę projektu i stronę repozytorium (jeśli istnieją). W sekcji *Readme* znajdziesz opis działania rozszerzenia, przykłady użycia oraz sposób instalacji.



UWAGA

Słowo *pakiet* w kontekście Node.js ma bardzo szerokie znaczenie — pakietem może być narzędzie oferujące daną funkcjonalność, biblioteka oraz framework. Pakiet dołączony do projektu nazywamy zaś modułem.

Ogromna liczba pakietów sprawia również, że niektóre są **zależne od innych**. Lista wykorzystanych pakietów znajduje się w zakładce *Dependencies*. W naszym przykładzie *colors* nie korzysta z żadnych „gotowców", ale jest używany przez wiele innych, których lista jest dostępna w zakładce *Dependents*. Z kolei sekcja *Version* zawiera historię rozszerzenia — listę wszystkich jego wersji.

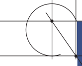
Wersja pakietu npm jest określana trzycyfrowym kodem (to tzw. wersjonowanie semantyczne — *https://semver.org/lang/pl/*), w którym cyfry są oddzielone od siebie kropką.

Przykładowe zapisy wersji pakietu należy interpretować jako:

- zmiana z 1.1.1 na 1.1.2 — drobna poprawka (ang. *patch*);
- zmiana z 1.3.6 na 1.4.0 — pakiet zawiera nową funkcję/mechanizm (ang. *minor*);
- zmiana z 1.6.3 na 2.0.0 — nowa generacja — zmiana dotyczy sposobu działania funkcji oferowanych przez pakiet, a instalacja nowszej wersji najczęściej będzie się wiązała z przebudową kodu (ang. *major*).

Pliki *package.json* i *package-lock.json*, tworzone po utworzeniu nowego projektu i zainstalowaniu wybranego pakietu (zobacz punkt 2.2.1), mogą zostać użyte do odtworzenia projektu. Zawartość obu plików po instalacji pakietów *colors* i *nodemon* jest pokazana na rysunku 3.5.

Informacja o zainstalowanym pakiecie *colors* znajduje się w sekcji `dependencies`, a wpis odnoszący się do pakietu *nodemon* został umieszczony w `devDependencies`. Stało się tak, ponieważ finalna wersja aplikacji będzie korzystała z funkcjonalności oferowanej jedynie przez pakiet *colors*, a pakiet *nodemon* został użyty na etapie jej tworzenia (innymi słowy, ten pakiet jest zależnością deweloperską, *nodemon* nie jest wymagany do działania aplikacji). Pakiety tego typu instalujemy z użyciem flagi `--save-dev` lub `-D`.



WSKAZÓWKA

Przy informacji o wersji obu pakietów znajduje się znak `^`. Użycie go zabrania aktualizacji pakietu do nowej generacji. Na przykład będzie możliwe uaktualnienie pakietu *colors* z 1.4.0 do 1.5.0, ale do 2.0.0 — już nie.

File Edit Selection View Go Run ··· package.json - intro - Visual Stud...

{} package.json × {} package-lock.json ×

```
{} package.json > ...
1   {
2     "name": "intro",
3     "version": "1.0.0",
4     "description": "",
5     "main": "index.js",
      ▷ Debug
6     "scripts": {
7       "test": "echo \"Error: nc
8     },
9     "keywords": [],
10    "author": "",
11    "license": "ISC",
12    "dependencies": {
13      "colors": "^1.4.0"
14    },
15    "devDependencies": {
16      "nodemon": "^2.0.7"
17    }
18  }
19
```

```
{} package-lock.json > ...
1   {
2     "name": "intro",
3     "version": "1.0.0",
4     "lockfileVersion": 1,
5     "requires": true,
6     "dependencies": {
7       "@sindresorhus/is": {
8         "version": "0.14.0",
9         "resolved": "https://registry.n
10        "integrity": "sha512-9NET910DNa
11        "dev": true
12      },
13      "@szmarczak/http-timer": {
14        "version": "1.1.2",
15        "resolved": "https://registry.n
16        "integrity": "sha512-XIB2XbzHTN
17        "dev": true,
18        "requires": {
19          "defer-to-connect": "^1.0.1"
20        }
```

⊗ 0 ⚠ 0 Ln 19, Col 1 Spaces: 2 UTF-8 LF JSON - Terminal 17-pt +

Rysunek 3.5. Zawartość plików package.json i package-lock.json

Jeśli mamy oba pliki, **nie musimy** na przykład przekazywać współpracownikom całej zawartości katalogu *node_modules*, w którym znajdują się wszystkie wykorzystywane moduły, gdyż po wydaniu polecenia `npm install` ich stan zostanie odtworzony.

3.1.3. Format JSON

Oba przedstawione pliki są w formacie **JSON** (ang. *JavaScript Object Notation*). Format ten został stworzony z myślą o przechowywaniu i przekazywaniu danych. Obowiązująca notacja JSON jest prawie w całości zbieżna z definicją obiektu w języku JavaScript. Przykładowe dane zapisane w formacie JSON są przedstawione na listingu 3.1.

Wartościami w JSON mogą być: **ciąg znaków**, **liczba**, **obiekt**, **tablica**, **wartości prawda/fałsz** oraz **null**.

Klucz od wartości oddziela dwukropek (`:`), a poszczególne wartości od siebie — przecinek (`,`).

Listing 3.1. Zapis danych w formacie JSON

```
{
    "quiz": {
        "sport": {
            "q1": {
                "question": "W którym roku odbyły się mistrzostwa Europy
w piłce nożnej organizowane przez Polskę i Ukrainę?",
```

```
                "options": [
                    "2008",
                    "2012",
                    "2016",
                    "Polska i Ukraina nigdy nie były gospodarzami turnieju"
                ],
                "answer": "2012"
            }
        },
        "maths": {
            "q1": {
                "question": "5! = ?",
                "options": [
                    "10",
                    "25",
                    "75",
                    "120"
                ],
                "answer": "120"
            },
            "q2": {
                "question": "5 ^ 3 = ?",
                "options": [
                    "15",
                    "25",
                    "53",
                    "125"
                ],
                "answer": "125"
            }
        }
    }
}
```

UWAGA

Słowo „prawie” w opisie notacji nie pojawiło się przypadkowo. Aby kod JSON działał, klucze i wartości muszą być objęte znakami cudzysłowu ("). Nie można przy tym użyć znaku przecinka po ostatnim elemencie listy. Walidację kodu można przeprowadzić na przykład z wykorzystaniem strony *https://jsonlint.com/*.

Format JSON jest bezpośrednio obsługiwany przez JavaScript i Node.js, o czym zaraz się przekonasz. Wykonajmy dwa ćwiczenia. W pierwszym zamienimy dane zapisane w forma-

cie JSON na obiekt JavaScriptu, a w drugim — odwrotnie: informacje o obiekcie zapiszemy w formacie JSON.

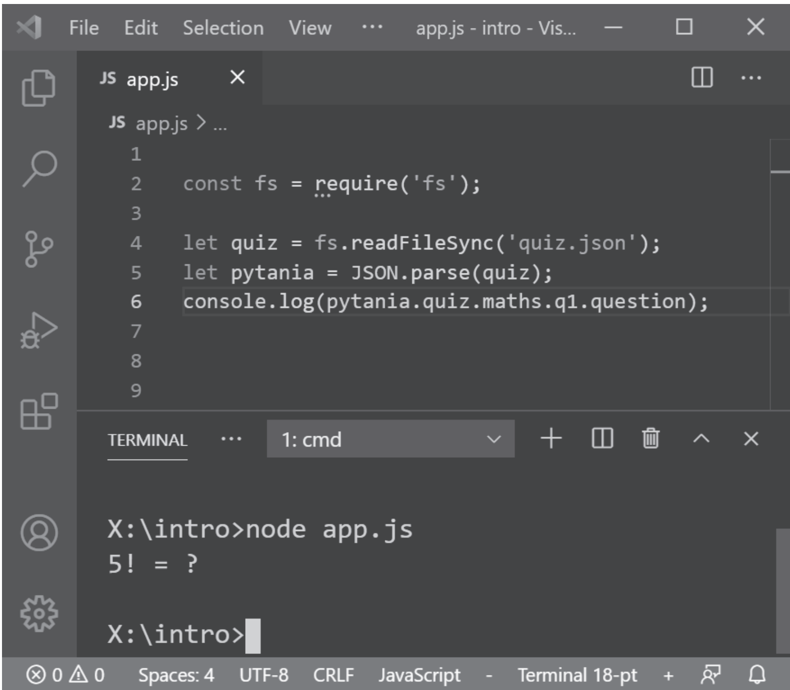
W pierwszym zadaniu należy użyć **obiektu `JSON`** i udostępnianej przez niego metody **`.parse()`**. Jako argument metoda przyjmuje łańcuch znaków zawierający kod JSON. Kod ten pobierzemy z zewnętrznego pliku. Kod pokazany na listingu 3.1 został zapisany w pliku *quiz.json*.

Rozwiązanie pierwszego zadania jest przedstawione na listingu 3.2. Do pobrania zawartości pliku *quiz.json* użyto modułu `fs` i metody `readFileSync` — zawartość pliku *quiz.json* zostaje przypisana do zmiennej `quiz`. W następnym kroku użyto metody `parse`, przekształcającej dane na obiekt. Dostęp do nich uzyskamy z zastosowaniem notacji z kropką.

***Listing 3.2.** Zamiana informacji zapisanych w pliku JSON na obiekt JavaScriptu*

```
const fs = require('fs');

let quiz = fs.readFileSync('quiz.json');
let pytania = JSON.parse(quiz);
console.log(pytania.quiz.maths.q1.question);
```

Rezultatem wywołania kodu jest wyświetlenie treści pierwszego pytania z matematyki (rysunek 3.6).

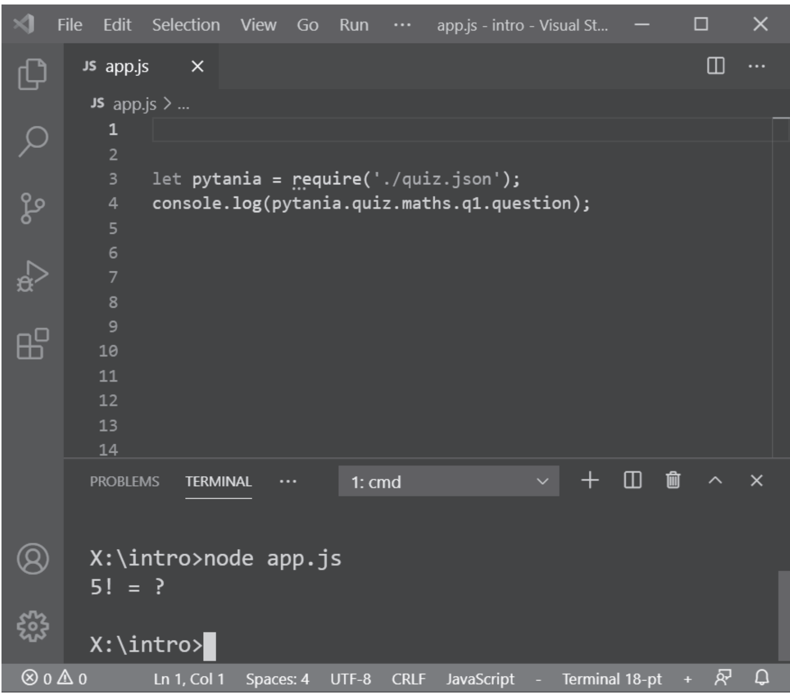


Rysunek 3.6. Zamiana informacji zapisanych w pliku JSON na obiekt JavaScriptu

UWAGA

Metoda `readFileSync` odczytuje plik w sposób synchroniczny. Aby rozpocząć proces odczytu zawartości bez zatrzymania działania programu, należy użyć metody `readFile`.

Podobny efekt uzyskamy za pomocą funkcji **`require`**, która jest stosowana do załadowania modułu — można jej również użyć do załadowania danych zapisanych z wykorzystaniem formatu JSON. Nie trzeba stosować metody `.parse()`, gdyż dane te są automatycznie parsowane (rysunek 3.7). Załączany plik musi mieć rozszerzenie *.json*.



Rysunek 3.7. Użycie słowa kluczowego require w połączeniu z plikiem .json

UWAGA

Jeśli zaktualizujesz plik, nie będziesz mógł ponownie go odczytać z użyciem tej metody.

Metoda **`.stringify()`** przekształca obiekt w łańcuch znaków (rysunek 3.8). Obiektem jest pierwsze pytanie z kodu przedstawionego na listingu 3.1.

```
let pytanie = {
    "sport": {
        "q1": {
            "question": "W którym roku odbyły się mistrzostwa Europy w piłce n
            "options": [
                "2008",
                "2012",
                "2016",
                "Polska i Ukraina nigdy nie były gospodarzami turnieju"
            ],
            "answer": "2012"
        }
    },
}
let toString = JSON.stringify(pytanie);
console.log(toString);
```

```
{"sport":{"q1":{"question":"W którym roku odbyły się mistrzostwa Euro
iłce nożnej organizowane przez Polskę i Ukrainę?","options":["2008","
2012","2016","Polska i Ukraina nigdy nie były gospodarzami turnieju"]
,"answer":"2012"}}}
```

Rysunek 3.8. Użycie metody .stringify()

Zadanie 3.1.

Sprawdź w wyszukiwarce pakietów przeznaczenie frameworka Express. Oceń jego popularność oraz sprawdź, z ilu innych pakietów korzysta i ile jest od niego zależnych.

Zadanie 3.2.

Zaproponuj plik w formacie JSON, który przechowywałby informacje o księgozbiorze biblioteki lub ofercie wypożyczalni filmów.

Pytania kontrolne

1. Czym jest REPL i w jaki sposób można wykorzystać to narzędzie?
2. Jaką funkcję pełni menedżer pakietów npm?
3. Omów przeznaczenie i budowę pliku w formacie JSON.
4. Jak działają metody `.parse()` i `.stringify()`?

3.2. Asynchroniczność w Node.js

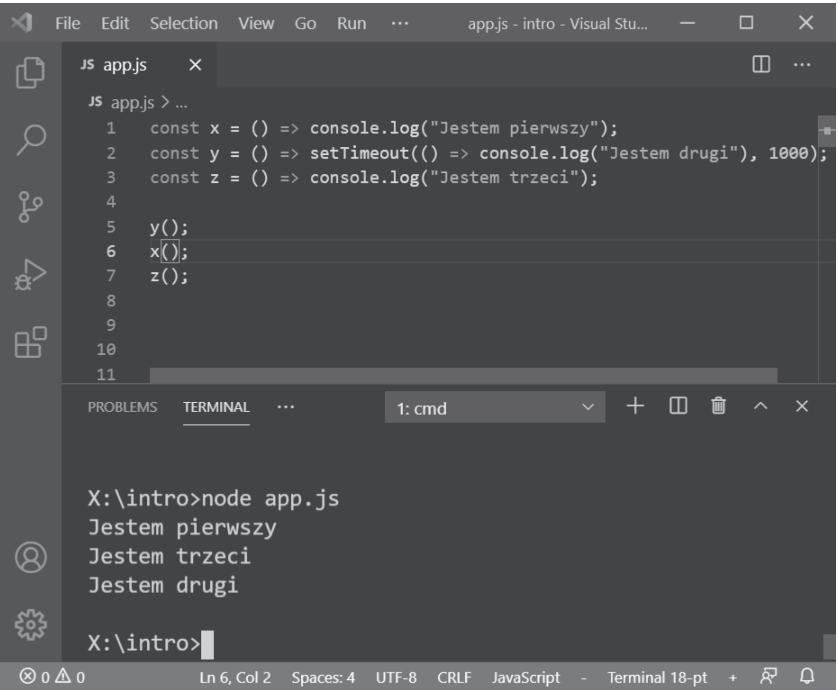
Skoro silnik V8, na którym Node.js opiera swoje działanie, pracuje w sposób synchroniczny, podobnie zresztą jak sam język JavaScript, rodzi się pytanie: jakie rozwiązania sprawiły, że Node.js pomimo użycia synchronicznego silnika pracuje w sposób asynchroniczny, czyli bez blokowania programu?

Przyjrzyjmy się kodowi z listingu 3.3.

Listing 3.3. Asynchroniczność

```
const x = () => console.log("Jestem pierwszy");
const y = () => setTimeout(() => console.log("Jestem drugi"), 1000);
const z = () => console.log("Jestem trzeci");

y();
x();
z();
```

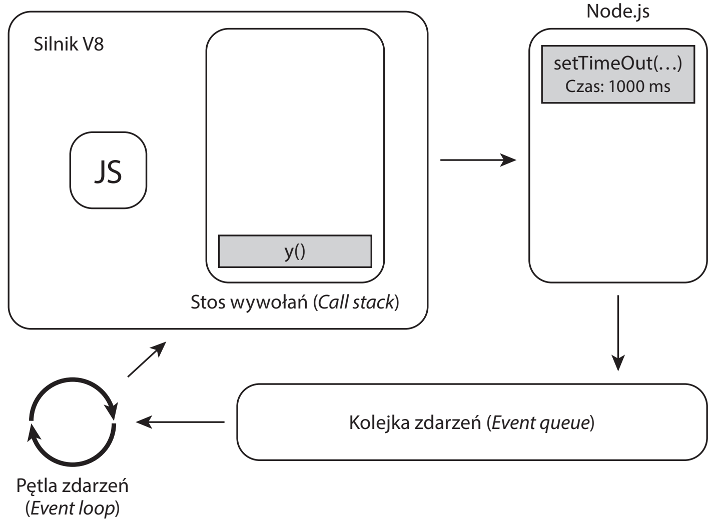
Efekt działania kodu jest pokazany na rysunku 3.9. W pierwszej kolejności wywoływana jest funkcja `y()`, tuż po niej `x()`, a na samym końcu `z()`. Jak można zaobserwować, program nie zatrzymał się na pierwszej linijce (tak by się stało, gdyby działał w trybie synchronicznym), lecz przeszedł do drugiej i trzeciej — wyświetlone zostają teksty *Jestem pierwszy*, *Jestem trzeci*, a dopiero na samym końcu, z 1-sekundowym opóźnieniem, *Jestem drugi*.



Rysunek 3.9. Asynchroniczność

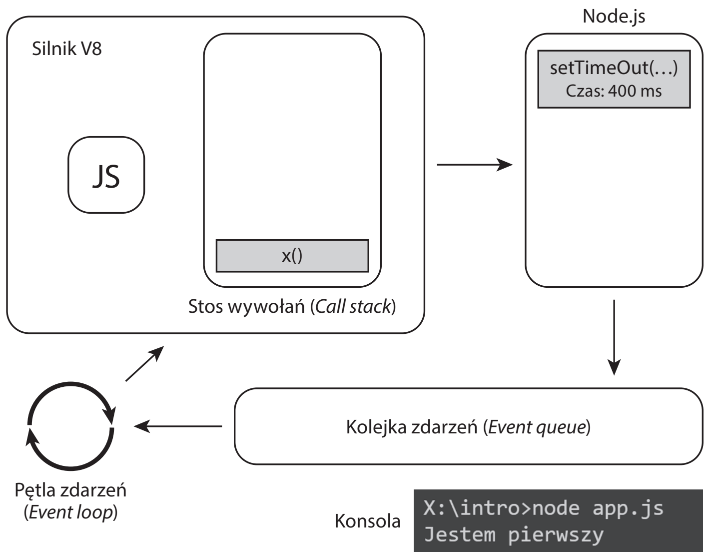
Zobrazujmy to za pomocą serii grafik.

Rozpoczynamy od wywołania funkcji `y()`. Trafia ona na **stos wywołań** (ang. *call stack*) silnika V8. Ponieważ **wywołanie zwrotne** przekazane do `setTimeout` ma się wykonać za 1 s (nie może czekać na stosie wywołań), zostaje przekazane do Node.js, tak aby zrobić miejsce dla dalszych linijek kodu — rozpoczyna się odliczanie (rysunek 3.10).



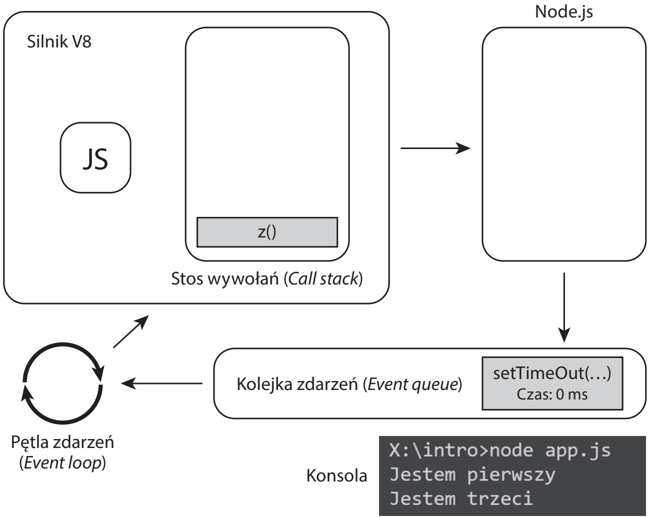
Rysunek 3.10. Wywołanie funkcji y() — krok pierwszy

W drugim kroku zostaje wywołana funkcja `x()`, której zadaniem jest wyświetlenie napisu *Jestem pierwszy*. Ciąg ten pojawia się w wierszu konsoli. Timer nadal odlicza czas, a wywołanie zwrotne czeka, aż upłynie (rysunek 3.11).



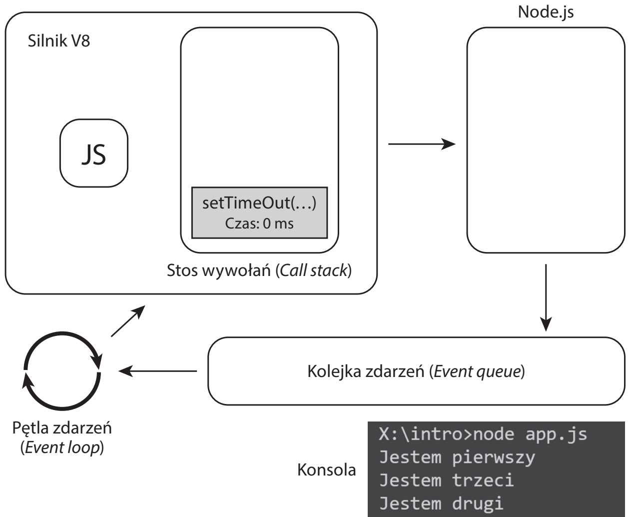
Rysunek 3.11. Wywołanie funkcji x() — krok drugi

Funkcja `x()` kończy swoje działanie i zostaje usunięta ze stosu wywołań. Jej miejsce zajmuje funkcja `z()`. Czas dobiega końca, wywołanie zwrotne może zostać przeniesione do **kolejki zdarzeń** (ang. *event queue*). Ponieważ na stosie wywołań znajduje się funkcja `z()`, nie może ona zostać wywołane natychmiast. W konsoli pojawia się napis *Jestem trzeci* (rysunek 3.12).



Rysunek 3.12. Wywołanie funkcji z() — krok trzeci

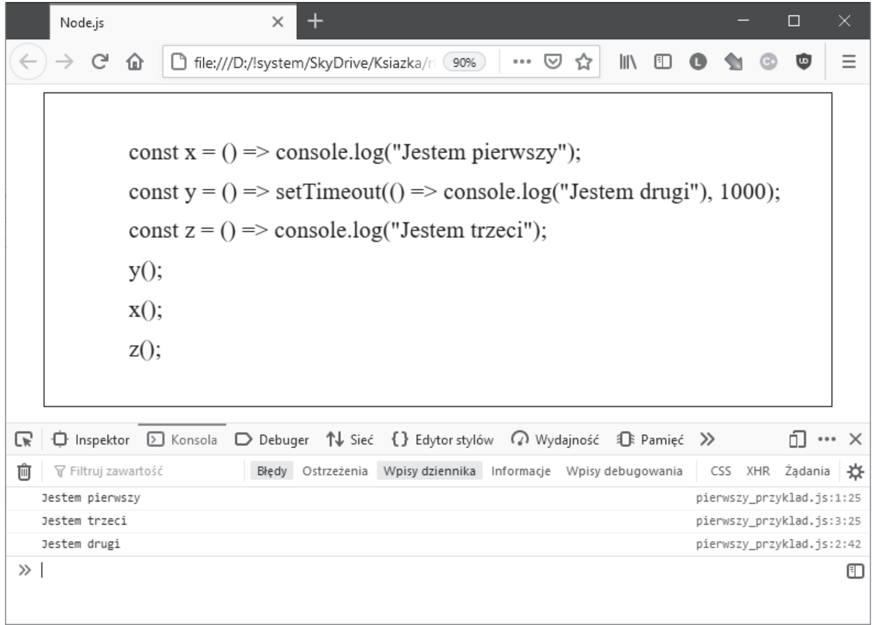
Pętla zdarzeń (ang. *event loop*) nieustannie monitoruje stos wywołań i kolejkę zdarzeń — **zdarzenie znajdujące się w kolejce nie może zostać przeniesione na stos, gdy coś się na nim znajduje**. Po zakończeniu funkcji `z()` stos wywołań jest zwalniany — wywołanie może zostać na nim umieszczone — i w konsoli pojawia się napis *Jestem drugi* (rysunek 3.13).



Rysunek 3.13. Zakończenie programu — krok czwarty

Użycie kolejki zdarzeń w połączeniu z pętlą zdarzeń pozwoliło **uzyskać asynchroniczność** działania kodu pomimo użycia synchronicznego silnika V8.

Identyczny efekt asynchroniczności zapewniają przeglądarki — ten sam kod, lecz uruchomiony w przeglądarce, jest pokazany na rysunku 3.14.



Rysunek 3.14. Asynchroniczność w przeglądarce

Pytania kontrolne

1. Co oznacza, że aplikacja/program działa w sposób synchroniczny, a co — że w sposób asynchroniczny?
2. W jaki sposób uzyskano asynchroniczność w Node.js?

3.3. Moduły w Node.js

Moduł w Node.js jest elementem (cegiełką), który można wykorzystać we własnym projekcie — raz zaimportowany, może być używany wielokrotnie. Może mieć postać pojedynczego pliku, ale także składać się z wielu plików umieszczonych we wspólnym katalogu. Każdy moduł ma własny **zakres** (ang. *scope*), co sprawia, że **kod zdefiniowany w module nie jest dostępny poza nim**.

Moduły dzielą się na **wbudowane** (dostarczane wraz z Node.js), **zewnętrzne** (rozwijane i tworzone przez niezależnych programistów, jak również społeczność skupioną wokół projektu) oraz **własne** (napisane przez ich użytkownika).

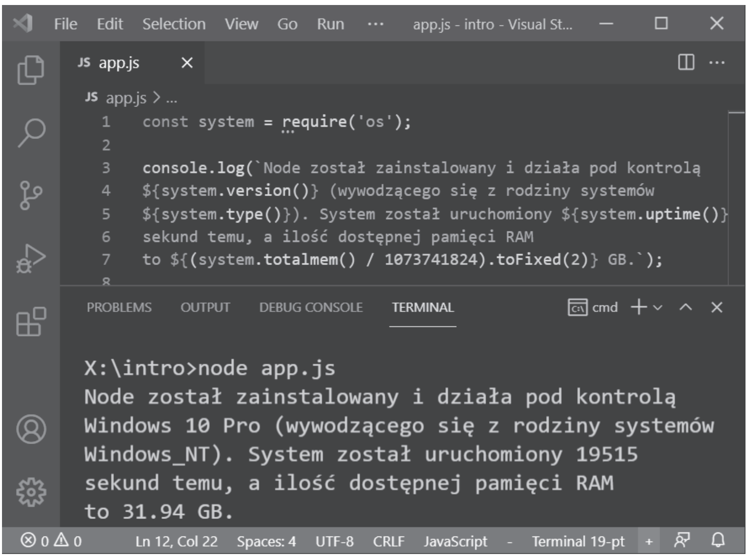
Moduły w Node.js należy traktować jako pełnoprawne aplikacje, a także biblioteki czy frameworki, dlatego aby można było użyć modułu, trzeba go zaimportować. Służy do tego funkcja **`require()`**.

Aby móc w swoim projekcie wykorzystać metody przedstawione na rysunku 3.3, należy rozpocząć od zaimportowania modułu `os`. Import i użycie metod zdefiniowanych w module `os` przedstawia kod z listingu 3.4. Moduł `os` zostaje przypisany do zmiennej `system` — od tej pory wszystkie właściwości i metody modułu są dostępne po użyciu **znaku kropki** (`.`).

Listing 3.4. Użycie modułu os

```
const system = require('os');

console.log(`Node został zainstalowany i działa pod kontrolą ${system.
version()} (wywodzącego się z rodziny systemów ${system.type()}). System został
uruchomiony ${system.uptime()} sekund temu, a ilość dostępnej pamięci RAM to
${(system.totalmem() / 1073741824).toFixed(2)} GB.`);
```

Efekt wywołania kodu jest pokazany na rysunku 3.15.



Rysunek 3.15. Import i użycie modułu os

Do podstawowych modułów Node.js zaliczamy:

- **`url`** — służy do parsowania (analiza ciągu znaków w celu ustalenia struktury) adresów URL;
- **`fs`** — udostępnia metody przeznaczone do pracy z systemem plików;
- **`path`** — jest odpowiedzialny za przechowywanie i przetwarzanie ścieżek do plików i katalogów;
- **`http`** — pozwala utworzyć serwer WWW;
- **`querystring`** — przekształca obiekt na łańcuch.

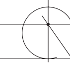
Do modułów, których nie trzeba importować, należą: **process** (dostarczanie informacji o procesie), **console** (obsługa strumienia wyjścia) i **timers** (za jego pomocą wywołamy funkcje w określonym czasie, np. z użyciem metody setTimeOut lub setInterval).

3.3.1. Moduł http

Do zbudowania serwera HTTP został wykorzystany moduł podstawowy — **http**. Jego użycie jest pokazane na listingu 3.5.

Aby można było skorzystać z modułu, jest tworzona zmienna http, do której za pomocą metody require() zostaje on zaimportowany. Dodatkowe dwie zmienne, host i port, są stosowane do definiowania adresu serwera.

Metoda **.createServer()** pozwala zainicjować serwer. Otwarcie strony generuje zapytanie request, na które serwer udziela odpowiedzi (response). Obiekty request i response użyte jako parametry wywołania zwrotnego (ang. *callback*) pozwalają na obsługę żądań pochodzących od klienta i udzielonych mu odpowiedzi. Status nagłówka odpowiedzi HTTP zostaje ustawiony na 200 (OK), a przekazywana zawartość to HTML.



CIEKAWOSTKA

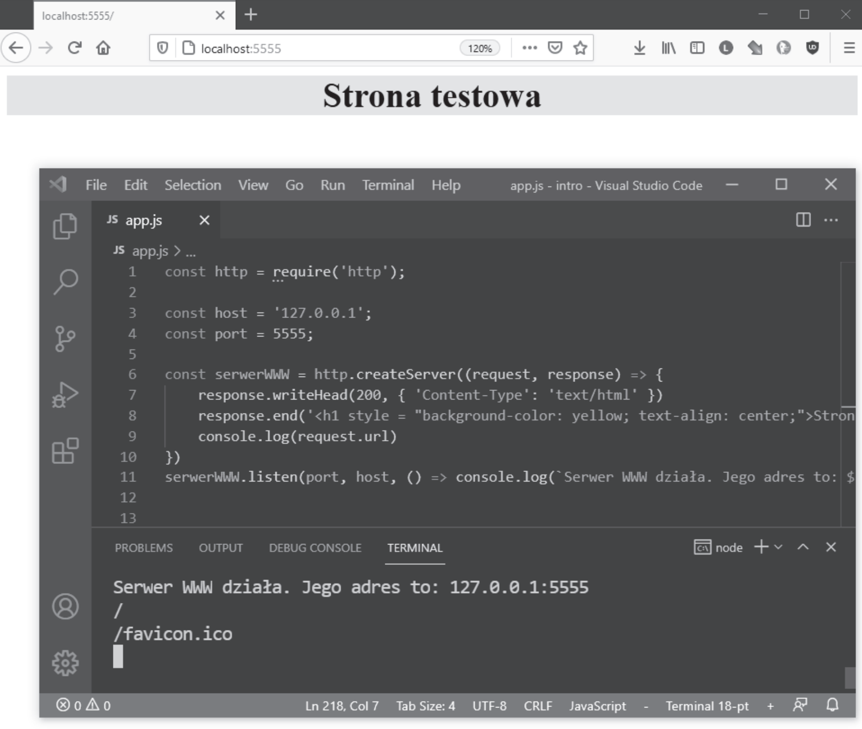
Użyta **wartość 200** określa status odpowiedzi, który wynika ze sposobu działania protokołu HTTP — szczegóły opisuje dokument RFC 2616 (ang. *Request for Comments*). **RFC** to zbiór dokumentów opisujących działanie protokołów i usług związanych z internetem i sieciami komputerowymi. Każdy z nich ma przypisany unikatowy numer identyfikacyjny, często spotykany w odniesieniach bądź komentarzach.

Użycie response.end pozwala zdefiniować odpowiedź, która zostanie odesłana do klienta. Za pomocą obiektu console został wyświetlony adres URL zapytania (adres wpisany w oknie przeglądarki) wraz z adresem IP i numerem portu serwera HTTP.

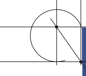
Listing 3.5. Moduł http

```
const http = require('http');

const host = '127.0.0.1';
const port = 5555;

const serwerWWW = http.createServer((request, response) => {
    response.writeHead(200, { 'Content-Type': 'text/html' });
    response.end('<h1 style = "background-color: yellow; text-align:
center;">Strona testowa</h1>');
    console.log(request.url);
})
serwerWWW.listen(port, host, () => console.log(`Serwer WWW działa. Jego adres
to: ${host}:${port}`));
```

Działanie kodu jest pokazane na rysunku 3.16.



Rysunek 3.16. Działanie serwera HTTP



UWAGA

Jak widać na rysunku 3.16, oprócz adresu strony generowane jest zapytanie o *favicon* — ikonę, która pojawia się przed adresem w polu adresowym przeglądarki internetowej.

3.3.2. Moduł url

Moduł `url` służy do rozpoznawania i analizowania adresów URL. Najważniejszą i najczęściej wykorzystywaną metodą tego modułu jest `.parse()`. Jej użycie jest pokazane na listingu 3.6.

Listing 3.6. Moduł url i metoda .parse()

```
const url = require('url');
const addr = 'http://localhost:5555/strona.html';
const q = url.parse(addr, true);

console.log(q.host);
console.log(q.pathname);
console.log(q.hostname);
console.log(q.href);
```

Efektem działania kodu jest wyświetlenie w konsoli: `localhost:5555`, `/strona.html`, `localhost`, `http://localhost:5555/strona.html`.

Jednak metoda `.parse()` jest od wydania 11 Node.js przestarzała (choć nadal działa i jest powszechnie stosowana).

Parsowanie adresów URL należy zacząć od utworzenia za pomocą `new URL()` nowego obiektu (listing 3.7). Przypisanie obiektu do zmiennej i wyświetlenie jej spowoduje zwrócenie adresu w formacie JSON. Użycie metody `.toString()` na polu `href` pozwoli wyodrębnić adres URL.

Listing 3.7. Użycie obiektu URL

```
const url1 = new URL('http://localhost:5555/strona.html');
console.log(url1);
const url2 = new URL(
    { toString: () => 'http://localhost:5555/strona.html' });
console.log(url2.href);
```

3.3.3. Moduł fs

Moduł **`fs`** zapewnia metody (synchroniczne i asynchroniczne), które umożliwiają pracę z systemem plików. Użycie tego modułu pozwala zarówno odczytywać zawartość plików, jak i je zapisywać.

Kod z listingu 3.8 pokazuje, w jaki sposób można sprawdzić, czy plik istnieje i czy możliwy jest zapis do niego. Za pomocą metody **`.access()`** w połączeniu z parametrem `fs.constants.F_OK` sprawdzane jest istnienie pliku. Drugi parametr, `fs.constants.W_OK`, sprawdza stan ustawienia flagi „tylko do odczytu". Gdy we właściwościach pliku jest ustawiony atrybut „tylko do odczytu", zapis do pliku jest niemożliwy. Zaistnienie jednej z dwóch opisanych sytuacji — ponieważ oba parametry łączy operator pionowej kreski (`|`), czyli „lub" — wygeneruje błąd. Użycie instrukcji `if` w połączeniu z numerem błędu (`-4058` oznacza brak pliku) pozwala go zidentyfikować. Za wyświetlenie komunikatu błędu odpowiada operator warunkowy.

Listing 3.8. Moduł fs — metoda access

```
const fs = require('fs');
const plik = 'test.txt';

fs.access(plik, fs.constants.F_OK | fs.constants.W_OK, (err) => {
    if (err) {
        console.log(
            `${plik} ${err.errno === -4058 ? 'nie istnieje' : 'tylko do
odczytu'}`);
    } else {
        console.log(`plik ${plik} istnieje i pozwala na zapis`);
    }
});
```

WSKAZÓWKA

Jeśli błędu nie ma, `err` jest równe `null`. Identyfikację błędu można przeprowadzić również z użyciem kodu błędu. Błąd o numerze `-4058` spowoduje wyświetlenie kodu `ENOENT` (rysunek 3.17). Użycie kodu błędu wymusi zmianę kodu skryptu — `err.errno === -4058` należy zamienić na `err.code === 'ENOENT'`. Brak możliwości zapisu do pliku to błąd o numerze `-4048`, co odpowiada kodowi `EPERM`. Wartości numeru błędu wraz z jego kodem sprawdzimy z wykorzystaniem kodu `console.log(err)`.

Metoda **`.readdir()`** odpowiada za wyświetlenie zawartości katalogu (listing 3.9). Parametrem metody jest zmienna `katalog`, przechowująca ścieżkę do katalogu, którego zawartość ma być pokazana. W wywołaniu zwrotnym znajdują się parametry `err` i `dane`. Do `err` zostanie przypisana wartość ewentualnego błędu, a `dane` przechowa informacje o zawartości katalogu. Do wyświetlenia zawartości katalogu zastosowano metodę `forEach` — jej użycie jest dozwolone, ponieważ zwróconym typem jest **tablica**. Dodatkowa zmienna `licznik` zwraca liczbę wszystkich elementów (plików i katalogów) znajdujących się w wyświetlanym katalogu. Jej wartość jest zwiększana o jeden w każdym następnym przebiegu pętli.

Listing 3.9. Moduł fs — metoda readdir

```
const fs = require('fs');
const katalog = './';
let licznik = 0;
fs.readdir(katalog, (err, dane) => {
    if (err) console.log(err)
    else {
        console.log("\nZawartość katalogu:");
        dane.forEach(plik => {
            console.log(plik);
            licznik++;
        })
    } console.log(`\nLiczba wszystkich elementów w katalogu: ${licznik}`);
})
```

Brak katalogu (lub błędnie wpisana ścieżka) spowoduje wyświetlenie komunikatu o błędzie (rysunek 3.17).

Następna metoda, **`.readFile()`**, odczytuje zawartość pliku. Sposób jej użycia jest przedstawiony na listingu 3.10. Do metody jest przekazywana ścieżka do pliku, którego zawartość ma zostać wyświetlona, oraz kodowanie. W funkcji wywołania zwrotnego znajdują się dwa elementy: parametr `err`, do którego zostanie przekazany błąd, np. wynikły ze złej definicji ścieżki, oraz parametr `zawartosc`, którego zadaniem jest przechowanie i przekazanie zawartości pliku.

```
const fs = require('fs');
const katalog = '.brak/';
let licznik = 0;
fs.readdir(katalog, (err, dane) => {
    if (err) console.log(err);
    else {
        console.log("\nZawartość katalogu:");
        dane.forEach(plik => {
            console.log(plik);
            licznik++;
        })
    } console.log(`\nLiczba wszystkich elementów w katalogu: ${licznik}`);
})
```

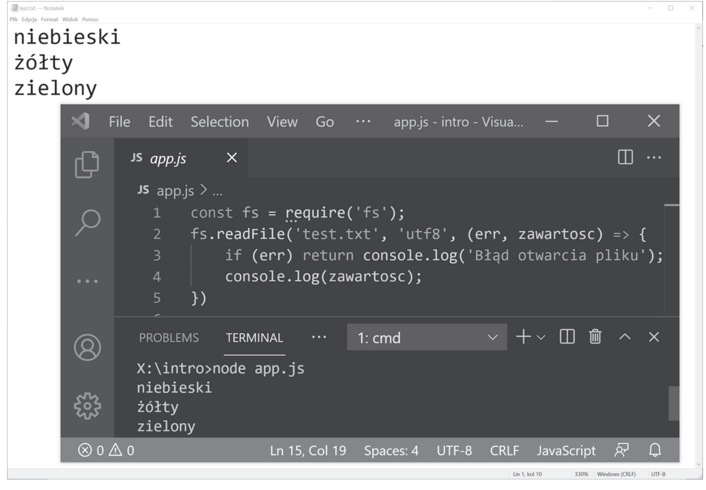
```
o\.brak'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'scandir',
  path: 'X:\\x\\intro\\.brak'
}
```

Rysunek 3.17. Błąd — katalog o zdefiniowanej nazwie nie istnieje

Listing 3.10. Użycie metody .readFile()

```
const fs = require('fs');
fs.readFile('test.txt', 'utf8', (err, zawartosc) => {
    if (err) return console.log('Błąd otwarcia pliku');
    console.log(zawartosc);
})
```

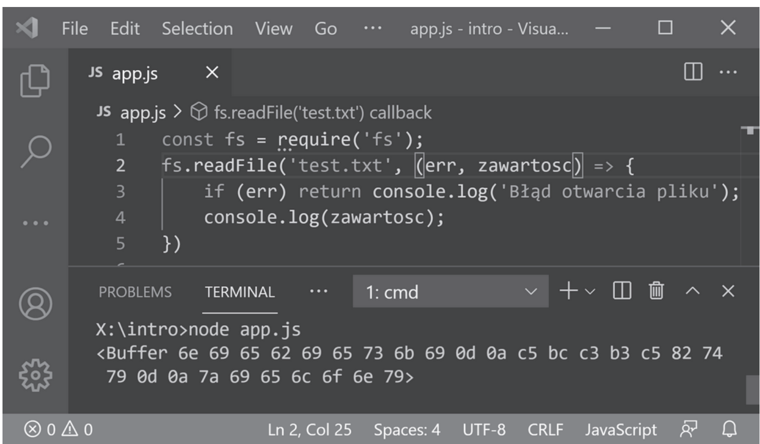
Zawartość pliku *test.txt* jest pokazana na rysunku 3.18.



Rysunek 3.18. Użycie metody .readFile()

WSKAZÓWKA

Pominięcie parametru `utf8` spowoduje (działanie domyślne) otwarcie pliku binarnie (rysunek 3.19).



Rysunek 3.19. Użycie metody .readFile() — odczyt binarny

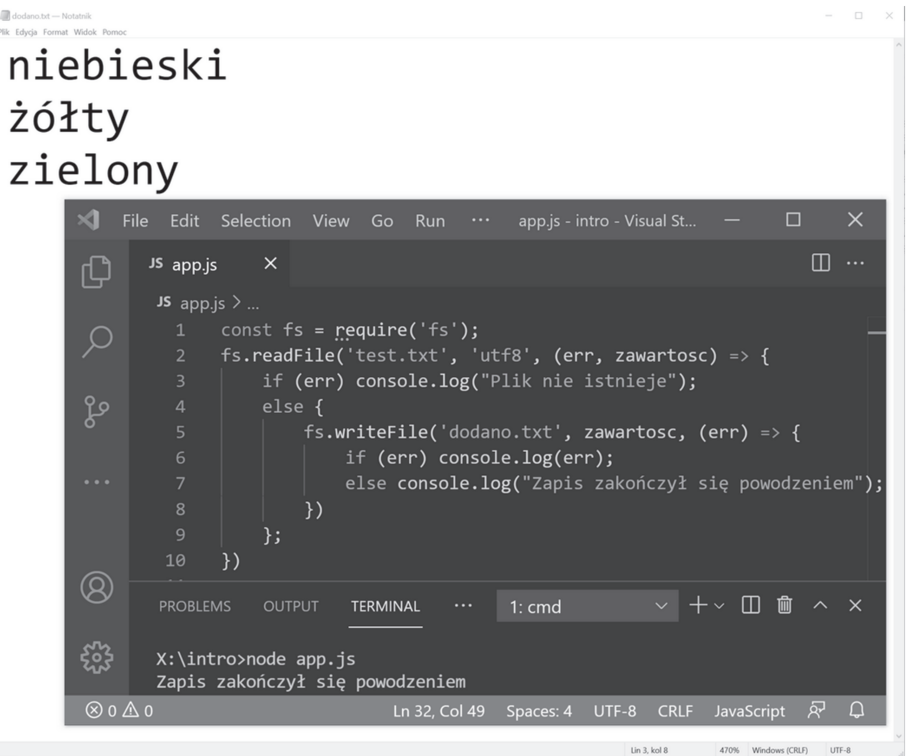
Metoda **`.writeFile()`** zapisuje informacje do pliku. Jej użycie jest przedstawione na listingu 3.11. Kod tworzy nowy plik, *dodano.txt*, który jest kopią pliku *test.txt* (zawartość tego pliku jest pokazana na rysunku 3.18). Za odczytanie pliku *test.txt* odpowiada zaś metoda `.readFile()`. Po odczytaniu zawartość pliku zostaje przekazana do metody `.writeFile()` (rysunek 3.20).

Listing 3.11. Użycie metody .writeFile()

```
const fs = require('fs');
fs.readFile('test.txt', 'utf8', (err, zawartosc) => {
    if (err) console.log("Plik nie istnieje");
    else {
        fs.writeFile('dodano.txt', zawartosc, (err) => {
            if (err) console.log(err);
            else console.log("Zapis zakończył się powodzeniem");
        })
    };
})
```

UWAGA

Określenie kodowania w metodzie `.writeFile()` jest zbędne — domyślnie ustawione jest UTF-8.



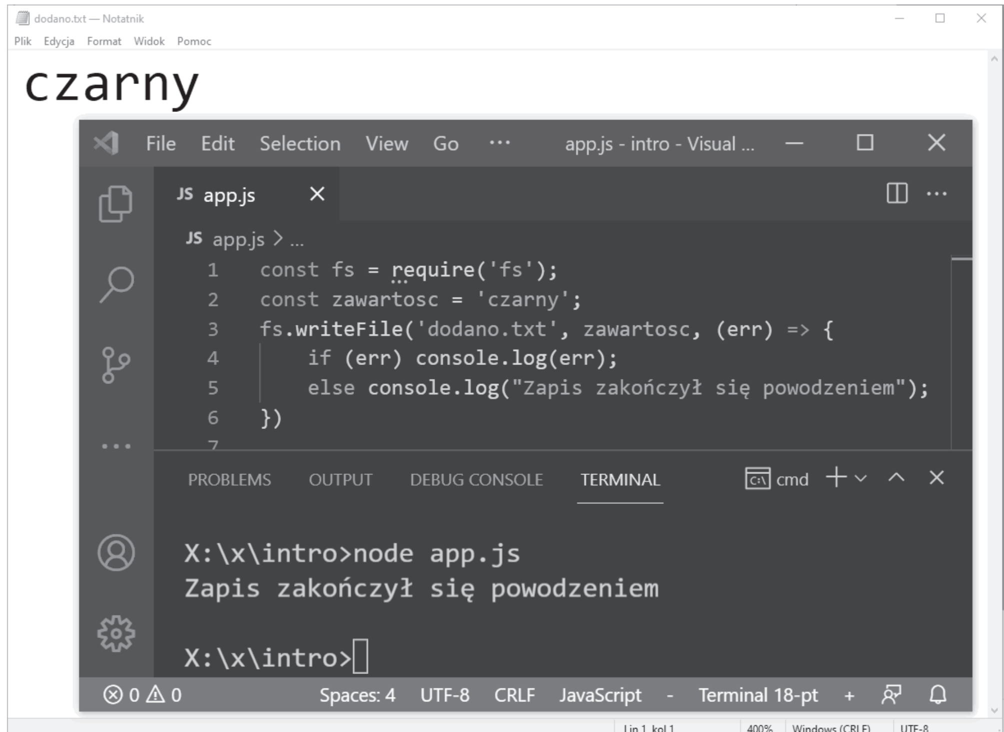
Rysunek 3.20. Użycie metody .writeFile() w połączeniu z metodą .readFile()

Kod na listingu 3.12 ma dopisać do pliku *dodano.txt* nowy kolor — `czarny`. Niestety próba dopisania kończy się **niepowodzeniem** — użycie metody `.writeFile()` sprawi, że **zawartość pliku zostanie zastąpiona**.

Listing 3.12. Próba dodania do pliku nowej zawartości

```
const fs = require('fs');
const zawartosc = 'czarny';
fs.writeFile('dodano.txt', zawartosc, (err) => {
    if (err) console.log(err);
    else console.log("Zapis zakończył się powodzeniem");
})
```

Zawartość pliku *dodano.txt* zostaje zastąpiona. Znajduje się w nim tylko wartość `czarny` (rysunek 3.21).



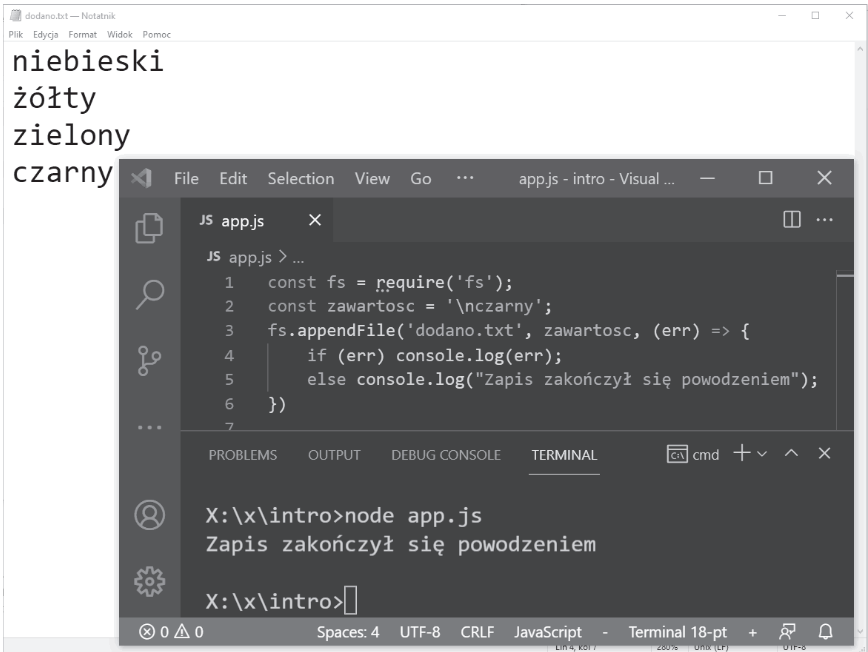
Rysunek 3.21. Zawartość pliku dodano.txt

Aby **dopisać nową wartość do pliku bez utraty bieżącej zawartości**, należy użyć metody **`.appendFile()`**. Kod na listingu 3.13 jest prawie identyczny z kodem na listingu 3.12. Zmieniła się metoda `.writeFile()`, która została zamieniona na `.appendFile()`, a przed wartością `czarny` pojawił się znak nowego wiersza, `\n`, którego użycie sprawi, że kolor zostanie dopisany pod `zielony`, a nie za.

Listing 3.13. Użycie metody .appendFile()

```
const fs = require('fs');
const zawartosc = '\nczarny';
fs.appendFile('dodano.txt', zawartosc, (err) => {
    if (err) console.log(err);
    else console.log("Zapis zakończył się powodzeniem");
})
```

Efekt wykonania kodu jest pokazany na rysunku 3.22.



Rysunek 3.22. Użycie metody .appendFile()

3.3.4. Moduł path

Moduł **path** pozwala zbudować ścieżki do plików (listing 3.14).

Pierwszy sposób wykorzystuje metodę **`path.join`** w połączeniu ze zmienną `__dirname` i nazwą pliku. Zmienna `__dirname` przechowuje ścieżkę do katalogu, w którym znajduje się plik ze skryptem; do niej zostaje doklejona nazwa pliku.

Druga metoda to użycie **konkatenacji** — aby znak odwrotnego (lewego) ukośnika (\) mógł być wyświetlony, jest poprzedzony znakiem ucieczki, którym także jest odwrotny ukośnik.

Ostatni sposób opiera się na użyciu zmiennej `__filename`, która zwraca ścieżkę do pliku ze skryptem.

Listing 3.14. Moduł path

```
const path = require('path');
const sciezkaDoPliku = path.join(__dirname, 'app.js');
const innaSciezkaDoPliku = __dirname + '\\' + 'app.js';
const jeszczeInnaSciezkaDoPliku = __filename;
console.log(sciezkaDoPliku);
console.log(innaSciezkaDoPliku);
console.log(jeszczeInnaSciezkaDoPliku);
```

Niezależnie od tego, jaka metoda zostanie użyta, wynik będzie ten sam — w konsoli zostanie wyświetlona ścieżka do pliku *app.js*, w którym znajduje się wywoływany kod, czyli `X:\intro\app.js`.

3.3.5. Własny moduł

Oprócz tego, że możemy pobierać i instalować gotowe moduły za pomocą narzędzia npm, możemy tworzyć własne, by następnie dołączać je do swoich projektów.

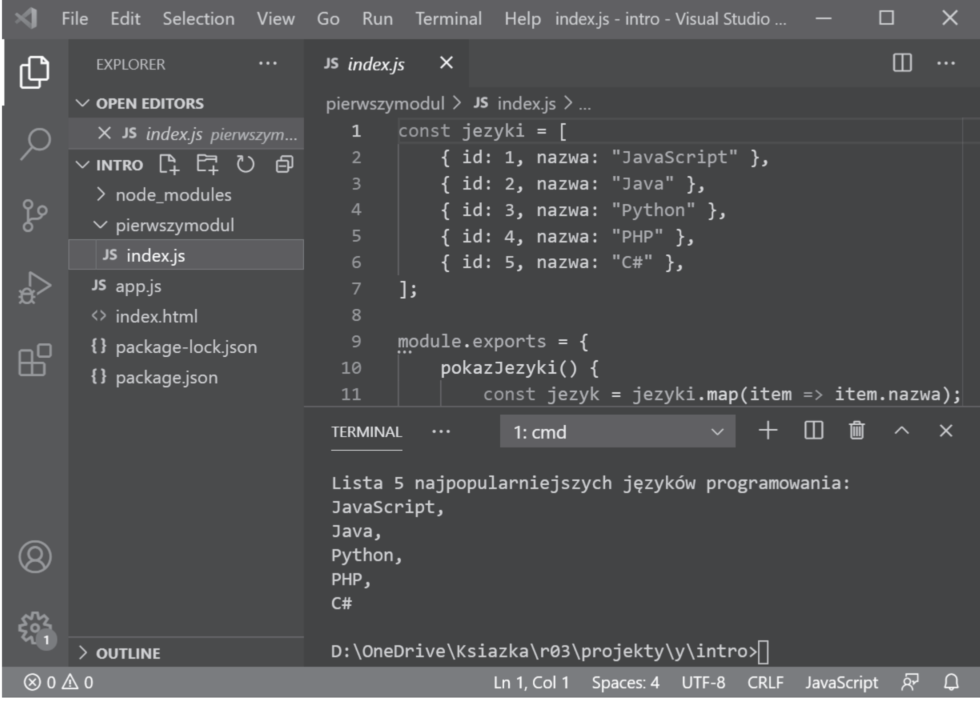
Na listingu 3.15 jest pokazany przykład prostego modułu, który ma wyświetlić listę pięciu najpopularniejszych języków programowania. W pierwszym kroku zostaje utworzona tablica obiektów `jezyki`, zawierająca nazwy języków programowania. **Aby móc odwołać się z poziomu innego pliku do metody, należy ją udostępnić.** Udostępnienie przeprowadza się z zastosowaniem `module.exports` — udostępnianą metodą jest `.pokazJezyki()`. Użycie metody `map()` na obiekcie `jezyki` tworzy tablicę zawierającą nazwy języków. Elementy tablicy utworzonej za pomocą metody `join()` są ze sobą łączone — separatorem są znaki przecinka (`,`) i nowej linii (`\n`). Metoda `.length()` zlicza zaś obiekty w tablicy.

Listing 3.15. Własny moduł

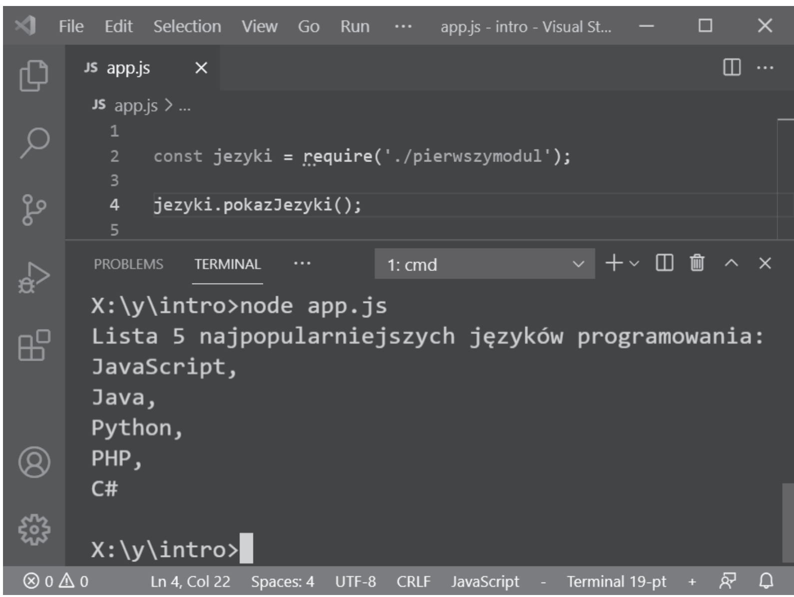
```
const jezyki = [
    { id: 1, nazwa: "JavaScript" },
    { id: 2, nazwa: "Java" },
    { id: 3, nazwa: "Python" },
    { id: 4, nazwa: "PHP" },
    { id: 5, nazwa: "C#" },
];

module.exports = {
    pokazJezyki() {
        const jezyk = jezyki.map(item => item.nazwa);
        const liczbaJezykow = jezyki.length;
        console.log(`Lista ${liczbaJezykow} najpopularniejszych języków
programowania:\n${jezyk.join(',\n')}`);
    }
}
```

Kod modułu musi być umieszczony w pliku *index.js*, a ten zostaje zapisany w katalogu *pierwszymodul* (rysunek 3.23).

Aby w pliku projektu, np. *app.js*, wykorzystać moduł, należy go zaimportować — zostaje podana ścieżka do katalogu *pierwszymodul*. Po przeprowadzeniu importu możliwe jest użycie wyeksportowanej metody `.pokazJezyki()` (rysunek 3.24).



Rysunek 3.23. Eksportowana metoda — plik index.js



Rysunek 3.24. Użycie wyeksportowanej metody

Dozwolone jest również umieszczenie kodu w pliku o innej nazwie niż *index.js*, ale ta informacja musi być umieszczona w pliku *package.json* modułu. W tym celu należy utworzyć plik i w sekcji `main` zdefiniować nazwę pliku, w którym znajdują się eksportowane metody.

WSKAZÓWKA

Przedstawione zasady obowiązują także w przypadku modułów zewnętrznych instalowanych za pomocą narzędzia npm.

Zadanie 3.3.

Napisz prosty skrypt, który wykorzysta dwa z opisanych modułów.

Zadanie 3.4.

Napisz moduł, którego zadaniem będzie zamiana tekstu napisanego małymi literami na tekst napisany dużymi literami.

Pytania kontrolne

1. Jaką rolę w Node.js odgrywa moduł?
2. Wymień podstawowe moduły dostępne w Node.js i omów ich przeznaczenie.
3. Co trzeba zrobić, żeby w Node.js skorzystać z funkcjonalności danego modułu?

3.4. Serwer HTTP — Node.js

Opisanych modułów można użyć do zbudowania serwera HTTP. Zalążek kodu został już pokazany na listingu 3.4. Dzięki wiedzy przedstawionej na poprzednich stronach podręcznika możemy go rozbudować.

Zadaniem serwera WWW będzie wyświetlenie strony zapisanej w pliku *index.html*, załadowanie pliku arkusza stylów (*style.css*) oraz uruchomienie kodu w JavaScripcie zapisanego w zewnętrznym pliku *script.js*. Dodatkowo zostanie przesłana ikona *favicon.ico*.

WSKAZÓWKA

Favicon to niewielka ikona o rozmiarach 16×16 px, 32×32 px lub 48×48 px, która pojawia się przed adresem w polu adresowym przeglądarki, na karcie w przeglądarce internetowej oraz w zakładce ulubionych stron. Jej użycie pozwala wyróżnić stronę spośród wielu otwartych kart w przeglądarce. Grafikę przygotujemy w dowolnym programie graficznym, np. GIMP, lub za pomocą generatora online, np. *https://favicon.io/* lub *https://www.favicon-generator.org/*.

Zawartość wszystkich plików przedstawiono poniżej: listing 3.16 — plik *index.html*; listing 3.17 — plik *style.css*; listing 3.18 — plik *script.js*.

Listing 3.16. Plik index.html

```
<html>

<head>
    <title>Node.js</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <div id="container">
        <div id="strona">
            <p>Zawartość pliku ./index.html</p>
        </div>
    </div>
    <script src="script.js"></script>
</body>

</html>
```

Listing 3.17. Plik style.css

```
#container {
    width: 800px;
    margin: 0 auto;
}

#strona {
    font-size: 150%;
    text-align: center;
    padding: 20px;
    border: 1px solid black;
    background-color: yellow;
}
```

Listing 3.18. Plik script.js

```
alert('Skrypt załadowany poprawnie');
```

Kod skryptu uruchamiającego serwer WWW jest pokazany na listingu 3.19.

Listing 3.19. Serwer HTTP — Node.js, instrukcja if

```
// Deklaracja metod
const http = require('http');
const path = require('path');
const fs = require('fs');

// Deklaracja adresu serwera
const host = '127.0.0.1';
const port = 5555;

function odpowiedz(req, res) {

    // Deklaracja ścieżek do wczytywanych plików
    const plik = path.join(__dirname, 'index.html');
    const css = path.join(__dirname, 'style.css');
    const script = path.join(__dirname, 'script.js');
    const favicon = path.join(__dirname, 'favicon.ico');

    // Wczytanie strony głównej — plik index.html
    if (req.url === '/') {
        fs.readFile(plik, (err, dane) => {
            if (!err) {
                res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8'
});
                res.end(`${dane}`);
                console.log(`Otwarto stronę ${req.url}`)
            } else {
                res.writeHead(404, { 'Content-Type': 'text/html' });
                console.dir(err);
                res.end(`<h3>Strona o podanym adresie nie istnieje</h3>`);
            }
        });
    }

    // Wczytanie arkusza CSS — plik style.css
    if (req.url === '/style.css') {
        fs.readFile(css, (err, dane) => {
            if (!err) {
                res.writeHead(200, { 'Content-Type': 'text/css; charset=utf-8'
});
                res.end(`${dane}`);
                console.log(`Załadowano: ${req.url}`);
```

```
            } else {
                res.writeHead(404, { 'Content-Type': 'text/html' });
                console.dir(`Nie wczytano pliku ${css}`);
                res.end();
            }
        });
    }

    // Wczytanie pliku skryptu JS — plik script.js
    if (req.url === '/script.js') {
        fs.readFile(script, (err, dane) => {
            if (!err) {
                res.writeHead(200, { 'Content-Type': 'text/javascript;
charset=utf-8' });
                res.end(`${dane}`);
                console.log(`Załadowano: ${req.url}`);
            } else {
                res.writeHead(404, { 'Content-Type': 'text/html' });
                console.dir(`Nie wczytano pliku ${script}`);
                res.end();
            }
        });
    }

    // Wczytanie ikony favicon — plik favicon.ico
    if (req.url === '/favicon.ico') {
        fs.readFile(favicon, (err, dane) => {
            if (err) {
                res.writeHead(404, { 'Content-Type': 'text/html' });
                console.log(`Błąd - nie wczytano zasobu:${favicon}`);
                res.end();
            }
            else {
                res.writeHead(200, { 'Content-Type': 'image/x-icon' });
                console.log(`Załadowano: ${favicon}`);
                res.end(dane);
            }
        });
    }
}

// Uruchomienie serwera
const serwerWWW = http.createServer(odpowiedz);
serwerWWW.listen(port, host, () => console.log(`Serwer WWW działa. Jego adres
to: ${host}:${port}`));
```

Aby można było wyświetlić **zawartość plików**, trzeba zaimportować moduł `fs` — moduł ten zostaje przypisany do zmiennej `fs` (dobrą praktyką jest użycie nazwy zmiennej takiej samej jak nazwa importowanego modułu). Innym wymaganym modułem jest `http`. Dodatkowy moduł `path` został wykorzystany do zbudowania ścieżek do plików.

Użyty **numer portu** i **adres IP** serwera (127.0.0.1 to inaczej *localhost*) zostają powiązane ze zmiennymi `port` i `host`.

Ścieżki do plików są określone za pomocą metody `.join()` i modułu `path`. Ponieważ wszystkie pliki znajdują się we wspólnym katalogu, do utworzenia ścieżki zastosowano zmienną `__dirname`, która wskazuje na katalog zawierający skrypt serwera. Tę zmienną połączono z nazwą przekazywanych plików. Ścieżki do użytych plików zostają przypisane do stałych `plik`, `css`, `script` i `favicon`.

Do odczytania zawartości plików zostały użyte cztery **instrukcje** **`if`** (po jednej dla każdego zasobu). Warunkiem odczytania danego pliku jest **pojawienie się żądania** jego udostępnienia. Żądania są inicjowane po wywołaniu adresu serwera. Pierwsze żądanie dotyczy otwarcia strony głównej, drugie — pobrania ikony favicon, a trzecie i czwarte — pobrania pliku arkusza stylów i skryptu (odnośniki wywołujące żądania są umieszczone w kodzie HTML). Pojawiające się żądania są sprawdzane za pomocą metody `.url()`. Przykładowe żądanie `/style.css` (znak `/` określa katalog główny witryny) spowoduje otwarcie pliku *style.css* (nazwa pliku i żądania nie muszą być tożsame, np. żądanie `/style.css` może otwierać plik *mojArkuszCSS.css*).

Za **odczytanie pliku** odpowiada metoda `.readFile()`. Jeśli plik istnieje, za pomocą metody `.writeHead()` zostaje wysłany nagłówek ze statusem `200` (OK), a przekazywane dane to kod HTML (mówi o tym zapis `'Content-Type': 'text/html'`). Metoda **`.end()`** wysyła do przeglądarki odczytaną zawartość pliku i jednocześnie sygnalizuje serwerowi, że wszystkie nagłówki i treść zostały przekazane (metoda musi kończyć każdą odpowiedź). Alternatywnym sposobem wysłania odpowiedzi jest użycie metody **`.write()`** zakończonej wywołaniem metody `.end()`. Dodatkowo powodzenie operacji jest sygnalizowane komunikatem umieszczonym w konsoli.

WSKAZÓWKA

Content-Type (nazywany także typem *MIME*) to dwuczęściowy nagłówek HTTP formatu plików w internecie. Określa, jaki typ danych zostanie przekazany. Do najczęściej wykorzystywanych należą:

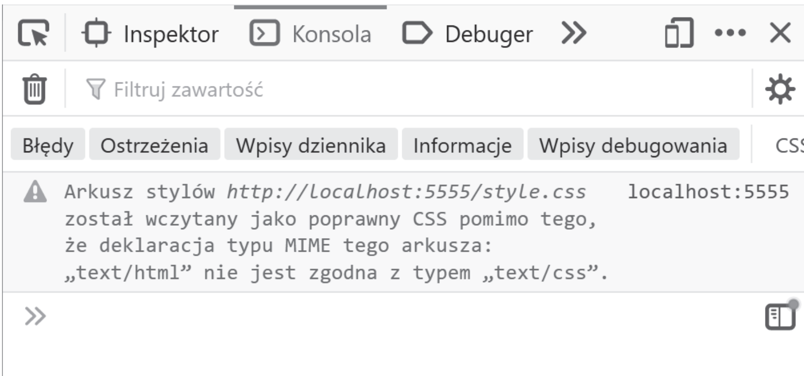
- `text/css` — arkusz stylów CSS;
- `text/html` — kod HTML;
- `text/javascript` (dawniej też `application/javascript`) — kod skryptu języka JavaScript;
- `text/plain` — dane tekstowe;

→

WSKAZÓWKA — ciąg dalszy

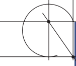
- `image/gif`, `image/jpeg` oraz `image/png` — obrazy w formatach (odpowiednio) GIF, JPEG i PNG;
- `audio/mpeg` — plik audio w formacie MP3 lub MPEG;
- `video/mp4` — plik wideo w formacie MP4;
- `application/json` — plik w formacie JSON.

Niepoprawnie określony parametr `Content-Type` spowoduje wyświetlenie w konsoli przeglądarki komunikatu o błędzie (rysunek 3.25).



Rysunek 3.25. Niepoprawnie określony typ przekazywanych danych

W razie niepowodzenia zostanie wysłany nagłówek `404` („nie znaleziono”), a w przeglądarce wyświetli się informacja *Strona o podanym adresie nie istnieje*. W konsoli zostanie również wyświetlony status błędu.



WSKAZÓWKA

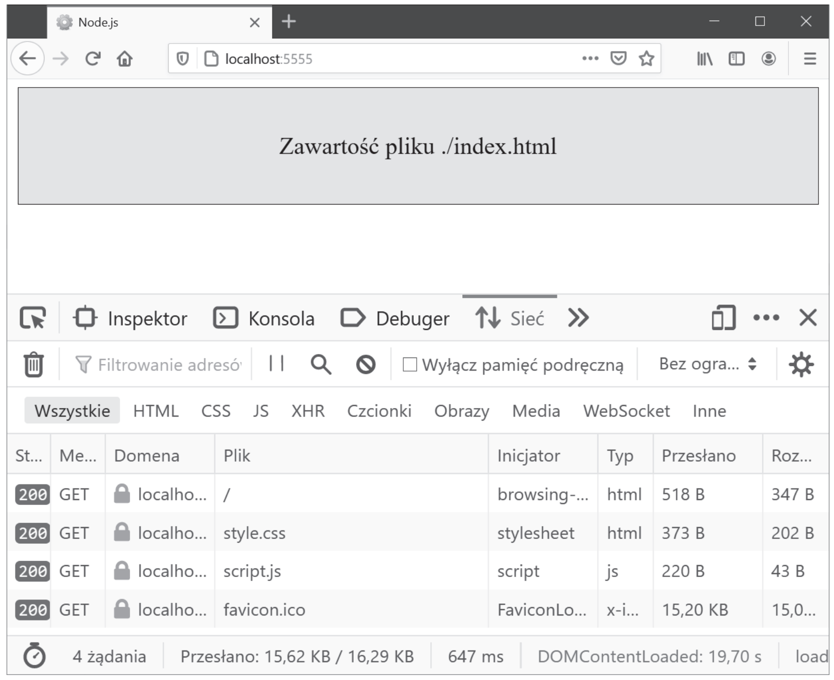
Status operacji jest zwracany w postaci trzycyfrowej liczby, np. **poprawne wczytanie strony** ma status `200`. Dodatkowo do **zasobów przeniesionych na inny serwer** stosuje się statusy 3xx: `301` (*Moved Permanently* — zasób dostępny pod tym adresem został na stałe przeniesiony pod inny adres) oraz `302` (*Found* — zasób został przeniesiony tymczasowo). **Statusy błędów** rozpoczynają się od 4. Najczęściej spotykane to: `400` (*Bad Request* — błędne zapytanie), `401` (*Unauthorized* — wymagana autoryzacja), `403` (*Forbidden* — dostęp zabroniony) oraz `404` (*Not Found* — na serwerze nie znaleziono pliku). Statusy rozpoczynające się od 5 oznaczają **błąd po stronie serwera**: `500` (*Internal Server Error* — wewnętrzny błąd serwera), `503` (*Service Unavailable* — przeciążenie lub brak dostępności serwera) oraz `504` (*Gateway Time-out* — przekroczenie czasu oczekiwania na odpowiedź).

Na podobnych zasadach wysyłane są pozostałe pliki. Różnica dotyczy jedynie sposobu informowania przeglądarki o typie przekazywanych treści.

Wszystkie instrukcje `if` zostają opakowane w jedną funkcję (`odpowiedz`), przekazywaną do metody `.createServer()`. Serwer uruchamia metoda **`.listen()`**, której zostają przekazane jako argumenty zmienne `port` i `host`.

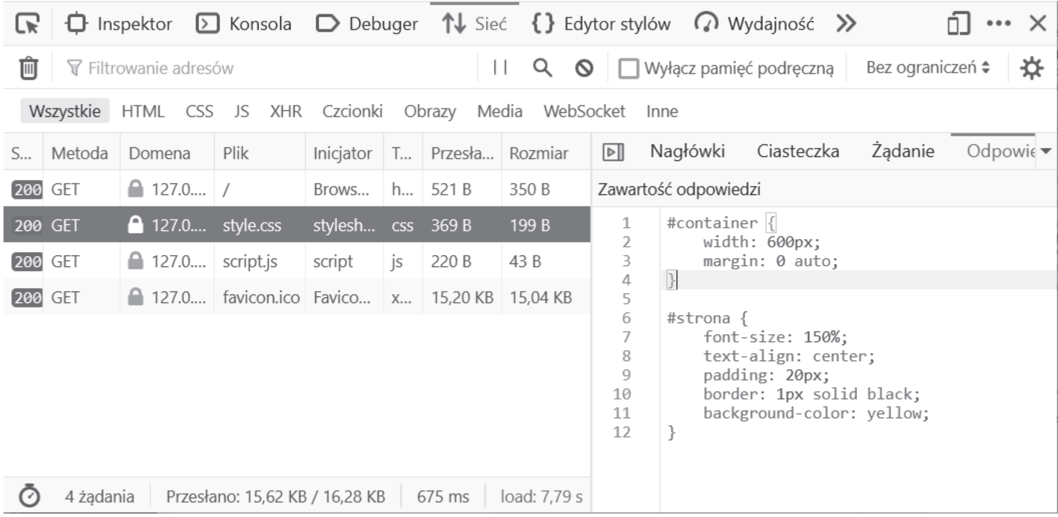
Efekt działania skryptu jest pokazany na rysunku 3.26.

Jak można zauważyć, wszystkie pliki zostały załadowane poprawnie (zakładka *Sieć*, kolumna *Stan*, status `200`). Typ otrzymanych danych (kolumna *Typ*) jest zgodny z oczekiwaniami. Zawartość pliku *index.html* jest formatowana za pomocą dołączonego arkusza stylów, a skrypt zapisany w zewnętrznym pliku uruchamia się poprawnie. Przy tytule strony widnieje favicon.



Rysunek 3.26. Serwer WWW — Node.js (z użyciem instrukcji if)

Dane, które otrzymała przeglądarka, możemy sprawdzić. W tym celu należy kliknąć żądanie, a następnie w nowo otwartym oknie wybrać *Odpowiedź*. Wówczas zostanie pokazana zawartość pliku (rysunek 3.27 — na przykładzie przeglądarki Firefox).



Rysunek 3.27. Sprawdzenie danych przesłanych do przeglądarki

Przedstawiony na listingu 3.19 kod serwera WWW można zapisać z użyciem **instrukcji wielokrotnego wyboru** `switch case`. Zmodyfikowany kod jest pokazany na listingu 3.20. Robi to samo co poprzedni, lecz dla niektórych osób to rozwiązanie będzie bardziej intuicyjne i prostsze do zrozumienia.

W stosunku do wcześniejszego rozwiązania zmienił się sposób deklarowania ścieżek do plików — bez użycia modułu `path`. Zamiast tego wykorzystano konkatenację — zmienna `__dirname` została połączona z nazwą pliku za pomocą znaku plusa (+).

Do rozpoznania żądania ponownie użyto metody `.url()`, lecz tym razem powiązano ją z instrukcją `switch`. Za uruchomienie kodu odpowiada instrukcja `case`.

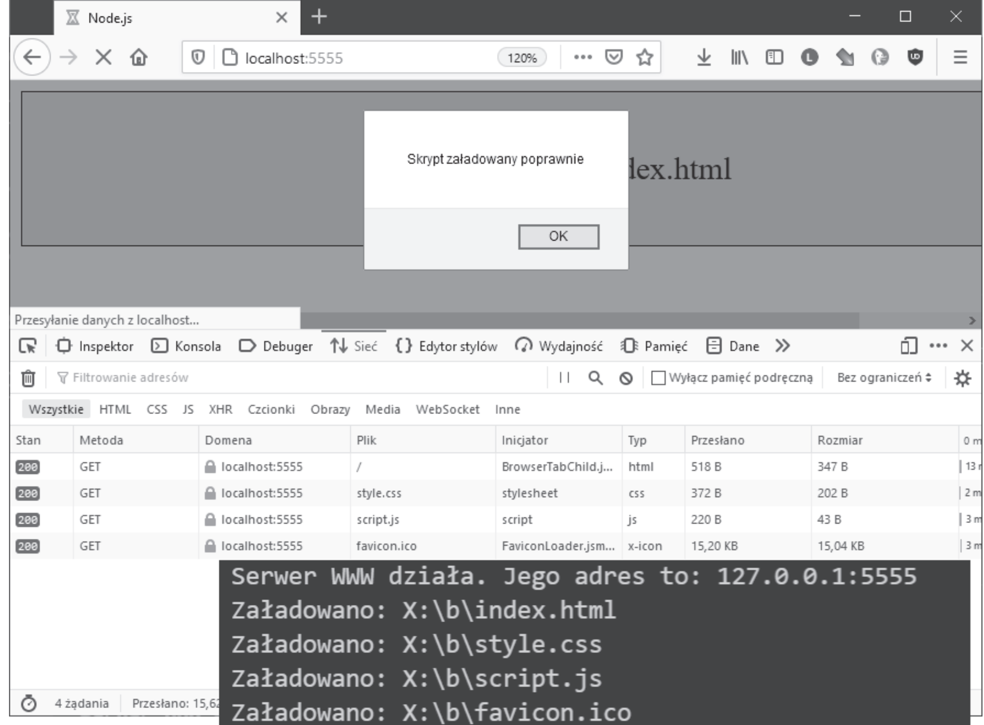
Listing 3.20. Serwer HTTP — Node.js, instrukcja switch case

```
// Deklaracja metod
const http = require('http');
const fs = require('fs');

const host = '127.0.0.1';
const port = 5555;

// Deklaracja ścieżek do wczytywanych plików
const plik = __dirname + '\\index.html';
const css = __dirname + '\\style.css';
const script = __dirname + '\\script.js';
const favicon = __dirname + '\\favicon.ico';
```

```
function odpowiedz(req, res) {
    switch (req.url) {
        // Wczytanie strony głównej — plik index.html
        case '/':
            fs.readFile(plik, (err, dane) => {
                if (err) {
                    res.writeHead(404, { 'Content-Type': 'text/css;
charset=utf-8' });
                    console.log(`Błąd - nie wczytano zasobu:${plik}`);
                    res.end();
                }
                else {
                    res.writeHead(200, { 'Content-Type': 'text/html;
charset=utf-8' });
                    res.end(dane);
                    console.log(`Załadowano: ${plik}`);
                }
            });
            break;
        // Wczytanie arkusza CSS — plik style.css
        case '/style.css':
            fs.readFile(css, (err, dane) => {
                if (err) {
                    res.writeHead(404, { 'Content-Type': 'text/css;
charset=utf-8' });
                    console.log(`Błąd - nie wczytano zasobu:${css}`);
                    res.end();
                }
                else {
                    res.writeHead(200, { 'Content-Type': 'text/css;
charset=utf-8' });
                    res.end(dane);
                    console.log(`Załadowano: ${css}`);
                }
            });
            break;
        // Wczytanie pliku skryptu JS — plik script.js
        case '/script.js':
            fs.readFile(script, (err, dane) => {
                if (err) {
                    res.writeHead(404, { 'Content-Type': 'text/css;
charset=utf-8' });
                    console.log(`Błąd - nie wczytano zasobu:${script}`);
                    res.end();
                }
```

```
                else {
                    res.writeHead(200, { 'Content-Type': 'text/javascript;
charset=utf-8' });
                    res.end(dane);
                    console.log(`Załadowano: ${script}`);
                }
            });
            break;
        // Wczytanie ikony favicon — plik favicon.ico
        case '/favicon.ico':
            fs.readFile(favicon, (err, dane) => {
                if (err) {
                    res.writeHead(404, { 'Content-Type': 'text/css;
charset=utf-8' });
                    console.log(`Błąd - nie wczytano zasobu:${favicon}`);
                    res.end();
                }
                else {
                    res.writeHead(200, { 'Content-Type': 'image/x-icon' });
                    res.end(dane);
                    console.log(`Załadowano: ${favicon}`);
                }
            });
            break;
        // Strona domyślna
        default:
            res.writeHead(404, { 'Content-Type': 'text/html; charset=utf-8' });
            res.end('<h2>Brak zasobu</h2>');
    }
}

// Uruchomienie serwera
const serwerWWW = http.createServer(odpowiedz);
serwerWWW.listen(port, host, () => console.log(`Serwer WWW działa. Jego adres
to: ${host}:${port}`));
```

Kod działa poprawnie. Wszystkie elementy zostają przesłane przez serwer do przeglądarki, a w konsoli serwera pojawia się seria komunikatów informujących o poprawnym wczytaniu plików (rysunek 3.28).



Rysunek 3.28. Serwer WWW — Node.js (z użyciem instrukcji switch case)

Niezależnie od użytego sposobu brak żądanego pliku spowoduje, że zostanie wygenerowany i wyświetlony w konsoli komunikat o błędzie, co pokazano na rysunku 3.29 (brak pliku arkusza stylów — *style.css*).

UWAGA

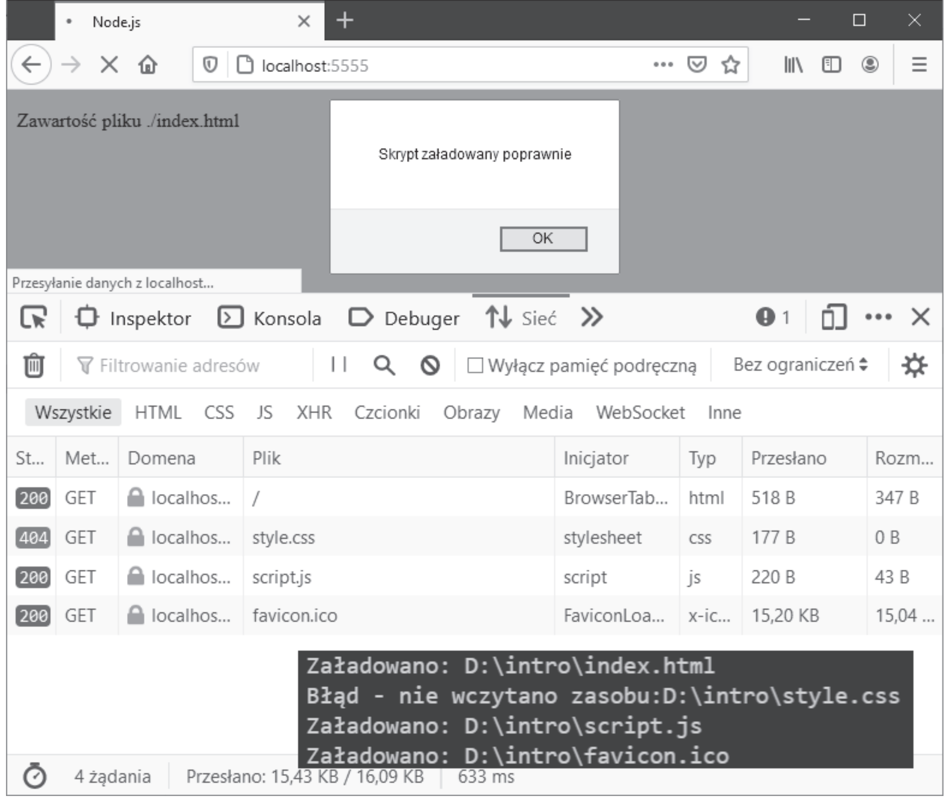
Brak pliku głównego *index.html* nie wywoła żądania załadowania arkusza stylów i skryptu, ponieważ odnośniki inicjujące żądanie znajdują się w kodzie HTML.

Zadanie 3.5.

Zmodyfikuj kod serwera HTTP tak, aby można go było użyć z innym zestawem plików.

Zadanie 3.6.

Sprawdź, jak zachowa się przeglądarka, gdy serwer odeśle nagłówek z numerem statusu innym niż `200`.



Rysunek 3.29. Błąd — brak pliku arkusza stylów

Zadanie 3.7.

Sprawdź, co się stanie, gdy dla pliku *index.html* definicja `Content-Type` zostanie zmieniona na `text/plain`.

Zadanie 3.8.

Do pliku *index.html* dodaj kod, który wyświetli dowolne zdjęcie. Zmodyfikuj kod serwera tak, aby można było je wyświetlić.

Pytania kontrolne

1. Za co odpowiadają metody `.createServer()` i `.listen()`?
2. Jak definiowany jest status operacji? Omów podstawowe numery statusów.
3. Co się stanie, jeśli w kodzie serwera pominiemy metodę `.end()`?

3.5. Framework Express

Express to jeden z najpopularniejszych frameworków webowych. Jest również wykorzystywany jako biblioteka, w tym w Node.js.

3.5.1. Serwer HTTP — Express

Aby zainstalować framework Express, należy się posłużyć poleceniem `npm install express --save` (alternatywną opcją jest `npm i express -S`).

UWAGA

Tworzony projekt nie może się nazywać *express*. Wybranie takiej nazwy uniemożliwiłoby instalację frameworka i spowodowałoby wyświetlenie komunikatu: `27 error Refusing to install package with name "express" under a package also called "express". Did you name your project the same as the dependency you're installing?`.

Aby móc rozpocząć pracę z frameworkiem, w pierwszym kroku należy do projektu zaimportować moduł `express`. Zrobimy to za pomocą funkcji `require()`: `const express = require('express');`.

Drugim krokiem jest wywołanie frameworka. Należy je powiązać ze zmienną/stałą, która będzie reprezentować aplikację tworzoną w Expressie: `const app = express();`.

Do uruchomienia serwera WWW służy metoda `.listen()`. Należy do niej jako argument przekazać numer portu (argument `hostname` jest opcjonalny), pod którym usługa będzie nasłuchiwać:

```
app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port}`);
});
```

Dodatkowa funkcja wyświetli w konsoli adres i port serwera HTTP.

Ostatnią czynnością jest zdefiniowanie odpowiedzi według wzorca:

```
<nazwa_aplikacji>.<metoda_HTTP>('<adres_URL>', () => {})
```

gdzie:

- **`nazwa_aplikacji`** to nazwa aplikacji — nazwa zmiennej, z którą powiązano framework Express;
- **`metoda_HTTP`** to metoda protokołu HTTP;
- **`adres_URL`** to adres URL, który zostanie wpisany przez klienta.

Odpowiedź jest generowana przez następujący kod:

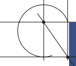
```
app.get('/', (req, res) => {
    res.send('<h1>Witaj, Express!</h1>');
    console.log(`Otwarto stronę główną z adresu ${req.ip}`);
});
```

Dla obiektu `app`, reprezentującego aplikację, zdefiniowano ścieżkę `/`, która prowadzi do katalogu głównego witryny. W momencie wywołania przez przeglądarkę żądania *GET* (otwarcie strony głównej, pobranie zasobu) uruchomi się funkcja, która odeśle odpowiedź.

WSKAZÓWKA

Express został przygotowany z myślą o obsłudze wielu metod protokołu *HTTP*. Oprócz *GET* można użyć metod *POST* (odpowiedzialnej m.in. za przekazanie formularza), *PUT* (wysłanie pliku) czy *DELETE* (usunięcie zasobu).

Podobnie jak w przypadku Node.js, za realizację żądań i odpowiedzi odpowiadają obiekty `request` i `response` (często deklarowane za pomocą form skróconych: `req` i `res`). Metoda `.send()` odsyła odpowiedź do klienta.



UWAGA

Metody omówione wcześniej, takie jak `.write()` i `.end()`, są również dostępne w Expressie.

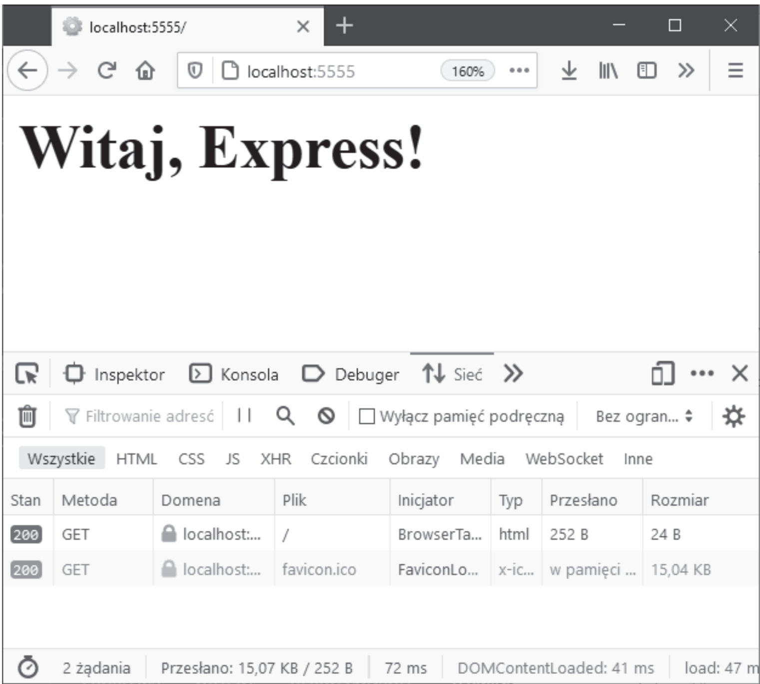
Wykorzystanie metody `.send()` jest o wiele wygodniejsze od przedstawionych do tej pory metod — `.write()` i `.end()`. Jej użycie sprawia, że nie trzeba ustawiać nagłówka `Content-Type`. Nagłówek **jest ustawiany automatycznie** w zależności od typu wysyłanych danych: przesłanie kodu HTML ustawi nagłówek na `text/html`, a przesłanie pliku arkusza stylów — na `text/css`. Dodatkowo metoda ta wykorzystuje *caching* oraz potrafi konwertować dane.

Całość kodu serwera WWW jest pokazana na listingu 3.21.

Listing 3.21. Serwer WWW — Express

```
const express = require('express');
const app = express();
const port = 5555;

app.get('/', (req, res) => {
    res.send('<h1>Witaj, Express!</h1>');
});

app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port}`);
});
```

Działanie kodu jest pokazane na rysunku 3.30. Jak można zauważyć, użycie metody `.send()` nie spowodowało błędu przedstawionego na rysunku 3.25 — metoda `.send()` automatycznie wykryła typ przesyłanych danych i ustawiła nagłówek na `text/html`.



Rysunek 3.30. Serwer WWW oparty na frameworku Express

Gdy już masz opanowane podstawy frameworka Express, możemy za jego pomocą zbudować serwer WWW, którego funkcjonalność nie będzie odbiegała od funkcjonalności serwera przedstawionego na listingach 3.19 i 3.20.

Kod serwera WWW wykorzystującego framework Express został pokazany na listingu 3.22.

Deklaracja metod i ścieżek przebiega identycznie jak na listingach 3.19 i 3.20.

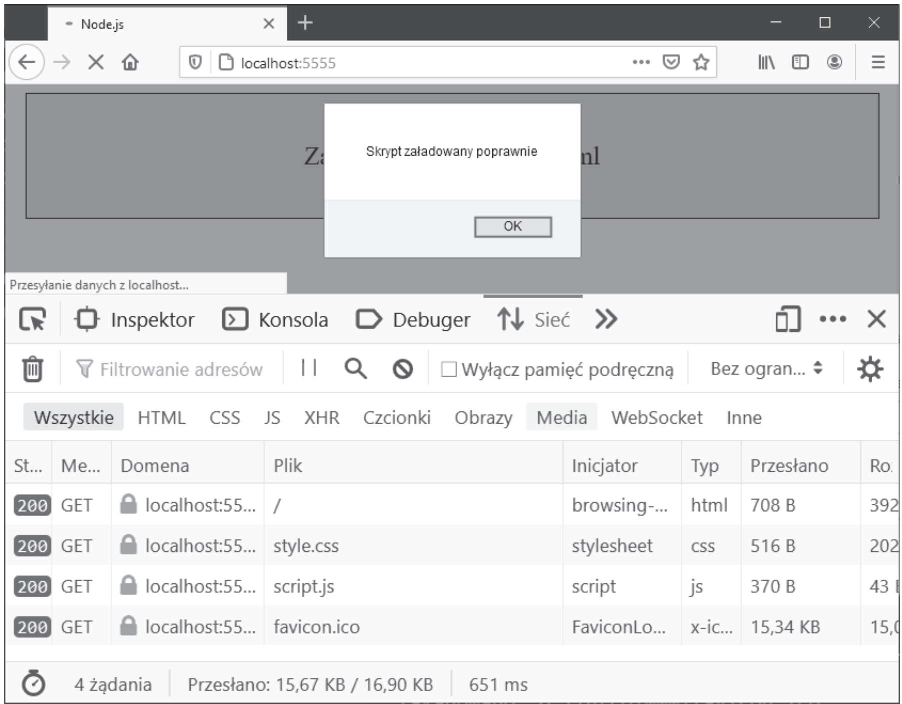
Za wysłanie pliku odpowiada metoda `.sendFile()` — nie trzeba już importować modułu `fs`. Wysłanie pliku jest odpowiedzią serwera, dlatego musi być ona powiązana z obiektem `response`.

Listing 3.22. Serwer HTTP — Express

```
// Deklaracja metod
const express = require('express');

const app = express();
const port = 5555;
```

```
// Deklaracja ścieżek do wczytywanych plików
const plik = __dirname + '\\www\\index.html';
const css = 'style.css';
const script = __dirname + '\\www\\script.js';
const favicon = __dirname + '\\www\\favicon.ico';

// Wczytanie strony głównej — plik index.html
app.get('/', (req, res) => {
    res.sendFile(plik);
    console.log(`Załadowano: ${plik}`);
    console.log(`Użycie .protocol(): ${req.protocol}`);
    console.log(`Użycie .secure(): ${req.secure} \n`);
});

// Wczytanie arkusza CSS — plik style.css
app.get('/style.css', (req, res) => {
    res.sendFile(css, { root: __dirname + '\\www' });
    console.log(`Załadowano: ${css} \n`);
});

// Wczytanie pliku skryptu JS — plik script.js
app.get('/script.js', (req, res) => {
    res.sendFile(script);
    console.log(`Załadowano: ${script}`);
    console.log(`Użycie .url(): ${req.url}`);
    console.log(`Użycie .originalUrl(): ${req.originalUrl}`);
    console.log(`Użycie .path(): ${req.path} \n`);
    console.log(`Pełen adres URL: ${req.protocol + "://" + req.get('host') +
req.originalUrl} \n`);
});

// Wczytanie ikony favicon — plik favicon.ico
app.get('/favicon.ico', (req, res) => {
    res.sendFile(favicon);
    console.log(`Załadowano: ${favicon}`);
    console.log(`Otwarto stronę z adresu ${req.ip} \n`);
});

// Uruchomienie serwera
app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port} \n`);
});
```

Działanie skryptu jest zilustrowane na rysunku 3.31. Jak można zauważyć, wszystkie pliki zostały odebrane przez przeglądarkę (status `200`), a nagłówek został poprawnie ustawiony w zależności od przesłanych danych.



Rysunek 3.31. Serwer WWW oparty na frameworku Express

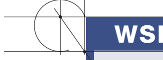
Definicja wczytania strony głównej (plik *index.html*) zawiera dwie właściwości, `.protocol()` i `.secure()`, które odpowiadają za detekcję protokołu. Użycie **`.protocol()`** w przypadku zastosowania nieszyfrowanej wersji protokołu wyświetli `http`, a użycie wersji szyfrowanej spowoduje zwrócenie `https`. Właściwość **`.secure()`** zwraca zaś `false`, gdy wykorzystanym protokołem jest HTTP, lub `true`, gdy jest to HTTPS (rysunek 3.32). Obie mogą zostać zastosowane do przekierowania adresu.

W sekcji kodu odpowiedzialnego za wczytanie pliku *script.js* zostały umieszczone trzy właściwości (w kolejności: `.url()`, `.originalUrl()` oraz `.path()`). Wszystkie one przechowują informacje na temat ścieżki adresu URL (rysunek 3.32). Pierwsza nie jest częścią frameworka Express i pochodzi bezpośrednio z Node.js. Użycie drugiej jest zalecanym sposobem pozyskiwania ścieżki — zachowuje oryginalną informację o ścieżce nawet w przypadku przekierowania. Trzecia przechowuje tylko ostatnią część adresu URL.

Właściwość **`.ip()`** zawiera informację o adresie IP (wersja 6 i 4 protokołu) hosta (klienta), z którego przyszło zapytanie do serwera (rysunek 3.32).

```
Pełen adres URL: http://localhost:5555/script.js

Załadowano: D:\intro\www\favicon.ico
Otwarto stronę z adresu ::ffff:127.0.0.1

Załadowano: D:\intro\www\index.html
Użycie .protocol(): http
Użycie .secure(): false

Załadowano: style.css

Załadowano: D:\intro\www\script.js
Użycie .url(): /script.js
Użycie .originalUrl(): /script.js
Użycie .path(): /script.js

Pełen adres URL: http://localhost:5555/script.js

Załadowano: D:\intro\www\favicon.ico
Otwarto stronę z adresu ::ffff:127.0.0.1
```

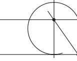
Rysunek 3.32. Zdefiniowane w kodzie serwera logi serwera, które są wyświetlane w konsoli



WSKAZÓWKA

Użycie `.ips()` pozwoli pozyskać informację o adresie IP klienta w sytuacji, w której użyto proxy (serwera pośredniczącego). Zmienna ta jest tablicą, ponieważ serwerów pośredniczących może być więcej niż jeden.

Za odczytanie i przesłanie pliku odpowiada metoda `.sendFile()`. Podobnie jak `.send()`, ustawia ona automatycznie nagłówek w zależności od typu przesyłanych plików (nagłówki można również ustawić ręcznie — należy w tym celu posłużyć się metodą `.set()` dostępną w obiekcie `response`). Jednak w przypadku pliku *style.css* użyty kod odbiega od pozostałych — zastosowano parametr `root`. Blokuje on wyjście poza zdefiniowany katalog. Ponieważ ścieżka do katalogu jest definiowana przy wywołaniu parametru, nie trzeba określać jej ponownie — podajemy tylko nazwę pliku (bądź ścieżkę do pliku, ale punktem wyjścia jest użyty katalog).



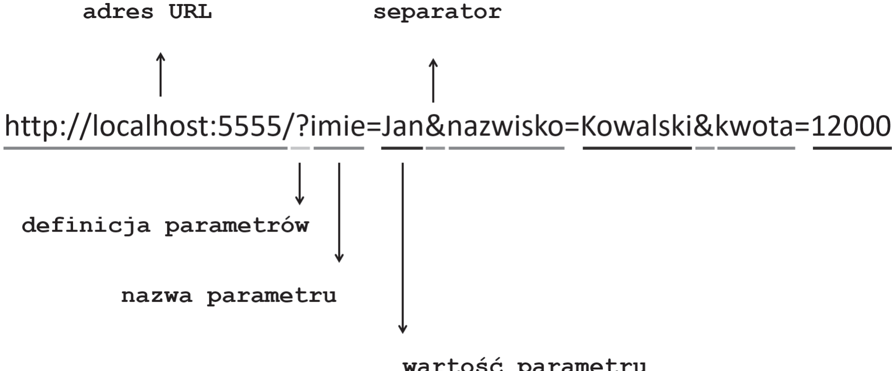
WSKAZÓWKA

Do przekazania pliku można również wykorzystać metodę **`.attachment()`** — podanie nazwy pliku jako argumentu spowoduje, że przeglądarka nie wyświetli jego zawartości, lecz zaproponuje zapisanie. Po użyciu tej metody wymagane jest zakończenie połączenia — należy dodać `res.end()`.

3.5.2. Przesyłanie informacji i mechanizm cookies

Przesyłanie informacji

Tworzenie witryn opartych na Node.js i Expressie będzie wymagać przekazania informacji pochodzącej ze strony (frontend) do serwera (backend). W tym celu możemy wykorzystać odnośnik, który przekaże parametry wraz z wartościami (rysunek 3.33).



Rysunek 3.33. Parametry adresu URL

Użycie na przykład odnośnika *http://localhost:5555/?imie=Jan&nazwisko=Kowalski&kwota=12000* sprawi, że serwer do zmiennej `imie` przypisze wartość `Jan`, do zmiennej `nazwisko` — wartość `Kowalski`, a do zmiennej `kwota` — wartość `12000`. Wszystkie one zostaną wykorzystane do wyświetlenia komunikatu w konsoli (rysunek 3.34). Oczywiście odnośnik ten jest budowany (generowany) na podstawie danych pochodzących z innych źródeł, na przykład z formularza.

Aby serwer mógł odebrać i przetworzyć informacje z przesłanego odnośnika, należy posłużyć się obiektem **`.query`**. Użycie go sprawi, że parametry z adresu URL staną się jego **kluczami**, do których zostaną przypisane **wartości**.

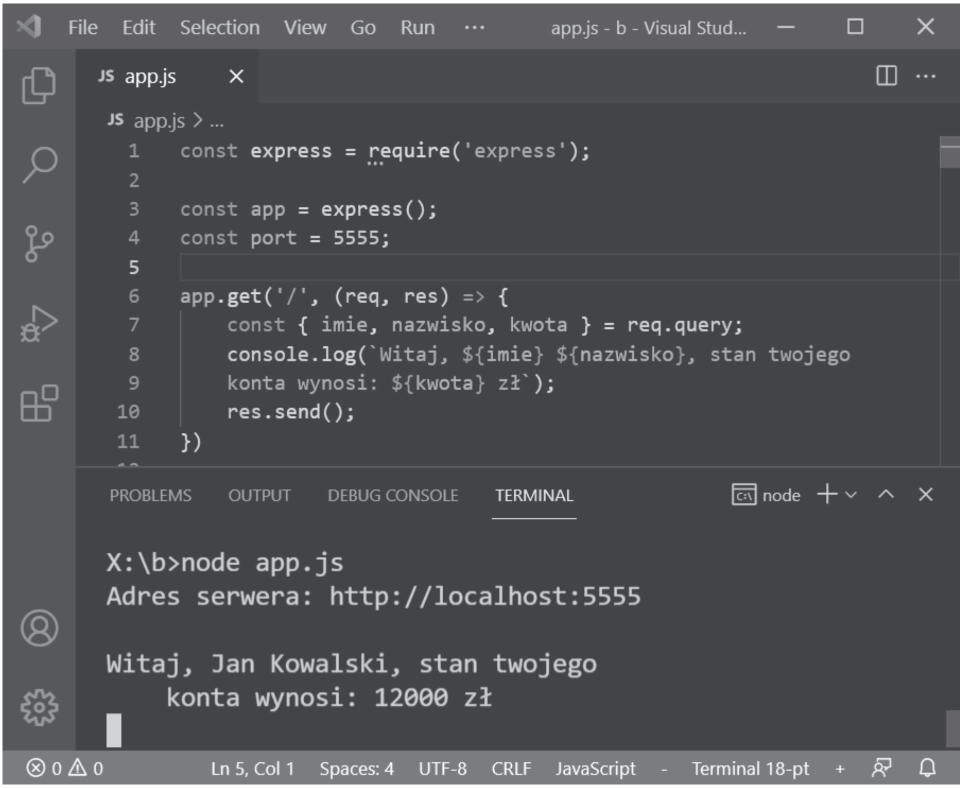
Kod przedstawiony na listingu 3.23 zinterpretuje pokazany wyżej odnośnik i wyświetli go w konsoli.

Listing 3.23. Użycie obiektu .query

```
const express = require('express');

const app = express();
const port = 5555;

app.get('/', (req, res) => {
    const { imie, nazwisko, kwota } = req.query;
    console.log(`Witaj, ${imie} ${nazwisko}, stan twojego konta wynosi:
${kwota} zł`);
```

```
    res.send();
})

app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port} \n`);
});
```

Działanie kodu jest pokazane na rysunku 3.34.



Rysunek 3.34. Obiekt query

W kodzie wykorzystano mechanizm **destrukturyzacji**. Aby odczytać wartości kluczy obiektu `.query` i przypisać je do stałych, moglibyśmy posłużyć się kodem:

```
const imie = req.query.imie;
const nazwisko = req.query.nazwisko;
const kwota = req.query.kwota;
```

Użycie destrukturyzacji sprawia jednak, że wystarczy napisać (nazwa zmiennej musi być tożsama z nazwą klucza):

```
const { imie, nazwisko, kwota } = req.query;
```

Sprawdźmy działanie kodu jeszcze raz i spróbujmy zamienić imię `Jan` na `X&Y`. Nasz adres URL przyjmie postać *http://localhost:5555/?imie=X&Y&nazwisko=Kowalski&kwota=12000*. Efekt zamiany jest pokazany na rysunku 3.35 — użycie znaku `&` spowodowało **błąd w odczytaniu parametrów**.

Rysunek 3.35. Błąd w odczytaniu parametrów

Aby poprawnie wygenerować parametry URL, należy posłużyć się funkcją **`encodeURIComponent()`**, która zamieni znaki specjalne na odpowiadające im kody ASCII. W przypadku użytego odnośnika po konwersji przyjmie on postać *http://localhost:5555/?imie=X%26Z&nazwisko=Kowalski&kwota=12000*.

Przykładowy kod realizujący zadanie konwersji odnośnika mógłby wyglądać tak:

```
const url = `http://localhost:5555/?imie=${encodeURIComponent(imie)}&nazwisko=$
{encodeURIComponent(nazwisko)}&kwota=${encodeURIComponent(kwota)}`;
```

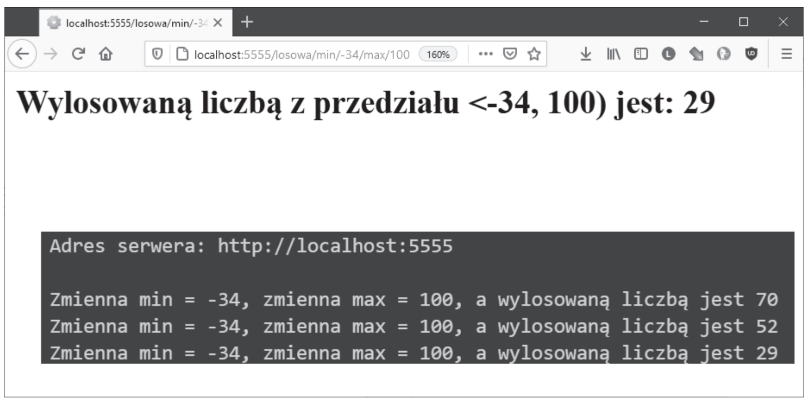
Drugim sposobem przesłania informacji jest użycie **parametrów ścieżek**.

Kod wykorzystujący ten sposób został przedstawiony na listingu 3.24. W użytym adresie, `/losowa/min/`**`:min`**`/max/`**`:max`**, parametry `:min` i `:max` to zmienne, pod które w odnośniku mogą zostać podstawione wartości. Zadaniem kodu jest wylosowanie liczby z zakresu (również liczb ujemnych) i wyświetlenie jej w oknie przeglądarki i w konsoli.

Listing 3.24. Parametry ścieżek

```
const express = require('express');

const app = express();
const port = 5555;

app.get('/losowa/min/:min/max/:max', (req, res) => {
    const min = Number(req.params.min);
    const max = Number(req.params.max);
    if (isNaN(min) || isNaN(max)) {
        res.send(`Nie wpisano liczb`);
    } else {
        const losowa = Math.floor((Math.random() * (max - min)) + min);
        res.send(`<h2>Wylosowaną liczbą z przedziału <${min}, ${max}) jest:
${losowa}</h2>`);
        console.log(`Zmienna min = ${min}, zmienna max = ${max}, a wylosowaną
liczbą jest ${losowa}`);
    }
});

app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port} \n`);
});
```

Aby kod mógł się wywołać, należy w pasku adresu przeglądarki wpisać np. *http://localhost:5555/losowa/min/-34/max/100*. Użycie tak skonstruowanego odnośnika spowoduje podstawienie pod stałą `min` wartości `-34`, a pod stałą `max` wartości `100` i w konsekwencji wylosowanie liczby z przedziału, przy czym zwrócona wartość jest większa lub równa `min` i jest mniejsza niż `max` (rysunek 3.36).



Rysunek 3.36. Użycie parametrów ścieżek

WSKAZÓWKA

W przykładzie do oddzielenia parametru od pozostałej części adresu użyto znaku ukośnika (`/`); dozwolony jest również znak kropki (`.`) oraz łącznika (`-`). Należy pamiętać, że użyty znak łącznika wpływa również na odnośnik — dla ścieżki `'/losowa/min.:min/max.:max'` obowiązuje odnośnik *http://localhost:5555/losowa/min.-34/max.100*, w przypadku zaś ścieżki `'/losowa/min-:min/max-:max'` odnośnik przyjmie postać *http://localhost:5555/losowa/min--34/max-100*. Konwencja nazewnicza parametrów jest zgodna z zasadami języka JavaScript: **wielkość liter ma znaczenie, nazwa parametru nie może zaczynać się od cyfry ani zawierać spacji, kropki, przecinka ani myślnika, a nazwą parametru nie może być słowo kluczowe zarezerwowane przez JavaScript**.

Mechanizm cookies

Express umożliwia również obsługę **mechanizmu cookies**. Użycie go pozwala przeglądarce zapisywać informację związaną z aktywnością osoby odwiedzającej daną witrynę. Cookies umożliwiają m.in. zapamiętanie preferowanych ustawień odwiedzanej strony (np. tematów wyświetlanych aktualności, układu strony, koloru), uwierzytelnianie (potwierdzanie tożsamości użytkownika) czy wyświetlanie indywidualnie dobranych reklam.

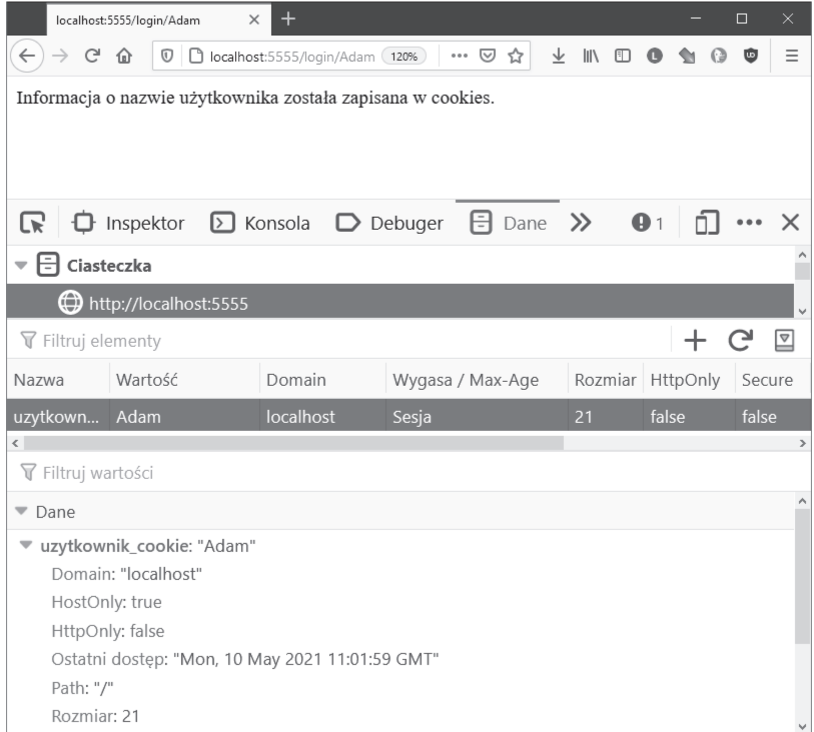
Zapisanie i usunięcie pliku cookie można przeprowadzić za pomocą kodu z listingu 3.25. Zostały w nim zdefiniowane trzy ścieżki. Pierwsza prowadzi do strony głównej, druga pod adres */login* (wymagane jest przekazanie parametru), trzecia zaś do strony o adresie */logout*.

Listing 3.25. Zapisanie i usunięcie pliku cookie

```
const express = require('express');

const app = express();
const port = 5555;

app.get('/', (req, res) => {
    res.send(`<h1>Poprawnie wylogowano</h1>
    Jesteś na stronie głównej`);
});

app.get('/login/:uzytkownik', (req, res) => {
    const uzytkownik = req.params.uzytkownik;
    res.cookie('uzytkownik_cookie', uzytkownik);
    res.send(`Informacja o nazwie użytkownika została zapisana w cookies.`);
});

app.get('/logout', (req, res) => {
    res.clearCookie('uzytkownik_cookie');
    res.redirect('/');
});
```

```
app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port} \n`);
});
```

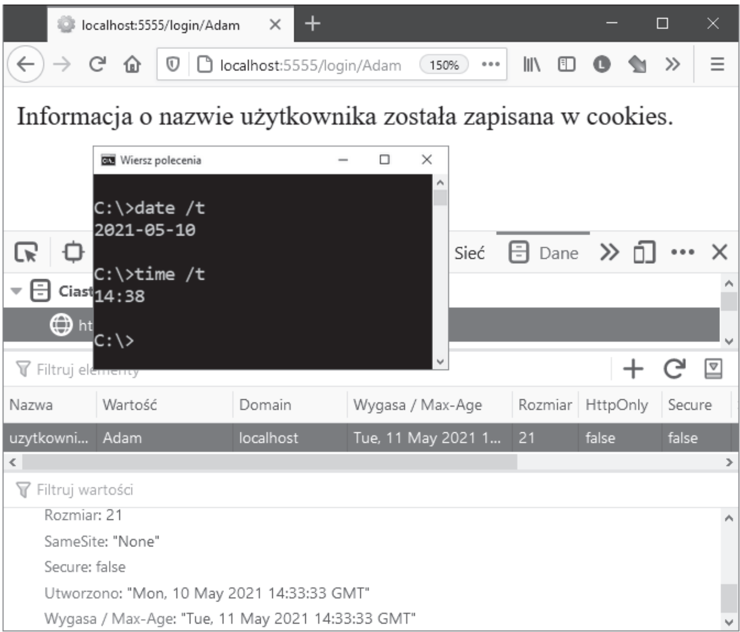
Użycie na przykład odnośnika *http://localhost:5555/login/Adam* spowoduje otwarcie strony i wyświetlenie informacji `Informacja o nazwie użytkownika została zapisana w cookies` (rysunek 3.37). Cookie zostało utworzone i zapisane (to, czy faktycznie zostało zapisane, można zweryfikować po otwarciu okna *Narzędzia dla twórców witryn* w przeglądarce Firefox lub o zbliżonej nazwie). Do zapisania cookie użyto metody **`.cookie()`** — jej parametrami są nazwa cookie (`'uzytkownik_cookie'`) oraz wartość (przekazany parametr `uzytkownik`).



Rysunek 3.37. Zapisanie cookies

Utworzony plik to tzw. **cookie sesyjne** — istnieje dopóty, dopóki otwarte jest okno przeglądarki. W momencie zamknięcia okna plik jest usuwany.

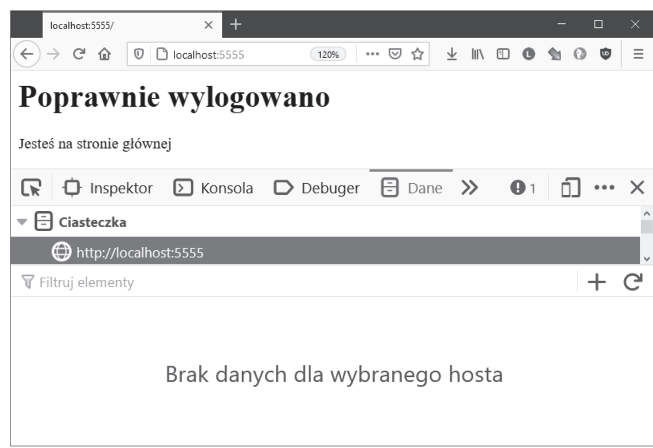
Dodanie do definicji utworzenia cookies opcjonalnego trzeciego parametru, `maxAge`, pozwoli zachować plik cookie przez dłuższy czas. Przekazywany argument to czas w milisekundach. Zmiana kodu na `res.cookie('uzytkownik_cookie', uzytkownik, { maxAge: 24 * 60 * 60 * 1000, });` spowoduje zapisanie pliku cookie, którego ważność została ustawiona na 24 godziny (rysunek 3.38).



Rysunek 3.38. Czas ważności pliku cookie

> **WSKAZÓWKA**
>
> Czas wygaśnięcia pliku cookie można również ustawić za pomocą parametru `expires` — ustawia on czas, do kiedy cookie jest ważne.

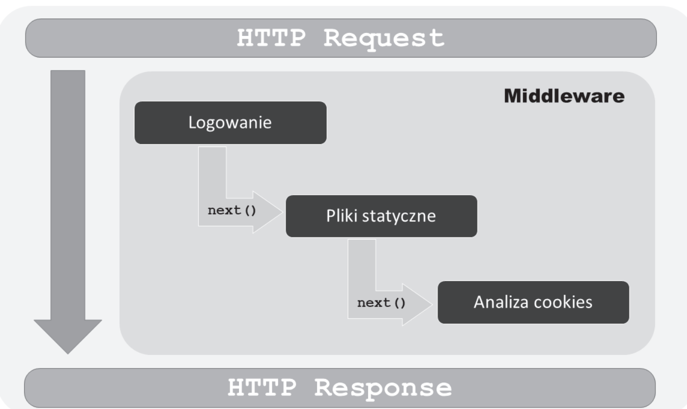
Przejście pod ścieżkę */logout* spowoduje usunięcie pliku cookie. Za skasowanie pliku odpowiada metoda `.clearCookie()`. Argumentem tej metody jest nazwa pliku. Po usunięciu cookie za pomocą metody `.redirect()` następuje przekierowanie do strony głównej (rysunek 3.39).



Rysunek 3.39. Usunięcie cookies

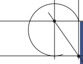
3.5.3. Warstwa pośrednia (middleware)

Wysłanie informacji do serwera rozpoczyna **sekwencję żądanie-odpowiedź** (request-response). Pomiędzy żądaniem a odpowiedzią znajduje się **warstwa pośrednia**, tzw. **middleware**. Ma ona dostęp do obiektów żądania i odpowiedzi i może te obiekty modyfikować, a także wywołać następne middleware (rysunek 3.40).



Rysunek 3.40. Middleware

Aby zarejestrować middleware, należy posłużyć się metodą `.use()` (to tylko jedno z zadań realizowanych przez tę metodę). Do rejestracji wykorzystuje się następujący kod: `<nazwa_zmiennej_reprezentująca_aplikację>.use(nazwa_middleware())`.



UWAGA

Middleware powinno się rejestrować przed zdefiniowaniem ścieżek.

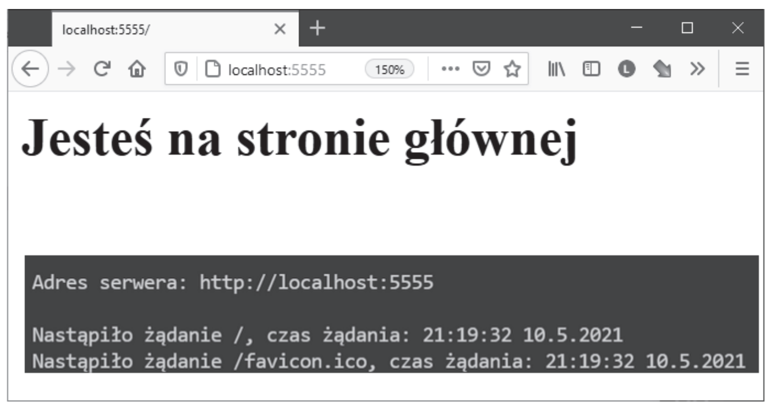
Kod pokazany na listingu 3.26 demonstruje działanie middleware. Zdefiniowane middleware rejestruje wystąpienie żądania — ścieżkę i czas. Użyta instrukcja `next()` nakazuje wywołanie następnego middleware.

Listing 3.26. Zasada działania middleware

```
const express = require('express');
const app = express();
const port = 5555;

const now = new Date();
const data = `${now.getDate()}.${now.getMonth() + 1}.${now.getFullYear()}`;
const czas = `${now.getHours()}:${now.getMinutes()}:${now.getSeconds()}`;
```

```
const middleware = (req, res, next) => {
    console.log(`Nastąpiło żądanie ${req.originalUrl}, czas żądania: ${czas}
${data}`);
    next();
};

app.use(middleware);

app.get('/', (req, res) => {
    res.send(`<h1>Jesteś na stronie głównej</h1>`);
});

app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port} \n`);
});
```

Każde odświeżenie strony to nowe żądanie (rysunek 3.41), przy czym jeśli połączymy je z wciśnięciem klawisza *Ctrl*, nastąpi ponowne przesłanie pliku arkusza stylów CSS i ikony favicon. Dodanie informacji przez middleware nastąpiło pomiędzy otrzymaniem żądania (request) a wysłaniem odpowiedzi (response).



Rysunek 3.41. Logowanie zdarzeń

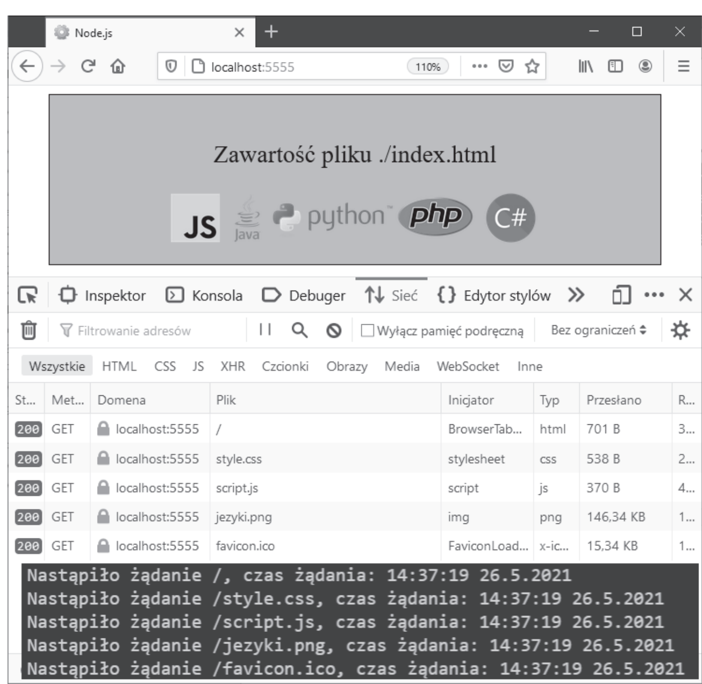
Jednym z najczęściej wykorzystywanych middleware jest **`express.static()`**. Funkcja ta pozwala na wysłanie plików znajdujących się we wspólnym katalogu (tzw. plików statycznych) w momencie wystąpienia żądania. **Oznacza to, że nie musimy definiować oddzielnych ścieżek dla każdego z żądań**.

To zadanie jest realizowane przez kod z listingu 3.27 (dodanie nowego middleware nastąpiło na podstawie kodu przedstawionego na listingu 3.25; różnice zaznaczono odrębnym formatowaniem). Stała www przechowuje ścieżkę do katalogu, w którym umieszczono pliki statyczne (użyta nazwa katalogu to *www*). Za pomocą kodu `app.use(express.static(www));` rejestrujemy nowe middleware, przy czym jako argument funkcji `express.static()` posłużyła nam stała www.

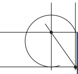
Listing 3.27. Pliki statyczne

```
const express = require('express');
const app = express();
const port = 5555;

const now = new Date();
const data = `${now.getDate()}.${now.getMonth() + 1}.${now.getFullYear()}`;
const czas = `${now.getHours()}:${now.getMinutes()}:${now.getSeconds()}`;

const www = __dirname + '\\www';

const middleware = (req, res, next) => {
    console.log(`Nastąpiło żądanie ${req.originalUrl}, czas żądania: ${czas}
${data}`);
    next();
};

app.use(middleware);
app.use(express.static(www));

app.listen(port, () => {
    console.log(`Adres serwera: http://localhost:${port} \n`);
});
```

Pliki (*index.html*, *style.css*, *script.js* oraz *favicon.ico*) użyte do omówienia działania serwera WWW opartego na Node.js i Expressie zostają wykorzystane ponownie (listingi 3.19 i 3.22). Znajdują się one w katalogu *www*. Dodatkowo w pliku *index.html* zostaje dodane nowe żądanie wczytania zdjęcia: `<img src="jezyki.png">`. Otwarcie strony głównej powoduje automatyczne wczytanie **wszystkich plików** (rysunek 3.42).



Rysunek 3.42. Użycie middleware express.static()



WSKAZÓWKA

www to jedna z chętniej stosowanych nazw katalogu przechowującego pliki statyczne. Bardzo często katalog ten jest również nazywany *public* bądź *static*.

Zadanie 3.9.

Zmodyfikuj kod serwera HTTP opartego na Expressie tak, aby pracował z innym zestawem plików.

Zadanie 3.10.

Stwórz stronę z dwoma polami formularza typu tekstowego. Prześlij do serwera tekst, który zostanie w nie wpisany.

Pytania kontrolne

1. Wskaż różnice pomiędzy serwerem HTTP opartym na Node.js a serwerem opartym na frameworku Express.
2. Jakie jest przeznaczenie metod `.send()` i `.sendFile()`?
3. W jaki sposób można przesłać informacje z frontendu do backendu?
4. Wskaż obszary zastosowania mechanizmu cookies.
5. Jak utworzyć i skasować plik cookie?
6. W jaki sposób działa middleware?

3.6. Baza danych MongoDB

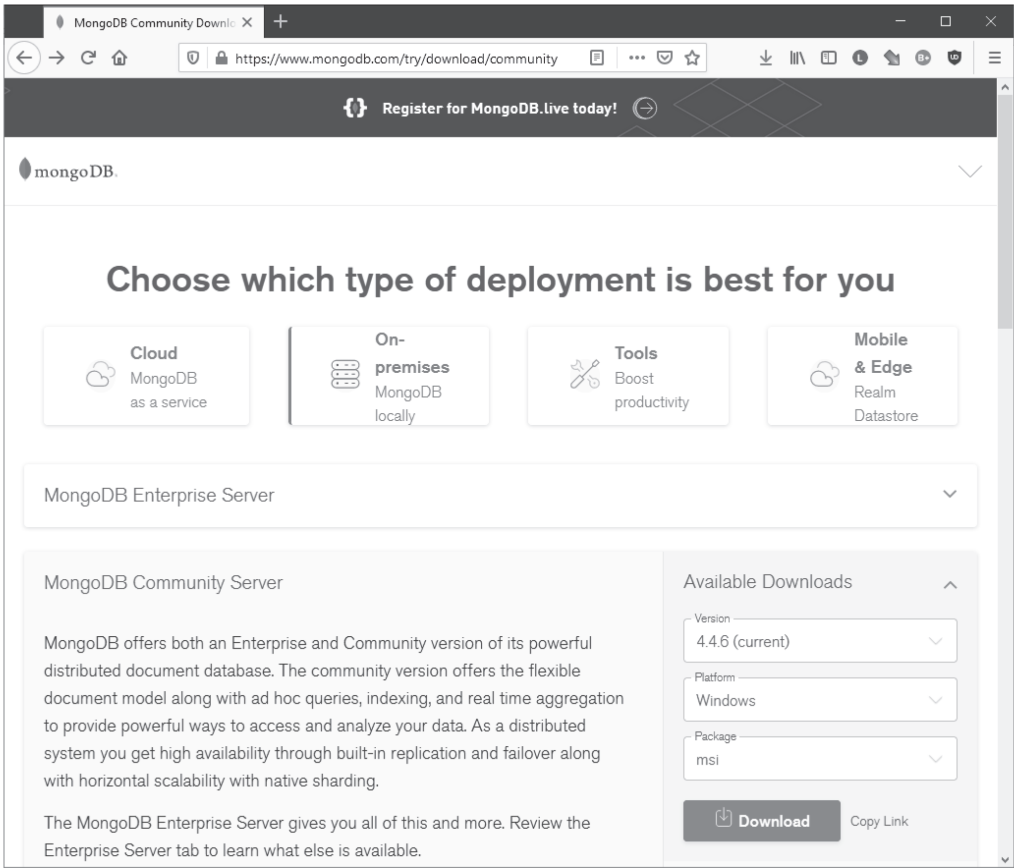
Bazy danych pozwalają przechowywać informacje. To dzięki nim możemy na stałe zapisać informacje wprowadzone przez użytkownika.

Bazy danych można podzielić na:

- **Relacyjne bazy danych** (MySQL, PostgreSQL, Oracle Database) — informacje w tego typu bazie są przechowywane w tabelach powiązanych ze sobą, a dostęp do nich uzyskuje się po zbudowaniu i wysłaniu zapytania SQL (kwerendy).
- **Nierelacyjne bazy danych** (MongoDB, CouchDB) — bazy danych zawierające kolekcje, w których przechowywane są dokumenty. W przeciwieństwie do baz relacyjnych kolekcje nie muszą mieć ściśle ustalonej struktury. Bazy danych tego typu często są określane jako NoSQL.

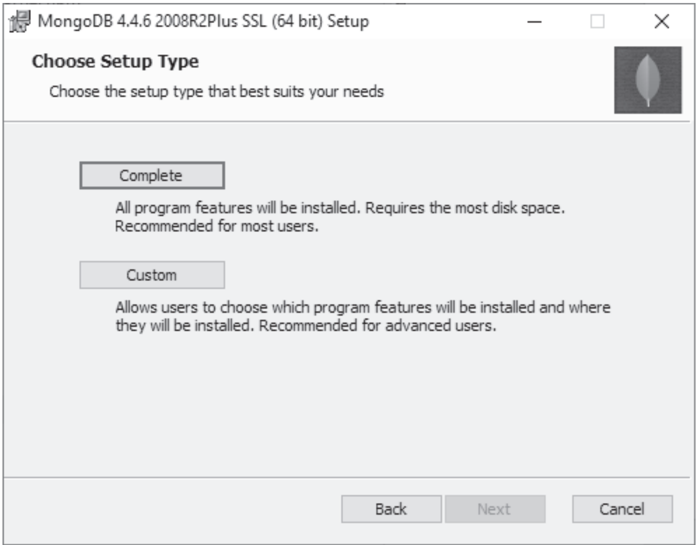
3.6.1. Instalacja i uruchomienie serwera MongoDB

Instalator serwera bazy danych **MongoDB** jest dostępny pod adresem *https://www.mongodb.com/try/download/community* (rysunek 3.43).



Rysunek 3.43. Strona pobierania MongoDB

Po pobraniu instalatora rozpoczynamy procedurę instalacji. Instalator MongoDB zapyta się o jej typ (rysunek 3.44). Po kliknięciu przycisku *Custom* będziemy mogli określić, jakie komponenty mają być obecne w systemie bazodanowym. Kliknięcie *Complete* spowoduje natomiast zainstalowanie wszystkich elementów serwera MongoDB.



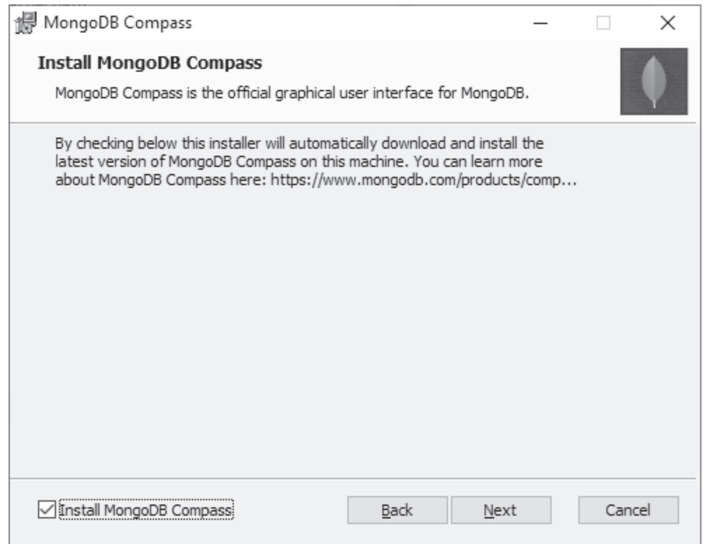
Rysunek 3.44. Wybór typu instalacji MongoDB

Następne okno pozwala określić, gdzie ma zostać uruchomiona usługa serwera MongoDB. W przypadku środowiska domenowego należy zmienić domyślnie zaznaczoną opcję na *Run service as a local or domain user* i podać dane użytkownika, który ma prawo do przeprowadzenia instalacji. Jeśli pracujemy w grupie roboczej, pozostawiamy zaznaczoną opcję *Run service as Network Service user*. Nazwę usługi można określić w polu *Service Name*. Dwa ostatnie pola definiują ścieżkę do katalogu, w którym będą zapisywane dane, oraz ścieżkę do katalogu przechowującego logi, które w razie wystąpienia problemów z działaniem serwera pozwolą ustalić przyczynę (rysunek 3.45).

MongoDB 4.4.6 2008R2Plus SSL (64 bit) Service Customization
Service Configuration
Specify optional settings to configure MongoDB as a service.
Install MongoD as a Service
Run service as Network Service user
Run service as a local or domain user:
Account Domain: .
Account Name: MongoDB
Account Password:
Service Name: MongoDB
Data Directory: C:\Program Files\MongoDB\Server\4.4\data\
Log Directory: C:\Program Files\MongoDB\Server\4.4\log\
< Back
Next >
Cancel

Rysunek 3.45. Opcje instalacji MongoDB

Ostatni krok pozwala dodać do instalacji narzędzie MongoDB Compass Community, służące do zarządzania danymi (rysunek 3.46). Po kliknięciu przycisku *Next*, a następnie *Install* serwer MongoDB zostanie zainstalowany.



Rysunek 3.46. Oprogramowanie dodatkowe MongoDB Compass

WSKAZÓWKA

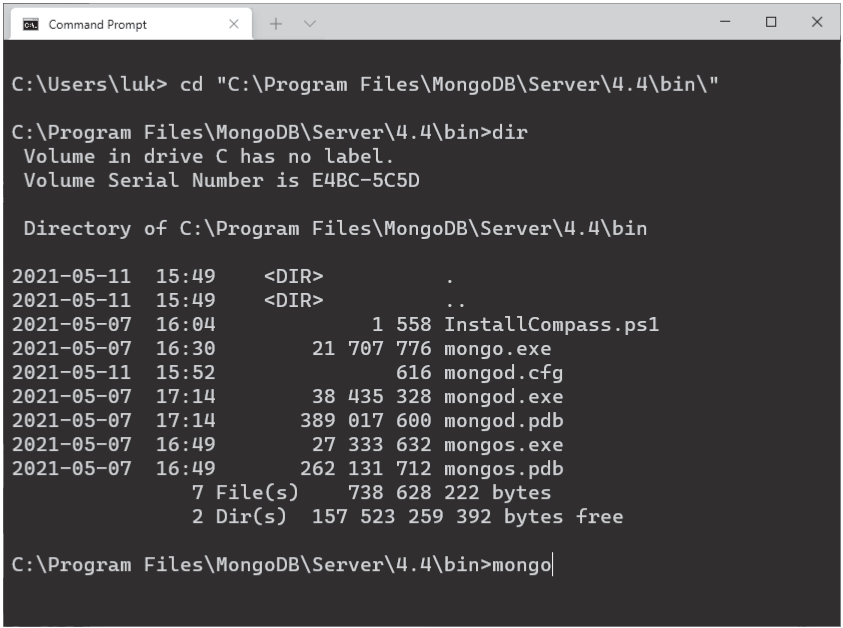
Przed zainstalowaniem MongoDB powinno się zamknąć wszystkie okna przeglądarek. Po zakończeniu instalacji będzie wymagane ponowne uruchomienia komputera.

Po zainstalowaniu MongoDB będzie się można komunikować z serwerem za pomocą konsoli **PowerShell** bądź tradycyjnego *wiersza poleceń*.

CIEKAWOSTKA

Od niedawna w systemie Windows dostępne jest narzędzie **Windows Terminal**, które w jednym oknie integruje obie powłoki, pozwala pracować na odrębnych kartach oraz umożliwia bezpośrednie kopiowanie i wklejanie poleceń. Narzędzie można pobrać bezpłatnie za pośrednictwem sklepu Microsoft Store.

Aby nawiązać połączenie z serwerem MongoDB, otwieramy dowolną konsolę i za pomocą polecenia `cd "C:\Program Files\MongoDB\Server\<wersja_serwera>\bin\"` przechodzimy do katalogu zawierającego pliki serwera (rysunek 3.47).



Rysunek 3.47. Uruchomienie powłoki serwera MongoDB

Wydanie polecenia `mongo` (`./mongo` w konsoli PowerShell) uruchamia **powłokę serwera** (ang. *shell*) MongoDB. Na wstępie jesteśmy informowani o wersji serwera i zostaje wyświetlone ostrzeżenie — dostęp do serwera w żaden sposób nie jest ograniczony, każdy może go uzyskać (rysunek 3.48).

```
Command Prompt - mongo

C:\Program Files\MongoDB\Server\4.4\bin>mongo
MongoDB shell version v4.4.6
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiSer
viceName=mongodb
Implicit session: session { "id" : UUID("48705389-825d-44d1-b62a-c004e01
4e62f") }
MongoDB server version: 4.4.6
---
The server generated these startup warnings when booting:
        2021-05-11T15:52:52.886+02:00: Access control is not enabled for
 the database. Read and write access to data and configuration is unrest
ricted
---
>
```

Rysunek 3.48. Powłoka serwera MongoDB

3.6.2. Podstawowe operacje — CRUD (create, read, update, delete)

Polecenia serwera MongoDB wykorzystują składnię języka JavaScript. Wydanie polecenia (rozpoczynamy od `db`) `db.pracownicy.insertOne({id_prac: 10, imie: "Jan", nazwisko: "Kowalski", etat: "prezes", zatrudniony: new Date("2001-01-01"), placa: 5730.00, id_zesp: 10})` będzie miało dwojaki skutek:

- zostanie utworzona **kolekcja** *pracownicy* (kolekcja to odpowiednik tabeli z relacyjnych baz danych);
- w kolekcji *pracownicy* zostanie utworzony nowy **dokument** (odpowiednik rekordu), który gromadzi dane.

Zatwierdzenie polecenia spowoduje zapisanie informacji w bazie danych (informuje o tym pole `acknowledged` ustawione na wartość `true`). Zwrócony numer jest identyfikatorem dodanego obiektu — posłuży on do jego identyfikacji.

Polecenie można napisać w jednym ciągu, ale też rozłożyć na wiele linijek (za pomocą klawisza *Enter*). Zamknięcie wszystkich nawiasów i umieszczenie średnika (;) na końcu kończy polecenie (rysunek 3.49).

```
Command Prompt - mongo
> db.pracownicy.insertOne({id_prac: 10, imie: "Jan", nazwisk
o: "Kowalski", etat: "prezes", zatrudniony: new Date("2001-0
1-01"), placa: 5730.00, id_zesp: 10})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("60c1caf1441d2748dae2bfac")
}
> db.pracownicy.insertOne({
... id_prac: 20,
... imie: "Tadeusz",
... nazwisko: "Nowak",
... etat: "Dyrektor",
... zatrudniony: new Date("2004-06-14"),
... placa: 4000.00,
... id_zesp: 20
... });
{
        "acknowledged" : true,
        "insertedId" : ObjectId("60c1df65441d2748dae2bfae")
}
>
```

Rysunek 3.49. Zapisanie informacji — utworzenie kolekcji i dokumentu

Można również jednym poleceniem utworzyć kilka dokumentów. Wydanie polecenia:

```
db.pracownicy.insertMany([{id_prac: 30, imie: "Beata", nazwisko: "Tryla", etat:
'pracownik', zatrudniony: new Date("2010-03-02"), placa: 2300.00, id_zesp:
30}, {id_prac: 40, imie: "Katarzyna", nazwisko: "Czarny", etat: 'pracownik',
zatrudniony: new Date("2013-10-22"), placa: 2500.00, id_zesp: 30}])
```

spowoduje utworzenie dwóch dokumentów. Informacje dotyczące pojedynczego dokumentu są zawarte pomiędzy **nawiasami klamrowymi**, a wszystkie dodawane dokumenty są określone pomiędzy **nawiasami kwadratowymi** (tak jak w tablicy) i rozdzielone znakiem przecinka (,). Metodę `.insertOne()` należy zamienić na `.insertMany()`.

Podobnie jak wtedy, gdy tworzony jest jeden dokument, po zatwierdzeniu polecenia jest wyświetlana informacja zwrotna — potwierdzenie dodania wraz z identyfikatorami (rysunek 3.50).

```
Command Prompt - mongo
> db.pracownicy.insertMany([{id_prac: 30, imie: "Beata", nazw
isko: "Tryla", etat: 'pracownik', zatrudniony: new Date("2010
-03-02"), placa: 2300.00, id_zesp: 30}, {id_prac: 40, imie: "
Katarzyna", nazwisko: "Czarny", etat: 'pracownik', zatrudnion
y: new Date("2013-10-22"), placa: 2500.00, id_zesp: 30}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("60c1dfde441d2748dae2bfb3"),
                ObjectId("60c1dfde441d2748dae2bfb4")
        ]
}
> 
```

Rysunek 3.50. Dodanie wielu dokumentów

Aby zweryfikować, czy informacja została zapisana, należy posłużyć się poleceniem `db.pracownicy.find()`, które wyświetli wszystkie dokumenty znajdujące się w kolekcji `pracownicy` (rysunek 3.51).

```
Command Prompt - mongo
> db.pracownicy.find()
{ "_id" : ObjectId("609aa63aa51dc4830351daa3"), "id_prac" : 10, "im
ie" : "Jan", "nazwisko" : "Kowalski", "etat" : "prezes", "zatrudnio
ny" : ISODate("2001-01-01T00:00:00Z"), "placa" : 5730, "id_zesp" :
10 }
{ "_id" : ObjectId("609aa63aa51dc4830351daa4"), "id_prac" : 20, "im
ie" : "Tadeusz", "nazwisko" : "Nowak", "etat" : "dyrektor", "zatrud
niony" : ISODate("2004-06-14T00:00:00Z"), "placa" : 4000, "id_zesp"
 : 20 }
{ "_id" : ObjectId("609aa63aa51dc4830351daa5"), "id_prac" : 30, "im
ie" : "Beata", "nazwisko" : "Tryla", "etat" : "pracownik", "zatrudn
iony" : ISODate("2010-03-02T00:00:00Z"), "placa" : 2300, "id_zesp"
: 30 }
{ "_id" : ObjectId("609aa63aa51dc4830351daa6"), "id_prac" : 40, "im
ie" : "Katarzyna", "nazwisko" : "Czarny", "etat" : "pracownik", "za
trudniony" : ISODate("2013-10-22T00:00:00Z"), "placa" : 2500, "id_z
esp" : 30 }
> 
```

Rysunek 3.51. Efekt wydania polecenia db.pracownicy.find()

Taka forma prezentacji danych może się okazać mało czytelna, dlatego warto na końcu polecenia dodać `.pretty()` (rysunek 3.52).

```
> db.pracownicy.find().pretty()
{
        "_id" : ObjectId("609aa63aa51dc4830351daa3"),
        "id_prac" : 10,
        "imie" : "Jan",
        "nazwisko" : "Kowalski",
        "etat" : "prezes",
        "zatrudniony" : ISODate("2001-01-01T00:00:00Z"),
        "placa" : 5730,
        "id_zesp" : 10
}
{
        "_id" : ObjectId("609aa63aa51dc4830351daa4"),
        "id_prac" : 20,
        "imie" : "Tadeusz",
        "nazwisko" : "Nowak",
        "etat" : "dyrektor",
        "zatrudniony" : ISODate("2004-06-14T00:00:00Z"),
        "placa" : 4000,
        "id_zesp" : 20
}
```

Rysunek 3.52. Użycie metody .pretty()

Do wyszukania informacji w dokumencie służy metoda **`.find()`**. W nawiasach okrągłych wpisujemy to, czego szukamy. Wydanie na przykład polecenia `db.pracownicy.find({id_zesp: 30})` spowoduje wyświetlenie wszystkich pracowników należących do zespołu, którego ID równa się `30` (*Beata Tryla* i *Katarzyna Czarny*).

Użycie przecinka pozwala dodać kolejny warunek, np. polecenie `db.pracownicy.find({etat: "pracownik", placa: 2300})` wyświetli pracownika, którego płaca wynosi `2300`.

Użycie **kryterium `$in`** (w MongoDB kryterium poprzedzamy znakiem dolara) pozwoli wyświetlić osoby, które spełniają dane kryteria. Wydanie na przykład polecenia `db.pracownicy.find({etat: {$in: ["prezes", "dyrektor"]}})` wyświetli osoby, których etat to `prezes` bądź `dyrektor` (*Jan Kowalski* i *Tadeusz Nowak*). Zastąpienie `$in` kryterium `$nin` wyświetli wszystkie osoby, których etat **nie jest** ustawiony na `prezes` lub `dyrektor` (*Beata Tryla* i *Katarzyna Czarny*).

Pozostałe kryteria to:

- **`$eq`** — równe,
- **`$ne`** — nierówne,
- **`$gt`** — większe niż,
- **`$lt`** — mniejsze niż,
- **`$gte`** — większe lub równe,
- **`$lte`** — mniejsze lub równe.

Oto trzy przykłady zapytań z użyciem zaprezentowanych kryteriów:

- `db.pracownicy.find({placa: {$gt: 3000}})` — wyświetli osoby, których płaca jest wyższa niż 3000 zł (*Tadeusz Nowak* i *Jan Kowalski*);
- `db.pracownicy.find({zatrudniony: {$lt: ISODate("2011-01-01")}})` — wyświetli osoby, które zostały zatrudnione przed 1 stycznia 2011 r. (*Jan Kowalski*, *Tadeusz Nowak* i *Beata Tryla*);
- `db.pracownicy.find({placa: {$gt: 3000, $lt: 5000}})` — wyświetli osoby, których płaca mieści się w przedziale od 3000 do 5000 zł (*Tadeusz Nowak*).

MongoDB pozwala również tworzyć warunki z wykorzystaniem spójnika **`$or`**.

Polecenie `db.pracownicy.find({$or: [{etat: "pracownik"}, {placa: {$gt: 4000}}]})` sprawi, że zostaną wyświetleni wszyscy pracownicy niezależnie od zarobków, ale także wszystkie pozostałe osoby, które zarabiają więcej niż 4000 zł (*Beata Tryla*, *Katarzyna Czarny* i *Jan Kowalski*).

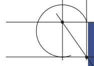
Tak jak bazy relacyjne, MongoDB pozwala używać **funkcji agregujących**. Na przykład wyliczenie średniej płacy sprowadza się do wydania polecenia `db.pracownicy.aggregate([{$group: {"_id": "_id", sredniaPlac: { $avg: "$placa" }}}])`. Zmiana `$avg` na `$sum` spowoduje dodanie do siebie płac, a `$min` lub `$max` — wyświetlenie, odpowiednio, najniższej i najwyższej płacy.

Aby zaktualizować dokumenty, należy posłużyć się metodą **`.update()`**. Uaktualnienie będzie wymagało przekazania od dwóch do trzech argumentów. Pierwszy dotyczy aktualizowanego dokumentu — wskazujemy, który dokument lub które dokumenty mają zostać uaktualnione (dokumenty wskazuje się w taki sam sposób, w jaki wyszukuje się je z użyciem `.find()`). Drugi argument określa informacje, jakie w wybranym dokumencie zostaną zmodyfikowane. Trzeci argument to pole opcji.

Spróbujmy zatem zmienić etat pracownika *Beata Tryla* na `kierownik`. Aktualizację przeprowadzimy przez wydanie polecenia `db.pracownicy.update({nazwisko: 'Tryla'}, {$set: {etat: 'kierownik'}})`. W pierwszej kolejności został odszukany dokument, który będzie modyfikowany, `{nazwisko: 'Tryla'}`, a następnie w polu `etat` ustawiono nową wartość, `kierownik {$set: {etat: 'kierownik'}}`. Po wydaniu polecenia jest wyświetlane podsumowanie — zapytanie pasuje do jednego dokumentu i jeden dokument został zmieniony (rysunek 3.53).

```
> db.pracownicy.update({nazwisko:'Tryla'},
... {$set:{etat:'kierownik'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModifie
d" : 1 })
> db.pracownicy.find({nazwisko:'Tryla'}).pretty()
{
        "_id" : ObjectId("609aa63aa51dc4830351daa5"),
        "id_prac" : 30,
        "imie" : "Beata",
        "nazwisko" : "Tryla",
        "etat" : "kierownik",
        "zatrudniony" : ISODate("2010-03-02T00:00:00Z"),
        "placa" : 2300,
        "id_zesp" : 30
}
>
```

Rysunek 3.53. Aktualizacja — użycie metody .update()



UWAGA

Pominięcie `$set` usunie wszystkie pola dokumentu, zastępując je polem określonym w poleceniu. Efekt pominięcia `$set` i wydania polecenia `db.pracownicy.update({nazwisko: 'Tryla'}, {etat: 'kierownik'})` jest pokazany na rysunku 3.54. Niemożliwe jest ponowne odszukanie pracownika z użyciem nazwiska, gdyż takie pole w dokumencie nie istnieje — należy użyć identyfikatora.

```
db.pracownicy.update({nazwisko: 'Tryla'}, {etat:'kierownik'})})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1
})
> db.pracownicy.find({"_id" : ObjectId("609aa63aa51dc4830
351daa5")}).pretty()
{ "_id" : ObjectId("609aa63aa51dc4830351daa5"), "etat" :
"kierownik" }
>
```

Rysunek 3.54. Efekt pominięcia w zapytaniu parametru $set

Na rysunku 3.55 przedstawiono próbę zmiany płacy wszystkim pracownikom na kwotę `3000`; niestety wydane polecenie `db.pracownicy.update({etat: 'pracownik'}, {$set: {placa:3000}})` żądaną kwotę ustawiło tylko w jednym dokumencie. Stało się tak, ponieważ wydane zapytanie **jest stosowane do pierwszego dopasowanego dokumentu**.

```
Command Prompt - mongo
> db.pracownicy.update({etat: 'pracownik'}, {$set: {placa:3000}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.pracownicy.find({etat: 'pracownik'}).pretty()
{
        "_id" : ObjectId("609aa63aa51dc4830351daa5"),
        "id_prac" : 30,
        "imie" : "Beata",
        "nazwisko" : "Tryla",
        "etat" : "pracownik",
        "zatrudniony" : ISODate("2010-03-02T00:00:00Z"),
        "placa" : 3000,
        "id_zesp" : 30
}
{
        "_id" : ObjectId("609aa63aa51dc4830351daa6"),
        "id_prac" : 40,
        "imie" : "Katarzyna",
        "nazwisko" : "Czarny",
        "etat" : "pracownik",
        "zatrudniony" : ISODate("2013-10-22T00:00:00Z"),
        "placa" : 2500,
        "id_zesp" : 30
}
```

Rysunek 3.55. Próba zmiany informacji w wielu dokumentach

Aby zmodyfikować wiele elementów, należy dodać trzeci argument, w którym zostanie przekazana opcja `multi: true`. Po dołączeniu opcji ponowne wydanie polecenia spowoduje wprowadzenie zmian we wszystkich pasujących dokumentach — `placa` wszystkich pracowników zostanie ustalona na `3000` (rysunek 3.56).

WSKAZÓWKA

Argument **`multi: true`** można pominąć, jeśli do aktualizacji jest używana metoda **`.updateMany()`**. Polecenie pokazane na rysunku 3.56 jest równoznaczne z `db.pracownicy.updateMany({etat: 'pracownik'}, {$set: {placa:3000}})`.

```
> db.pracownicy.update({etat: 'pracownik'}, {$set: {placa:3000}}, {
multi:true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
> db.pracownicy.find({etat: 'pracownik'}).pretty()
{
        "_id" : ObjectId("609aa63aa51dc4830351daa5"),
        "id_prac" : 30,
        "imie" : "Beata",
        "nazwisko" : "Tryla",
        "etat" : "pracownik",
        "zatrudniony" : ISODate("2010-03-02T00:00:00Z"),
        "placa" : 3000,
        "id_zesp" : 30
}
{
        "_id" : ObjectId("609aa63aa51dc4830351daa6"),
        "id_prac" : 40,
        "imie" : "Katarzyna",
        "nazwisko" : "Czarny",
        "etat" : "pracownik",
        "zatrudniony" : ISODate("2013-10-22T00:00:00Z"),
        "placa" : 3000,
        "id_zesp" : 30
}
>
```

Rysunek 3.56. Zmiana informacji w wielu dokumentach

Przy usuwaniu dokumentów stosujemy te same kryteria co przy ich wyszukiwaniu. Oczywiście metodę `.find()` zastępujemy metodą **`.deleteOne()`**, gdy będzie usuwany jeden obiekt, lub **`.deleteMany()`**, gdy dokumentów jest wiele.

Użycie polecenia `db.pracownicy.deleteOne({nazwisko: 'Nowak'})` spowoduje usunięcie dokumentu pracownika o nazwisku *Nowak*, a polecenie `db.pracownicy.deleteMany({etat: 'pracownik'})` usunie wszystkie dokumenty, w których pole `etat` ma wartość `pracownik` (rysunek 3.57).

```
> db.pracownicy.deleteOne({nazwisko: 'Nowak'})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.pracownicy.deleteMany({etat: 'pracownik'})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.pracownicy.find().pretty()
{
        "_id" : ObjectId("609ad77c8f2b61a5f3fea384"),
        "id_prac" : 10,
        "imie" : "Jan",
        "nazwisko" : "Kowalski",
        "etat" : "kierownik",
        "zatrudniony" : ISODate("2001-01-01T00:00:00Z"),
        "placa" : 3000,
        "id_zesp" : 10
}
>
```

Rysunek 3.57. Usuwanie dokumentów

Domyślnie kolekcja *pracownicy* została utworzona w bazie *test*. Aby to zmienić i utworzyć nową bazę, należy użyć polecenia use <nazwa_bazy>. Przedstawiony na rysunku 3.58 kod tworzy nową bazę danych *firma* wraz z kolekcją *pracownicy*.

```
Command Prompt - mongo
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
test    0.000GB
> use firma
switched to db firma
> db.pracownicy.insertOne({name: "Adam", wiek: 25})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("609bca7a8dc16b4e1d912747")
}
> show dbs
admin   0.000GB
config  0.000GB
firma   0.000GB
local   0.000GB
test    0.000GB
> |
```

Rysunek 3.58. Utworzenie nowej bazy danych

UWAGA

Aby utworzona baza danych była widoczna po użyciu polecenia `show dbs`, musi w niej być zapisany przynajmniej jeden dokument.

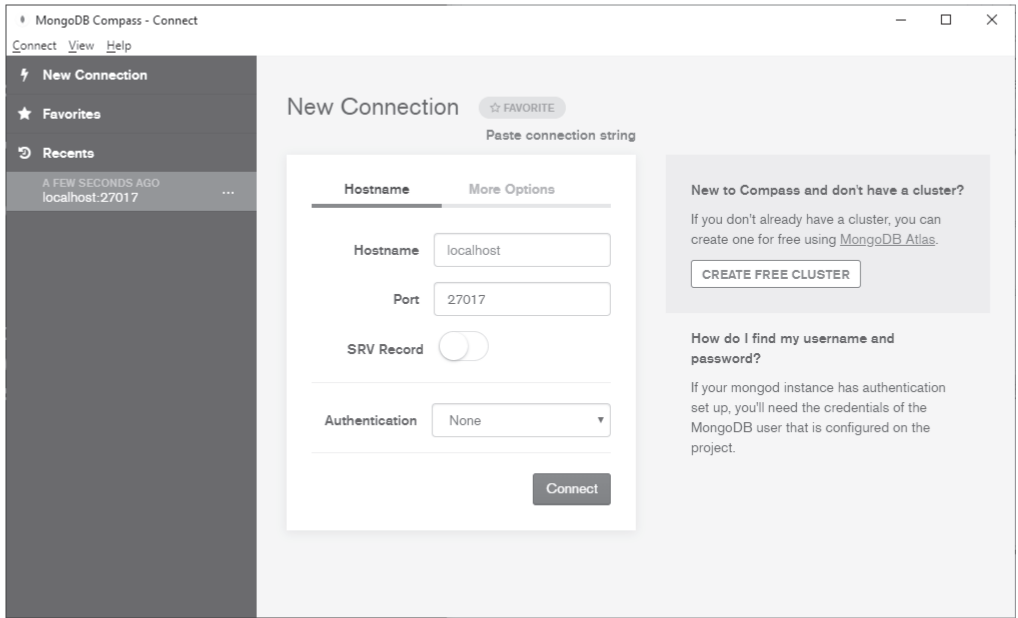
3.6.3. Narzędzie MongoDB Compass

MongoDB Compass to narzędzie, które pozwala zarządzać serwerem MongoDB i zgromadzonymi danymi (to niejako odpowiednik narzędzia phpMyAdmin, służącego do zarządzania bazą danych MySQL).

Na pierwszym ekranie po uruchomieniu aplikacji będziemy mogli zdecydować, jakie informacje o działaniu narzędzia będą zwracane jego twórcom oraz czy ma być włączona opcja automatycznej aktualizacji.

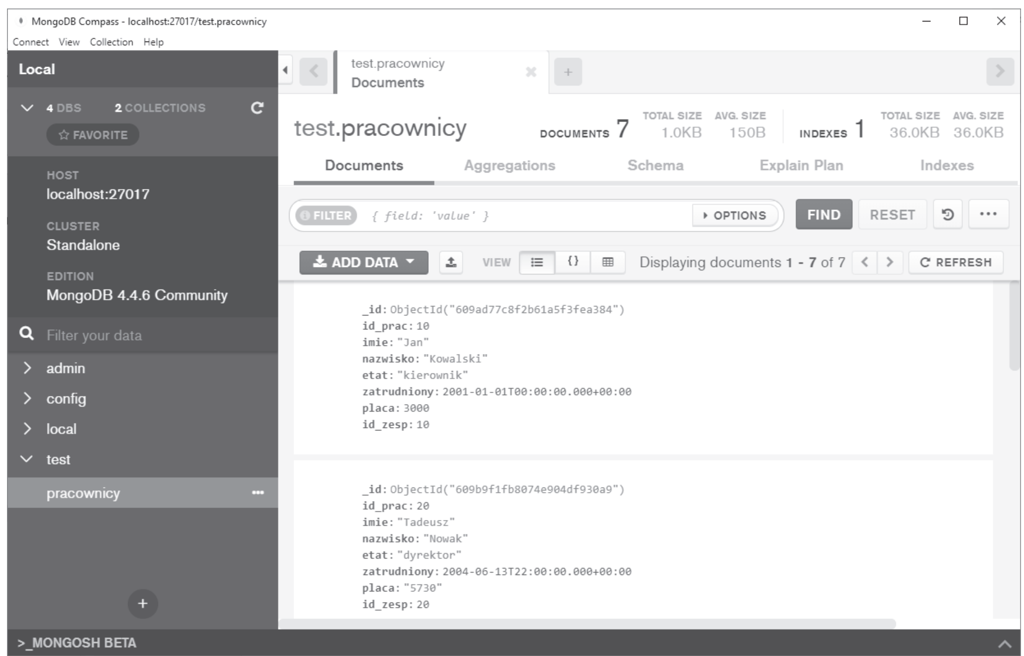
Na głównym ekranie należy wybrać odnośnik *Fill in connection fields individually*, co pozwoli określić opcje połączenia. Domyślnie wprowadzone ustawienia są wystarczające do nawiązania połączenia z serwerem MongoDB. Wybranie przycisku *Connect* inicjuje połączenie (rysunek 3.59).

Po nawiązaniu połączenia w lewej części okna zostaną wyświetlone wszystkie bazy danych. Wybranie odpowiedniej kolekcji wyświetli zawarte w niej dokumenty. W głównym oknie jest widoczny podgląd dokumentów. Po nakierowaniu wskaźnika myszy na dany dokument



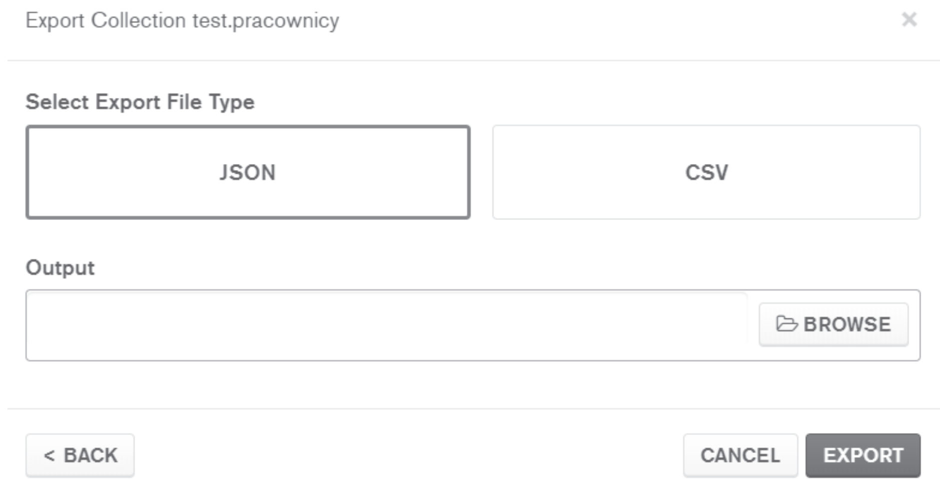
Rysunek 3.59. MongoDB Compass — połączenie z bazą

w prawym górnym rogu pojawią się cztery ikonki, których kliknięcie umożliwi edytowanie, skopiowanie, sklonowanie lub usunięcie dokumentu. Wybranie przycisku *ADD DATA* pozwoli utworzyć nowe dokumenty, a znajdujący się tuż obok przycisk *Export Collection* służy do eksportowania danych (rysunek 3.60).



Rysunek 3.60. Główne okno narzędzia MongoDB Compass

Dzięki MongoDB Compass wyeksportujemy zarówno wszystkie dokumenty, jak i wybrane. W obrębie dokumentu można także określić eksportowane pola. Po wskazaniu ścieżki dane można zapisać w formacie JSON, jak również CSV (rysunek 3.61).



Rysunek 3.61. Eksport dokumentów

Narzędzie to zapewnia znacznie więcej niż tylko podstawowe funkcje, takie jak graficzny przegląd danych czy wykonywanie na danych operacji CRUD (*create*, *read*, *update*, *delete*) — pozwala zarządzać indeksami, wykonywać zapytania *ad hoc*, zarządzać kopiami bezpieczeństwa i odzyskiwać dane, a także konfigurować zabezpieczenia.

3.6.4. MongoDB w Node.js

Aby móc wykorzystać bazę MongoDB w swoich projektach, należy rozpocząć od zainstalowania sterownika MongoDB dla Node.js. Sterownik instaluje się poleceniem `npm install mongodb --save`.

WSKAZÓWKA

Bazą MongoDB można zarządzać również z użyciem biblioteki mongoose.

Pierwszym krokiem do ustanowienia połączenia z bazą MongoDB jest zaimportowanie biblioteki. Importujemy ją dokładnie tak samo jak inne moduły — z użyciem funkcji `require()`. Kod `const mongo = require('mongodb');` dołącza sterownik do projektu.

Drugim krokiem jest utworzenie klienta, który będzie łączył aplikację z serwerem MongoDB. Kod tworzący klienta to: `const klient = new mongo.MongoClient('mongodb://localhost:27017', {useNewUrlParser: true, useUnifiedTopology: true});`.

Użycie metody `.connect()` ustanowi połączenie: `klient.connect();`.

Kod, który realizuje połączenie z serwerem MongoDB i wyświetla listę pracowników pobraną z kolekcji *pracownicy* zapisanej w bazie o nazwie *firma*, został przedstawiony na listingu 3.28.

W metodzie `.connect()` określono wywołanie zwrotne — w momencie wystąpienia błędu zostanie zwrócony komunikat `Błąd połączenia`. Jeśli połączenie dojdzie do skutku, w konsoli wyświetli się komunikat `Ustanowiono połączenie mongodb://localhost:27017`.

Stała `db` przechowuje informacje o wybranej bazie, a do stałej `pracownicy` zostaje przypisana kolekcja.

Wyszukiwanie informacji przebiega identycznie jak w konsoli serwera. Aby dane mogły być wyświetlone w konsoli, są one konwertowane za pomocą metody `.toArray()` na tabelę.

Listing 3.28. Połączenie z serwerem MongoDB i pobranie informacji

```
const mongo = require('mongodb');

// Konfiguracja klienta
const url = 'mongodb://localhost:27017';
const klient = new mongo.MongoClient(url, { useNewUrlParser: true,
useUnifiedTopology: true});

// Podłączenie do serwera
klient.connect(err => {
    if (!err) console.log(`Ustanowiono połączenie ${url}`);
    else console.log('Błąd połączenia', err);
});

// Wybranie bazy firma i kolekcji pracownicy
const db = klient.db('firma');
const pracownicy = db.collection('pracownicy');

// Pobranie i wyświetlenie listy pracowników
pracownicy.find({}).toArray((err, listaPracownikow) => {
    console.log(listaPracownikow);
});

// Zakończenie połączenia
klient.close();
```

Działanie kodu jest pokazane na rysunku 3.62.

```
const mongo = require('mongodb');

const url = 'mongodb://localhost:27017';
const klient = new mongo.MongoClient(url, { useNewUrlParser: true, useUnifi
klient.connect(err => {
    if (!err) console.log(`Ustanowiono połączenie ${url}`);
    else console.log('Błąd połączenia', err);
```

```
X:\mongo>node app.js
Ustanowiono połączenie mongodb://localhost:27017
[
  {
    _id: 609ad77c8f2b61a5f3fea384,
    id_prac: 10,
    imie: 'Jan',
    nazwisko: 'Kowalski',
    etat: 'kierownik',
    zatrudniony: 2001-01-01T00:00:00.000Z,
    placa: 3000,
```

Rysunek 3.62. Połączenie Node.js z MongoDB

WSKAZÓWKA

Listę aktywnych połączeń z serwerem można sprawdzić po wydaniu polecenia `db.serverStatus().connections`.

Zadanie 3.11.

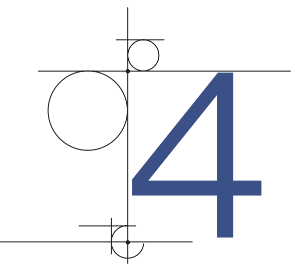
Zaprojektuj bazę danych dla hurtowni podzespołów komputerowych. Przedstaw i omów pięć dowolnych zapytań wyszukujących.

Zadanie 3.12.

Za pomocą narzędzia MongoDB Compass wykonaj eksport i import danych zapisanych w bazie danych.

Pytania kontrolne

1. Przedstaw wady i zalety relacyjnych baz danych i baz NoSQL.
2. Jakich kryteriów można użyć do budowania zapytań?
3. W jaki sposób uaktualnić dokument?
4. Co się stanie, jeśli w zapytaniu aktualizującym dokument pominiemy parametr `$set`?



Przykład użycia środowiska Node.js i frameworka Express w połączeniu z bibliotekami Bootstrap i jQuery

Nasz cel to stworzenie quizu, który będzie wyświetlał cztery pytania. Udzielenie błędnej odpowiedzi na którekolwiek z pytań kończy quiz. Wygrana następuje po udzieleniu poprawnych odpowiedzi na wszystkie pytania.

Pytania przesyła serwer, który po udzieleniu odpowiedzi odsyła również informację zwrotną — czy odpowiedź jest poprawna, czy błędna.

Quiz stworzymy z wykorzystaniem środowiska Node.js i frameworka Express. Biblioteka jQuery i framework Bootstrap zostaną użyte do stworzenia oprawy graficznej.

4.1. Utworzenie plików i przygotowanie serwera Node.js w oparciu o framework Express

Rozpoczynamy od przygotowania plików. Plik *index.html* odpowiada za wyświetlenie treści pytań i odpowiedzi. Ze stroną zostaje połączony arkusz stylów CSS (plik *style.css*), który formatuje wygląd strony, oraz plik *script.js*, w którym zostanie umieszczony kod napisany w JavaScripcie sterujący działaniem quizu. Listingi wszystkich trzech plików są przedstawione poniżej.

Na listingu 4.1 jest pokazana zawartość pliku *index.html*. W strukturze pliku osadzono nagłówek `<h1>` wyświetlający napis `Quiz`, kontener o identyfikatorze równym `pytanie`, wewnątrz którego zostanie umieszczone pytanie, oraz cztery przyciski `<button>` z przyporządkowanymi identyfikatorami. Użyte kody `link` i `script` łączą ze stroną arkusz stylów i plik skryptu.

Listing 4.1. Zawartość pliku index.html

```
<!DOCTYPE html>
<html lang="pl">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
shrink-to-fit=no">
    <title>Quiz</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <h1>Quiz</h1>

    <div id="pytanie"> </div>
    <button id="odp_1"></button>
    <button id="odp_2"></button>
    <button id="odp_3"></button>
    <button id="odp_4"></button>

    <script src="script.js"></script>
</body>

</html>
```

WSKAZÓWKA

Aby wygenerować strukturę strony HTML, należy po utworzeniu nowego pliku (koniecznie z rozszerzeniem *.html*) i otwarciu go w oknie *Visual Studio Code* wpisać znak ! i zatwierdzić klawiszem *Enter*.

Wpisanie w oknie edycji pliku *.html* button*4 i zatwierdzenie klawiszem *Enter* wygeneruje kod <button></button><button></button><button></button><button></button>, a wpisanie div#pytanie po zatwierdzeniu utworzy kod <div id="pytanie"></div>.

Plik *style.css* ustawia kolor tła strony na aliceblue (listing 4.2).

Listing 4.2. Zawartość pliku style.css

```
body {
    background-color: aliceblue;
}
```

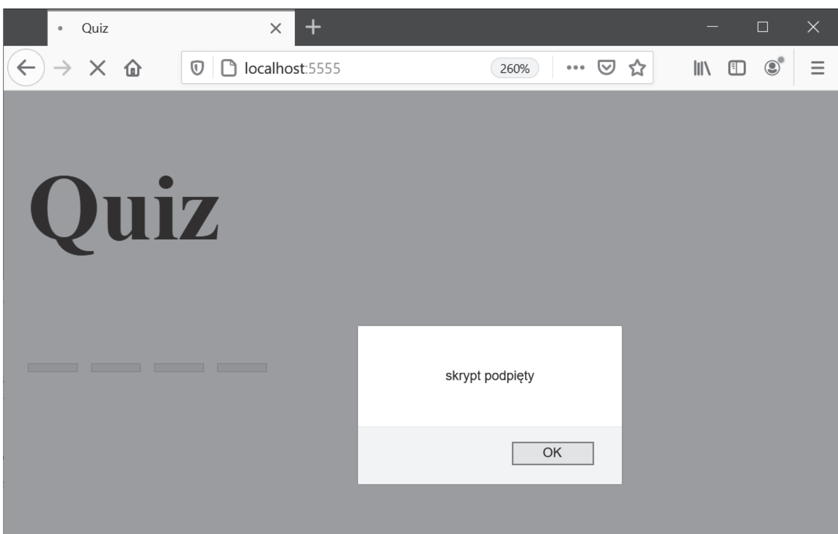
Plik *script.js* zawiera tylko jedną linijkę kodu, wyświetlającą okno typu *alert*, która została użyta wyłącznie do zweryfikowania poprawności podpięcia (listing 4.3).

Listing 4.3. Zawartość pliku script.js

```
alert('skrypt podpięty');
```

Na rysunku 4.1 przedstawiono efekt użycia kodów z wszystkich trzech listingów.

Wszystko działa. Możemy zrobić następny krok — połączyć utworzone pliki z Node.js.



Rysunek 4.1. Przygotowanie i połączenie plików będących szkieletem tworzonej aplikacji

Rozpoczynamy od utworzenia projektu o nazwie `quiz` (plik *app.js*) oraz zainstalowania pakietu *nodemon* i frameworka *Express*.

Utworzone do tej pory pliki zostają umieszczone we wspólnym katalogu *www* wraz z ikoną favicon. Do zainicjalizowania serwera posłużył kod z listingu 4.4.

Najpierw zaimportujemy do projektu framework Express i powiążemy go z naszą aplikacją (tak jak to zostało przedstawione w punkcie 3.5.1). Utworzone stałe `host` i `port` zostaną użyte przy uruchomieniu serwera HTTP, a stała `www` wskazuje na katalog, w którym znajdują się pliki. Stała `www` jest stosowana wraz z middleware `express.static` (zobacz punkt 3.5.3).

Za pomocą metody `.listen()` zostaje uruchomiony serwer, a w konsoli jest wyświetlany jego adres URL.

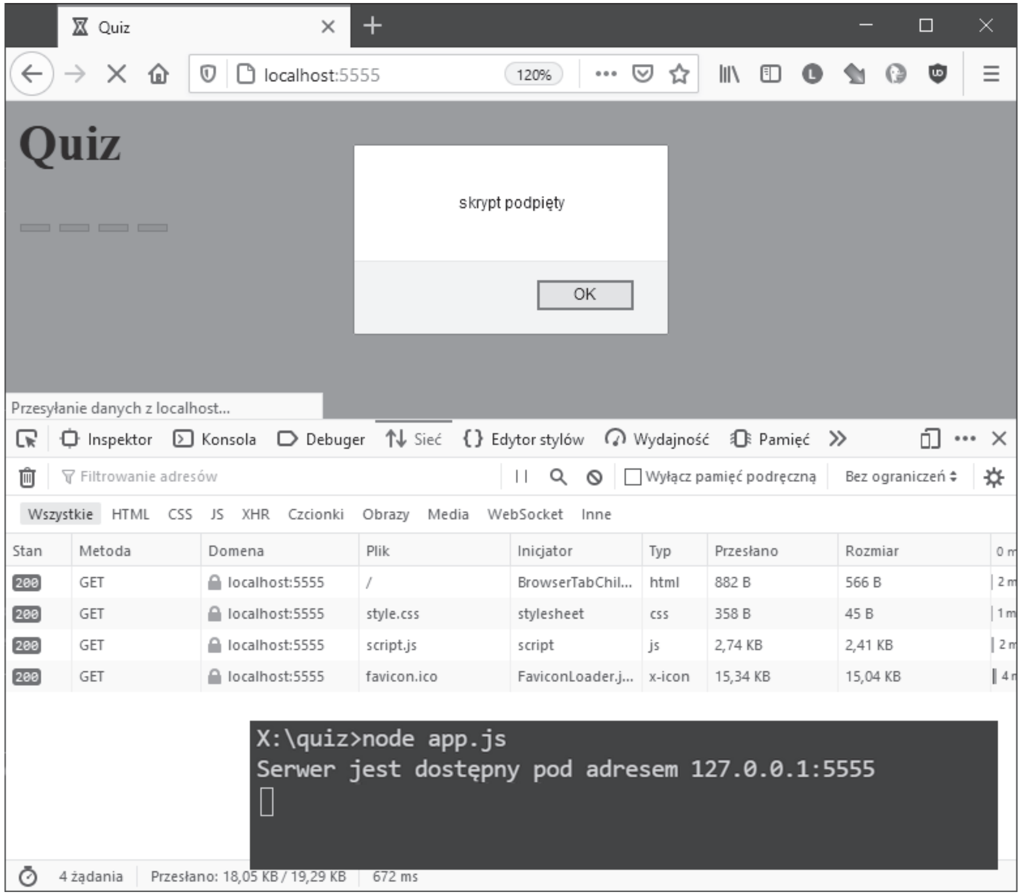
Listing 4.4. Serwer HTTP utworzony z użyciem frameworka Express — pliki statyczne (plik app.js)

```
const express = require('express');

const app = express();

const host = '127.0.0.1';
const port = 5555;
const www = __dirname + '\\www';

app.listen(port, () => {
    console.log(`Serwer jest dostępny pod adresem ${host}:${port}`)
});

app.use(express.static(www));
```

Działanie kodu jest pokazane na rysunku 4.2. Po wywołaniu adresu serwera wszystkie pliki statyczne (z katalogu *www*) zostają przesłane do przeglądarki.

4.2. Przesłanie informacji z backendu do frontendu. Użycie metody .fetch(). Czym jest obietnica?

Pierwszym wyzwaniem, przed jakim stajemy, jest **przesłanie treści pytań**. Pytania są definiowane po stronie serwera. Aby można je było wyświetlić, serwer musi je przekazać, lecz zanim to nastąpi, trzeba ustalić ich treść. Pytania zostają zapisane w formacie JSON (w tablicy obiektów) — do pliku *app.js* należy dodać kod pokazany poniżej (listing 4.5). Dodatkowo została zdefiniowana zmienna `przekazanePytanie`, odpowiedzialna za przekazywanie kolejnych pytań. Jej wartość zostaje ustalona na `0`, ponieważ rozpoczynamy od pierwszego pytania (pamiętamy, że pytania znajdują się w tablicy — indeks pierwszego pytania to `0`).



Rysunek 4.2. Uruchomienie serwera HTTP

Listing 4.5. Definicja pytań (plik app.js)

```
let przekazanePytanie = 0;
const pytania = [
    {
        pytanie: 'Wartość i typ zmiennej w języku PHP sprawdzisz za pomocą
funkcji?',
        odpowiedz: ['readfile()', 'var_dump()', 'implode()', 'strlen()'],
        poprawnaOdpowiedz: 1,
    },
    {
        pytanie: 'Instrukcja for może być zastąpiona instrukcją?',
        odpowiedz: ['foreach', 'switch', 'break', 'case'],
        poprawnaOdpowiedz: 0,
    },
    {
```

```
        pytanie: 'Prosta animacja może być zapisana w formacie?',
        odpowiedz: ['GIF', 'PNG', 'BMP', 'JPG'],
        poprawnaOdpowiedz: 0,
    },
    {
        pytanie: 'Użytkownik wprowadził adres zasobu, którego nie ma na
serwerze. Próba połączenia wygeneruje błąd?',
        odpowiedz: ['400', '503', '500', '404'],
        poprawnaOdpowiedz: 3,
    },
]
```

Aby pytania mogły zostać przesłane, zostaje określona ścieżka */quiz_pytanie.* Po jej wybraniu przeglądarka otrzyma treść pytania. Treść pytania wraz z zaproponowanymi odpowiedziami i poprawną odpowiedzią jest przechowywana w stałej `kolejnePytanie`. To, które pytanie zostanie pobrane, zależy od bieżącej wartości zmiennej `przekazanePytanie`. Wybrany sposób przechowania informacji o pobranym pytaniu to najprostsze rozwiązanie. Ma on swoje konsekwencje: brak powiązania tej informacji z sesją. Oznacza to, że rozwiązywanie quizu przez dwie osoby jednocześnie jest niemożliwe, a ponowne uruchomienie quizu (po wygranej bądź przegranej) będzie wymagało zrestartowania serwera.

Po otrzymaniu zapytania odpowiedź zostaje wysłana za pomocą metody `.json()`. Ponadto w konsoli jest wyświetlana udzielona odpowiedź (listing 4.6).

Listing 4.6. Pobranie treści pytania (plik app.js)

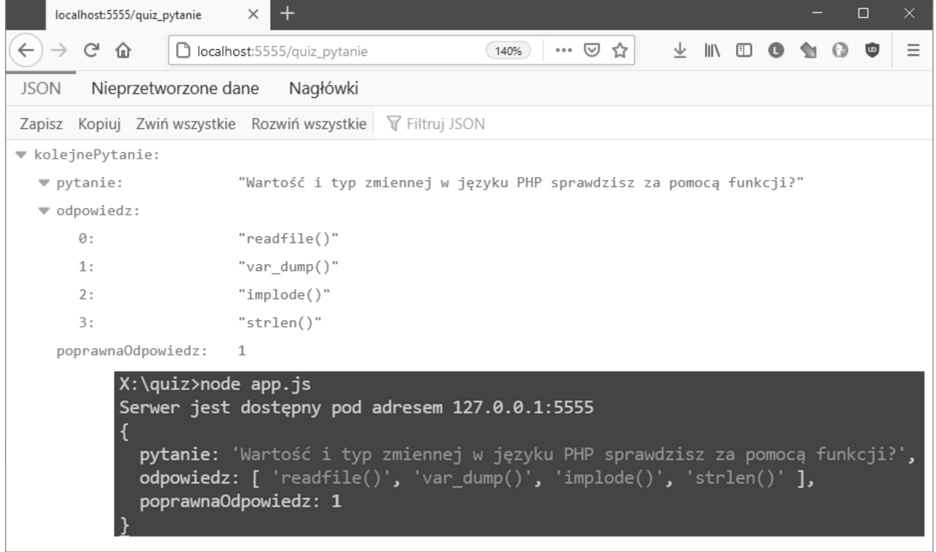
```
app.get('/quiz_pytanie', (req, res) => {
    const kolejnePytanie = pytania[przekazanePytanie];
    console.log(kolejnePytanie);
    res.json({
        kolejnePytanie,
    });
});
```

Po przejściu na stronę *http://localhost:5555/quiz_pytanie* zostanie wyświetlone pytanie — obiekt JSON (rysunek 4.3).

Zadanie 4.1.

W treści pytania jest przekazywana poprawna odpowiedź, co sprawia, że można ją podejrzeć. Czy istnieje sposób na jej ukrycie?

Całość kodu w pliku *app.js* po zmianach jest pokazana na listingu 4.7 (pogrubioną czcionką zaznaczono kod, który różni się od kodu z listingu 4.4).



Rysunek 4.3. Przekazane pytanie

Listing 4.7. Zawartość pliku app.js

```
const express = require('express');

const app = express();

const host = '127.0.0.1';
const port = 5555;
const www = __dirname + '\\www';

app.listen(port, () => {
    console.log(`Serwer jest dostępny pod adresem ${host}:${port}`)
});

app.use(express.static(www));

let przekazanePytanie = 0;

const pytania = [
    {
        pytanie: 'Wartość i typ zmiennej w języku PHP sprawdzisz za pomocą
funkcji?',
        odpowiedz: ['readfile()', 'var_dump()', 'implode()', 'strlen()'],
        poprawnaOdpowiedz: 1,
    },
    {
```

```
        pytanie: 'Instrukcja for może być zastąpiona instrukcją?',
        odpowiedz: ['foreach', 'switch', 'break', 'case'],
        poprawnaOdpowiedz: 0,
    },
    {
        pytanie: 'Prosta animacja może być zapisana w formacie?',
        odpowiedz: ['GIF', 'PNG', 'BMP', 'JPG'],
        poprawnaOdpowiedz: 0,
    },
    {
        pytanie: 'Użytkownik wprowadził adres zasobu, którego nie ma na
serwerze. Próba połączenia wygeneruje błąd?',
        odpowiedz: ['400', '503', '500', '404'],
        poprawnaOdpowiedz: 3,
    },
]
app.get('/quiz_pytanie', (req, res) => {
    const kolejnePytanie = pytania[przekazanePytanie];
    console.log(kolejnePytanie);
    res.json({
        kolejnePytanie,
    });
});
```

Naszym następnym celem jest wyświetlenie treści pytania wraz z odpowiedziami. Aby móc wykonać zadanie, musimy odwołać się do elementów strony *index.html* — kontenera `<div>` o identyfikatorze równym `pytanie` oraz czterech przycisków `<button>`.

W pliku *script.js* należy umieścić pierwszą część kodu, pokazaną na listingu 4.8.

Listing 4.8. Pobranie elementów <button> (plik script.js)

```
const pytanie = document.querySelector('#pytanie');
const odp_1 = document.querySelector('#odp_1');
const odp_2 = document.querySelector('#odp_2');
const odp_3 = document.querySelector('#odp_3');
const odp_4 = document.querySelector('#odp_4');
```

Pobranie elementu i przypisanie go do stałej jest realizowane za pomocą metody `.querySelector()` połączonej z identyfikatorem pobieranego elementu. Z użyciem tej metody pobierzemy również elementy określonej klasy bądź określonego znacznika HTML. Metoda `.querySelector()` zwraca **pierwszy** pasujący do selektora element, a jeśli nic nie znajdzie, zwraca **`null`**.

WSKAZÓWKA

Do pobrania elementu można także wykorzystać metodę `.getElementById()`. Wówczas na przykład kod `const pytanie = document.querySelector('#pytanie');` należy zastąpić kodem `const pytanie = document.getElementById('pytanie');` (nazwa identyfikatora bez znaku `#`). Aby wybrać element zdefiniowany jako klasę, należy posłużyć się metodą `.getElementsByClassName()`, aby zaś pobrać element o danym znaczniku, trzeba skorzystać z metody `.getElementsByTagName()`. Metoda `.getElementsByName()` pobiera elementy o danej wartości atrybutu `name`.

Aby wyświetlić treść pytania i odpowiedzi, posłużono się kodem z listingu 4.9.

Listing 4.9. Pobranie i wyświetlenie pytania (plik script.js)

```
function pokazNastepnePytanie() {
    fetch('/quiz_pytanie').then(odp => odp.json())
        .then(dane => {
            uzupelnijPytanie(dane);
        });
}

pokazNastepnePytanie();

function uzupelnijPytanie(dane) {
    pytanie.innerText = dane.pytanie;
    odp_1.innerText = dane.odpowiedz[0];
    odp_2.innerText = dane.odpowiedz[1];
    odp_3.innerText = dane.odpowiedz[2];
    odp_4.innerText = dane.odpowiedz[3];
}
```

Utworzono dwie funkcje: `pokazNastepnePytanie()` i `uzupelnijPytanie()`. Obie funkcje odpowiadają za obsługę pytań.

Funkcja `pokazNastepnePytanie()` do pobrania pytań wykorzystuje metodę `.fetch()`.

Podstawowa składnia metody `.fetch()` ma postać: **`fetch(url, [opcje]);`**.

Aby pobrać pytanie, należy wskazać adres URL, pod którym się ono znajduje (domyślnie wykonywane jest żądanie *GET*).

Efektem wywołania metody `.fetch()` jest zwrócenie **obietnicy** (ang. *promise*).

Obietnica to obiekt służący do obsługi kodu asynchronicznego. Reprezentuje stany powodzenie lub błąd, które **będą dostępne w przyszłości**.

Aby utworzyć obiekt typu *promise*, należy użyć konstruktora o takiej właśnie nazwie, do którego przekazujemy funkcję wywołania zwrotnego zawierającą dwie metody: `resolve` i `reject`. Metody te będą wywoływane w przypadku pomyślnego spełnienia obietnicy lub wystąpienia błędu.

Przykład użycia obietnicy jest pokazany na listingu 4.10. Obietnica losuje liczbę, a następnie zaokrągla ją do dwóch miejsc po przecinku. Jeśli wylosowana liczba jest większa od 0,5, obietnica kończy się powodzeniem (`resolve`), w przeciwnym razie zostaje zgłoszony błąd (`reject`).

Po zakończeniu obietnicy możemy zareagować na jej wynik (skonsumować ją). Służą do tego metody `.then()` i `.catch()`. Pierwsza obsłuży operacje zakończone powodzeniem (ale również błędem), druga jest przeznaczona do zarządzania błędami — w momencie kiedy *promise* jest odrzucany.

Listing 4.10. Obietnica

```
const obietnica = new Promise((resolve, reject) => {
    const wylosowanaLiczba = Math.random().toFixed(2);

    if (wylosowanaLiczba > 0.5) {
        resolve(wylosowanaLiczba);
    } else {
        reject(wylosowanaLiczba);
    }
});

obietnica.then(wylosowanaLiczba => {
    console.log(`Wylosowana liczba to: ${wylosowanaLiczba}, obietnica
spełniona`);
}).catch(wylosowanaLiczba => { console.log(`Wylosowana liczba to:
${wylosowanaLiczba}, obietnica nie spełniona`); })
```

WSKAZÓWKA

Aby pobrać zasób z serwera, można wykorzystać odwołanie do obiektu XMLHttpRequest (starsza metoda).

Funkcja `uzupelnijPytanie()` odpowiada za wstawienie treści do kontenera `<div>` wyświetlającego treść pytania oraz odpowiedzi, które pojawiają się na przyciskach.

Zmienił się również kod w pliku *app.js* (odpowiedź serwera po otrzymaniu żądania wyświetlenia zasobu strony */quiz_pytanie*). Utworzono dwie zmienne: `pytanie` i `odpowiedz`, które zostają wysłane do przeglądarki (listing 4.11). Modyfikacja kodu sprawiła, że wraz z pytaniem i odpowiedziami nie jest przesyłana poprawna odpowiedź, inaczej niż z użyciem kodu z listingu 4.10.

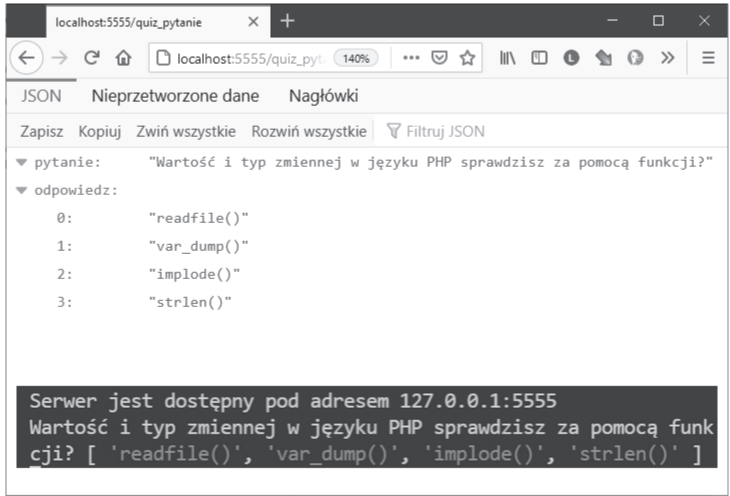
Listing 4.11. Pobranie treści pytania (plik app.js)

```
app.get('/quiz_pytanie', (req, res) => {
    const kolejnePytanie = pytania[przekazanePytanie];
    let pytanie = pytania[przekazanePytanie].pytanie;
    let odpowiedz = pytania[przekazanePytanie].odpowiedz;
    res.json({
        pytanie, odpowiedz,
    });
    console.log(pytanie, odpowiedz);
});
```

Zadanie 4.2.

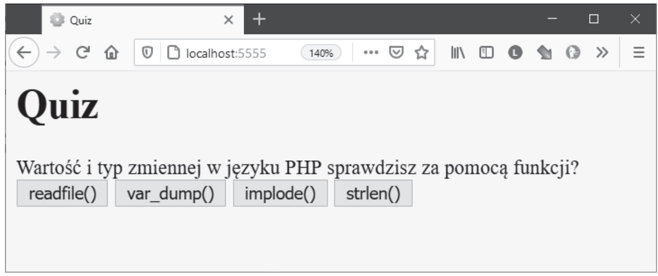
Czy obie zmienne można zadeklarować w inny sposób? (Wskazówki szukaj w punkcie 3.5.2).

Działanie kodu z listingu 4.11 jest pokazane na rysunku 4.4.



Rysunek 4.4. Przekazanie pytania

Efekt zmian wprowadzonych w kodzie powinien być taki jak na rysunku 4.5 — powinna być wyświetlona treść pytania wraz z odpowiedziami.



Rysunek 4.5. Przekazanie pytania i wyświetlenie jego treści wraz z odpowiedziami

4.3. Odpowiedź frontendu — użycie atrybutu data, definicja zdarzenia i ponownie metoda .fetch()

Jesteśmy w miejscu, w którym serwer przesyła pytanie wraz z odpowiedziami. Następny krok to **przesłanie udzielonej odpowiedzi**.

Aby sprawdzić, jaka odpowiedź została udzielona, w pliku *index.html* w definicji każdego przycisku definiujemy atrybut `data-odp`.

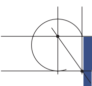
Własne atrybuty `data` są wykorzystywane do przechowywania niestandardowych danych, które mogą być użyte na przykład do zidentyfikowania elementu. Można je kojarzyć z dowolnym elementem HTML.

Atrybut `dataset` powinien rozpoczynać się od prefiksu `data-`, po którym następuje jego nazwa. Tak utworzony atrybut możemy uchwycić w kodzie JavaScript.

Przykład użycia atrybutu `data` jest pokazany na listingu 4.12. Na stronie są wyświetlane trzy przyciski, z których każdy ma ustawiony atrybut `data-kolor`. Kliknięcie przycisku wywołuje zdarzenie `onClick` i przekazanie przycisku jako argumentu do funkcji `pokazKolor()`. Funkcja za pomocą metody `.getAttribute()` pobiera argument `data-kolor` i wyświetla go w oknie typu *alert*.

Listing 4.12. Użycie atrybutu dataset

```
<!DOCTYPE html>
<html>

<head>
    <script>
        function pokazKolor(kolor) {
            let wybranyKolor = kolor.getAttribute("data-kolor");
            alert("Wybranym kolorem jest: " + wybranyKolor + ".");
        }
    </script>
</head>

<body>
    <p>Wybierz kolor:</p>

    <button onClick="pokazKolor(this)" data-kolor="czerwony">Czerwony</button>
    <button onClick="pokazKolor(this)" data-kolor="zielony">Zielony</button>
    <button onClick="pokazKolor(this)" data-kolor="niebieski">Niebieski</
button>
</body>

</html>
```

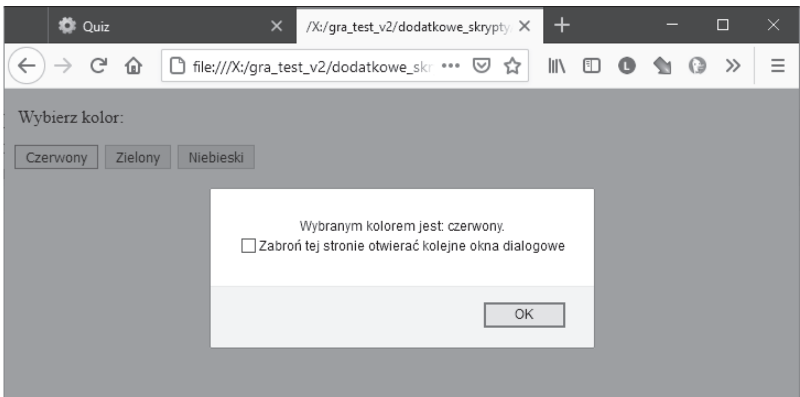


ZAPAMIĘTAJ

Wartość przechowywana w atrybucie `data` zawsze jest typu `string`. Użycie argumentu `data` pozwala przekazać wiele argumentów (nie musimy się ograniczać do jednego).

Obecna w bibliotece jQuery metoda `.data()` pozwala pracować z atrybutami `data`.

Użycie atrybutu `dataset` jest pokazane na rysunku 4.6.



Rysunek 4.6. Użycie atrybutu dataset

Zawartość pliku *index.html* po wszystkich zmianach jest pokazana na listingu 4.13 — do każdego przycisku został dodany atrybut `data-odp`, przechowujący numer przycisku (rozpoczynamy od `0`).

Listing 4.13. Zawartość pliku index.html

```
<!DOCTYPE html>
<html lang="pl">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
shrink-to-fit=no">
    <title>Quiz</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <h1>Quiz</h1>

    <div id="pytanie"> </div>
    <button id="odp_1" data-odp="0"></button>
```

```
    <button id="odp_2" data-odp="1"></button>
    <button id="odp_3" data-odp="2"></button>
    <button id="odp_4" data-odp="3"></button>

    <script src="script.js"></script>
</body>

</html>
```

Wiemy już, która odpowiedź została udzielona. Teraz należy ją zwrócić do backendu (zobacz punkt 3.5.2).

Rozpoczynamy od zdefiniowania zdarzenia, które wywoła kliknięcie przycisku, a tym samym zainicjuje proces przesyłania odpowiedzi.

Ponieważ każdy przycisk jest reprezentowany przez stałą (listing 4.8), możemy ją wykorzystać w pliku *script.js* do przypisania zdarzenia (listing 4.14).

Listing 4.14. Definicja zdarzenia po wybraniu przycisku — dla każdego przycisku z osobna (plik script.js)

```
odp_1.addEventListener('click', function () {
    const nrOdp = this.dataset.odp;
    wyslijOdp(nrOdp);
})

odp_2.addEventListener('click', function () {
    const nrOdp = this.dataset.odp;
    wyslijOdp(nrOdp);
})

odp_3.addEventListener('click', function () {
    const nrOdp = this.dataset.odp;
    wyslijOdp(nrOdp);
})

odp_4.addEventListener('click', function () {
    const nrOdp = this.dataset.odp;
    wyslijOdp(nrOdp);
})
```

Do każdego przycisku zostaje dodane zdarzenie, które w momencie wybrania go przypisze do stałej `nrOdp` numer wciśniętego przycisku (dzięki użyciu parametru `dataset`). Przypisana wartość jest przekazywana dalej, do funkcji `wyslijOdp()`.

Zaproponowany powyżej kod możemy zapisać inaczej, z użyciem pętli. W pierwszym kroku za pomocą metody `.querySelectorAll()` zostaje pobrana cała kolekcja przycisków. Kolekcja jest przekazywana do pętli `for of` (listing 2.17), która w każdym przebiegu do przycisku przypisuje to samo zdarzenie co w kodzie powyżej (listing 4.15).

Listing 4.15. Definicja zdarzenia po wybraniu przycisku — pętla (plik script.js)

```
const przyciski = document.querySelectorAll('button');

for (const przycisk of przyciski) {
    przycisk.addEventListener('click', function() {
        const nrOdp = this.dataset.odp;
        wyslijOdp(nrOdp);
    })
}
```

Odpowiedź na udzielone pytanie przekazuje funkcja `wyslijOdp()`, która jako argument otrzymuje stałą `nrOdp`. Funkcja realizuje jeszcze jedno zadanie: odbiera od serwera informacje o statusie odpowiedzi — czy jest ona poprawna, czy nie. Kod realizujący wysyłanie (odbieranie) odpowiedzi jest pokazany na listingu 4.16:

Listing 4.16. Wysłanie odpowiedzi oraz informacja o statusie odpowiedzi (poprawna/niepoprawna; plik script.js)

```
function wyslijOdp(nrOdp) {
    fetch('/quiz_odpowiedz/' + nrOdp, {
        method: 'POST',
    }).then(odp => odp.json())
        .then(dane => {
            console.log(dane);
    });
}
```

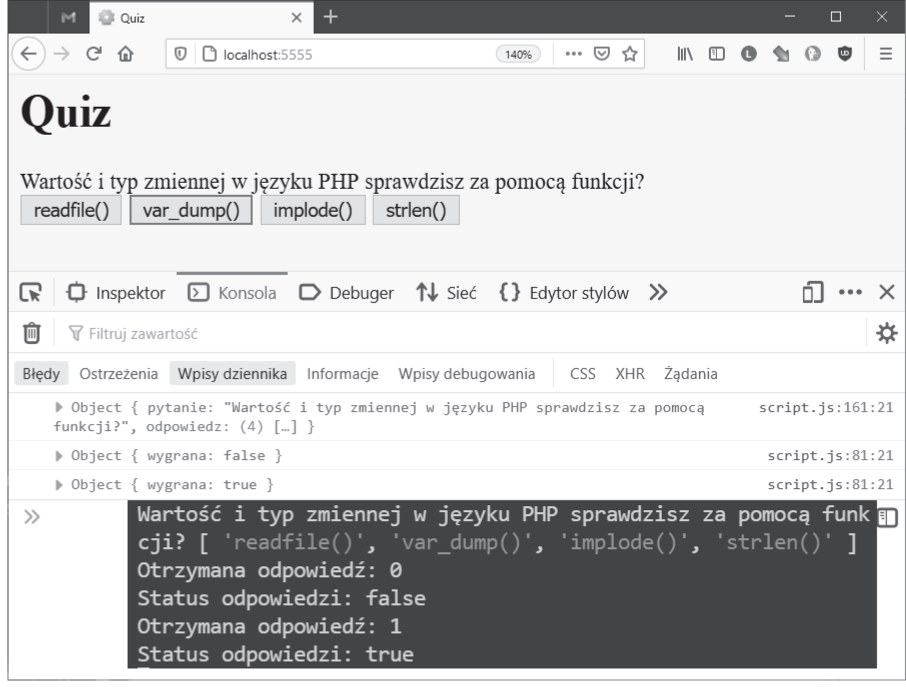
Ponownie posłużono się metodą `.fetch()` — do przesłania numeru wciśniętego przycisku użyto metody *POST*. Zwrócenie odpowiedzi jest możliwe dzięki użyciu parametru ścieżki (listing 3.24) — wartość stałej `nrOdp` zostaje doklejona do adresu `/quiz_odpowiedz/`.

Serwer (kod programu zaprezentowany poniżej — plik *app.js*) odczyta przekazaną wartość dzięki użyciu własności `req.params`. Stała `biezacePytanie` reprezentuje pytanie (treść, proponowane odpowiedzi oraz poprawną odpowiedź). Instrukcja `if` sprawdza, czy numer otrzymanej odpowiedzi jest zgodny z numerem odpowiedzi określonym w pytaniu. Jeśli numery są zgodne, odsyłana jest wartość `true` (format JSON), w przeciwnym razie — `false` (również format JSON). Ponieważ przekazany parametr jest typu `string`, musi on zostać przekształcony za pomocą funkcji `Number()` na liczbę. Dodatkowo otrzymana odpowiedź wraz z jej statusem jest wyświetlana w konsoli (listing 4.17).

Listing 4.17. Weryfikacja udzielonej odpowiedzi (plik app.js)

```
app.post('/quiz_odpowiedz/:i', (req, res) => {
    const { i } = req.params;
    const biezacePytanie = pytania[przekazanePytanie];
    console.log(`Otrzymana odpowiedź: ${i}`);
    console.log(`Status odpowiedzi: ${biezacePytanie.poprawnaOdpowiedz ===
Number(i)}`);
    if (biezacePytanie.poprawnaOdpowiedz === Number(i)) {
        res.json({
            wygrana: true,
        });
    } else {
        res.json({
            wygrana: false,
        });
    }
});
```

Dopisanie kodu sprawi, że po wybraniu przycisku jego wartość zostanie przesłana do serwera, a ten odeśle informację, czy udzielona odpowiedź jest poprawna (rysunek 4.7). Wciśnięto pierwszy przycisk (co jest błędem), dlatego serwer zwrócił `false`. W kroku drugim udzielono poprawnej odpowiedzi — zostaje zwrócona wartość `true`.



Rysunek 4.7. Przesłanie pytania i odesłanie odpowiedzi

Kod po zmianach jest pokazany poniżej (listingi od 4.18 do 4.20). Ponadto dodano kod, który sprawia, że zostaje wyświetlona liczba poprawnych odpowiedzi, a po udzieleniu poprawnej odpowiedzi jest wyświetlane następne pytanie.

Dodany znacznik `<p>` w połączeniu ze `<span>` pozwoli wyświetlić liczbę poprawnych odpowiedzi.

Listing 4.18. Plik index.html

```
<!DOCTYPE html>
<html lang="pl">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
shrink-to-fit=no">
    <title>Quiz</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <h1>Quiz</h1>
    <p>Poprawne odpowiedzi: <span id="poprawna">0</span></p>
    <div id="pytanie"> </div>
    <button id="odp_1" data-odp="0"></button>
    <button id="odp_2" data-odp="1"></button>
    <button id="odp_3" data-odp="2"></button>
    <button id="odp_4" data-odp="3"></button>

    <script src="script.js"></script>
</body>

</html>
```

Po udzieleniu poprawnej odpowiedzi wartość zmiennej `przekazanePytanie` jest zwiększana o 1, co sprawia, że może zostać przesłane następne pytanie — wartość zmiennej jest przesyłana wraz z informacją o udzieleniu poprawnej odpowiedzi.

Listing 4.19. Plik app.js

```
const express = require('express');
const app = express();

const host = '127.0.0.1';
const port = 5555;
const www = __dirname + '\\www';
```

```
app.listen(port, () => {
    console.log(`Serwer jest dostępny pod adresem ${host}:${port}`)
});

app.use(express.static(www));

let przekazanePytanie = 0;

const pytania = [
    {
        pytanie: 'Wartość i typ zmiennej w języku PHP sprawdzisz za pomocą
funkcji?',
        odpowiedz: ['readfile()', 'var_dump()', 'implode()', 'strlen()'],
        poprawnaOdpowiedz: 1,
    },
    {
        pytanie: 'Instrukcja for może być zastąpiona instrukcją?',
        odpowiedz: ['foreach', 'switch', 'break', 'case'],
        poprawnaOdpowiedz: 0,
    },
    {
        pytanie: 'Prosta animacja może być zapisana w formacie?',
        odpowiedz: ['GIF', 'PNG', 'BMP', 'JPG'],
        poprawnaOdpowiedz: 0,
    },
    {
        pytanie: 'Użytkownik wprowadził adres zasobu, którego nie ma na
serwerze. Próba połączenia wygeneruje błąd?',
        odpowiedz: ['400', '503', '500', '404'],
        poprawnaOdpowiedz: 3,
    },
]

app.get('/quiz_pytanie', (req, res) => {
    if (przekazanePytanie === pytania.length) {
        res.json({
            wygrana: true,
        });
    } else  {
        const kolejnePytanie = pytania[przekazanePytanie];
        let pytanie = pytania[przekazanePytanie].pytanie;
        let odpowiedz = pytania[przekazanePytanie].odpowiedz;
        res.json({
            pytanie, odpowiedz,
        });
        console.log(pytanie, odpowiedz);
    }
});
```

```
app.post('/quiz_odpowiedz/:i', (req, res) => {
    const { i } = req.params;
    const biezacePytanie = pytania[przekazanePytanie];
    console.log(`Otrzymana odpowiedź: ${i}`);
    console.log(`Status odpowiedzi: ${biezacePytanie.poprawnaOdpowiedz ===
Number(i)}`);
    if (biezacePytanie.poprawnaOdpowiedz === Number(i)) {
        przekazanePytanie++;
        res.json({
            wygrana: true,
            przekazanePytanie,
        });
    }
    else {
        res.json({
            wygrana: false,
        });
    }
});
```

Ze stałą `liczbaOdpowiedzi` zostaje powiązany element o identyfikatorze `poprawna` (znacznik `<span>` w pliku *index.html*). Dodatkowa funkcja `odpZwrotna()` aktualizuje liczbę poprawnych odpowiedzi i jednocześnie wywołuje funkcję `pokazNastepnePytanie()` — zostaje pobrane następne pytanie. Funkcję `odpZwrotna()` wywołuje zrealizowana obietnica — gdy odpowiedź na zadane pytanie jest poprawna.

Listing 4.20. Plik script.js

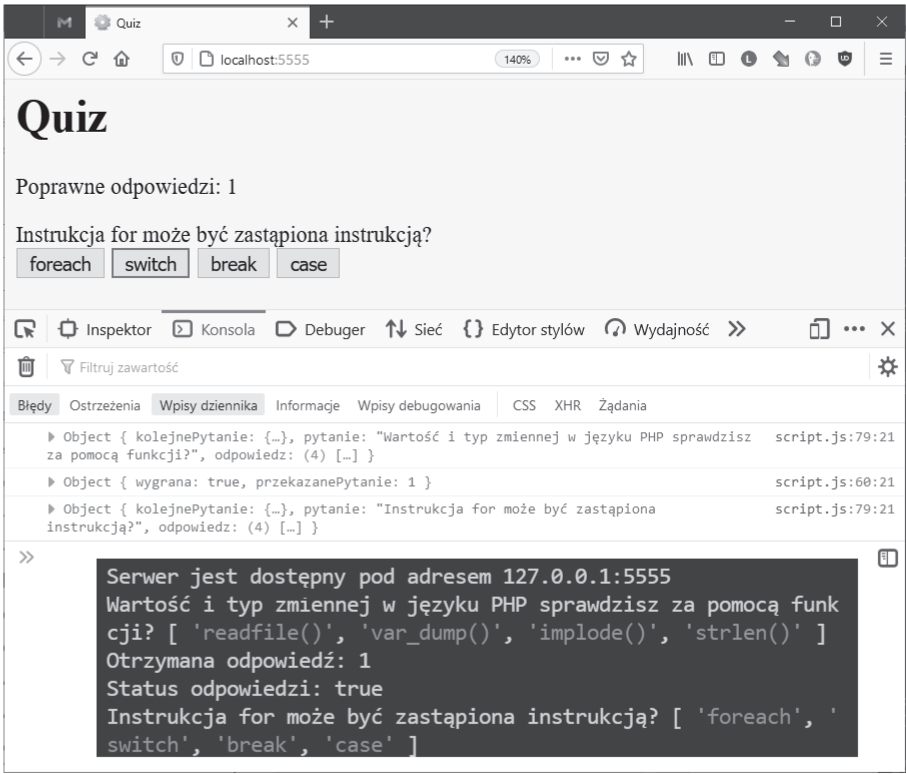
```
const pytanie = document.querySelector('#pytanie');
const odp_1 = document.querySelector('#odp_1');
const odp_2 = document.querySelector('#odp_2');
const odp_3 = document.querySelector('#odp_3');
const odp_4 = document.querySelector('#odp_4');

function uzupelnijPytanie(dane) {
    pytanie.innerText = dane.pytanie;
    odp_1.innerText = dane.odpowiedz[0];
    odp_2.innerText = dane.odpowiedz[1];
    odp_3.innerText = dane.odpowiedz[2];
    odp_4.innerText = dane.odpowiedz[3];
}

const liczbaOdpowiedzi = document.querySelector('#poprawna');

function odpZwrotna(dane) {
    liczbaOdpowiedzi.innerText = dane.przekazanePytanie;
    pokazNastepnePytanie();
}
```

```
function wyslijOdp(nrOdp) {
    fetch('/quiz_odpowiedz/' + nrOdp, {
        method: 'POST',
    }).then(odp => odp.json())
        .then(dane => {
            console.log(dane);
            odpZwrotna(dane);
        });
}

const przyciski = document.querySelectorAll('button');

for (const przycisk of przyciski) {
    przycisk.addEventListener('click', function () {
        const nrOdp = this.dataset.odp;
        wyslijOdp(nrOdp);
    })
}

function pokazNastepnePytanie() {
    fetch('/quiz_pytanie').then(odp => odp.json())
        .then(dane => {
            console.log(dane);
            uzupelnijPytanie(dane);
        });
}

pokazNastepnePytanie();
```

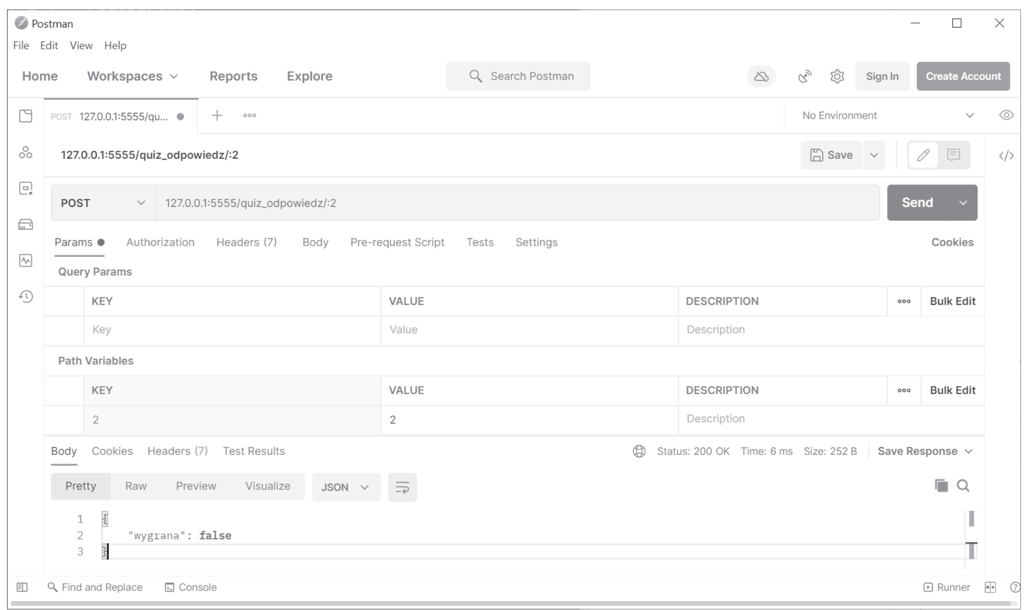
Działanie kodu jest pokazane na rysunku 4.8 — po udzieleniu poprawnej odpowiedzi zostaje wyświetlone następne pytanie.

WSKAZÓWKA

Do sprawdzenia odpowiedzi serwera na żądanie *POST* można użyć aplikacji Postman (*https://www.postman.com/*). Po zainstalowaniu i uruchomieniu narzędzia (rysunek 4.9) możemy określić adres, pod który zostanie wysłane żądanie (np. *127.0.0.1:5555/quiz_odpowiedz/:2*), i w ten sposób sprawdzać odpowiedzi udzielane przez serwer. Aplikacja oprócz standardowych metod *GET* i *POST* pozwala użyć m.in. *PUT*, *PATCH*, *DELETE* i *HEAD*.



Rysunek 4.8. Odpowiedzi otrzymane na udzielone pytanie i wyświetlenie następnego pytania



Rysunek 4.9. Użycie aplikacji Postman

Przedstawiony kod nie jest idealny — brakuje obsługi zdarzenia w przypadku, gdy zostanie wybrana błędna odpowiedź, a także po tym, jak gra zostanie zakończona (udzielenie poprawnych odpowiedzi na wszystkie wyświetlone pytania).

Rozpoczniemy od zmodyfikowania pliku *index.html*. Dodamy w nim element, który będzie wyświetlał komunikat (kontener o identyfikatorze `podsumowanie`) o przegranej bądź wygranej. Dodatkowo zostaje dodany drugi kontener, o id `plansza`, który posłuży do ukrycia treści pytania i przycisków. Uaktualniona zawartość pliku *index.html* jest pokazana na listingu 4.21.

Listing 4.21. Plik index.html

```
<!DOCTYPE html>
<html lang="pl">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
shrink-to-fit=no">
    <title>Quiz</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <h1>Quiz</h1>
    <p>Poprawne odpowiedzi: <span id="poprawna">0</span> </p>
    <div id="plansza">
        <div id="pytanie"> </div>
        <button id="odp_1" data-odp="0"></button>
        <button id="odp_2" data-odp="1"></button>
        <button id="odp_3" data-odp="2"></button>
        <button id="odp_4" data-odp="3"></button>
    </div>
    <div id="podsumowanie"></div>
    <script src="script.js"></script>
</body>

</html>
```

W pliku *script.js* należy uaktualnić kod, który obsłuży brakujące zdarzenia (listing 4.22). Zostają wybrane elementy strony — kontenery o identyfikatorach `plansza` i `podsumowanie` oraz akapit. Stałe reprezentujące wybrane elementy są użyte w dwóch instrukcjach `if`. Pierwsza (w momencie wygranej — gdy udzielono poprawnych odpowiedzi na wszystkie pytania) odpowiada za wyświetlenie tekstu `Wygrana!!!`. Druga (błędna odpowiedź) wyświetla tekst `Niestety to błędna odpowiedź. Spróbuj jeszcze raz.` W obydwu przypadkach zostają ukryte przyciski i licznik poprawnych odpowiedzi.

Listing 4.22. Uaktualniony plik script.js

```
const pytanie = document.querySelector('#pytanie');
const odp_1 = document.querySelector('#odp_1');
const odp_2 = document.querySelector('#odp_2');
const odp_3 = document.querySelector('#odp_3');
const odp_4 = document.querySelector('#odp_4');

const plansza = document.querySelector('#plansza');
const p = document.querySelector('p');
const podsumowanie = document.querySelector('#podsumowanie');

function uzupelnijPytanie(dane) {
    if (dane.wygrana === true) {
        plansza.style.display = 'none';
        p.style.display = 'none';
        podsumowanie.innerText = 'Wygrana!!!';
        podsumowanie.classList.add('wygrana');
        return;
    }

    if (dane.przegrana === true) {
        plansza.style.display = 'none';
        p.style.display = 'none';
        podsumowanie.innerText = 'Niestety to błędna odpowiedź. Spróbuj jeszcze raz.';
        podsumowanie.classList.add('przegrana');
        return;
    }

    pytanie.innerText = dane.pytanie;
    odp_1.innerText = dane.odpowiedz[0];
    odp_2.innerText = dane.odpowiedz[1];
    odp_3.innerText = dane.odpowiedz[2];
    odp_4.innerText = dane.odpowiedz[3];
}

const liczbaOdpowiedzi = document.querySelector('#poprawna');

function odpZwrotna(dane) {
    liczbaOdpowiedzi.innerText = dane.przekazanePytanie;
    pokazNastepnePytanie();
}
```

```
function wyslijOdp(nrOdp) {
    fetch('/quiz_odpowiedz/' + nrOdp, {
        method: 'POST',
    }).then(odp => odp.json())
        .then(dane => {
            console.log(dane);
            odpZwrotna(dane);
        });
}

const przyciski = document.querySelectorAll('button');

for (const przycisk of przyciski) {
    przycisk.addEventListener('click', function () {
        const nrOdp = this.dataset.odp;
        wyslijOdp(nrOdp);
    })
}

function pokazNastepnePytanie() {
    fetch('/quiz_pytanie').then(odp => odp.json())
        .then(dane => {
            uzupelnijPytanie(dane);
        });
}

pokazNastepnePytanie();
```

Wyświetlone komunikaty są stylizowane za pomocą klas `.wygrana` i `.przegrana`. Definicje użytych klas zostały dodane do pliku *style.css* (listing 4.23).

Listing 4.23. Zawartość pliku style.css

```
body {
    background-color: aliceblue;
}

.wygrana {
    font-family: "Lucida Sans Unicode", "Lucida Grande", sans-serif;
    font-size: 30px;
    letter-spacing: 1.8px;
    word-spacing: 2px;
    color: #000000;
    font-weight: 400;
    animation: blinker 2s linear infinite;
}

.przegrana {
    background-color: red;
```

```
    font-family: "Lucida Sans Unicode", "Lucida Grande", sans-serif;
    font-size: 30px;
    letter-spacing: 1.8px;
    word-spacing: 2px;
    color: #000000;
    font-weight: 400;
}

@keyframes blinker {
    50% {
        opacity: 0;
    }
}
```

Ostatnia zmiana dotyczy pliku *app.js*. Działanie instrukcji `if` zawartych w pliku *script.js* zależy od otrzymanej wartości `true` przekazanej w kluczu `przegrana` bądź `wygrana` (JSON). Zmienna `koniecGry` ustawiona na `true` kończy quiz (listing 4.24).

Listing 4.24. Zawartość pliku app.js

```
const express = require('express');

const app = express();

const host = '127.0.0.1';
const port = 5555;
const www = __dirname + '\\www';

app.listen(port, () => {
    console.log(`Serwer jest dostępny pod adresem ${host}:${port}`)
});

app.use(express.static(www));

let przekazanePytanie = 0;
let koniecGry = false;

const pytania = [
    {
        pytanie: 'Wartość i typ zmiennej w języku PHP sprawdzisz za pomocą
funkcji?',
        odpowiedz: ['readfile()', 'var_dump()', 'implode()', 'strlen()'],
        poprawnaOdpowiedz: 1,
    },
    {
        pytanie: 'Instrukcja for może być zastąpiona instrukcją?',
        odpowiedz: ['foreach', 'switch', 'break', 'case'],
        poprawnaOdpowiedz: 0,
    },
```

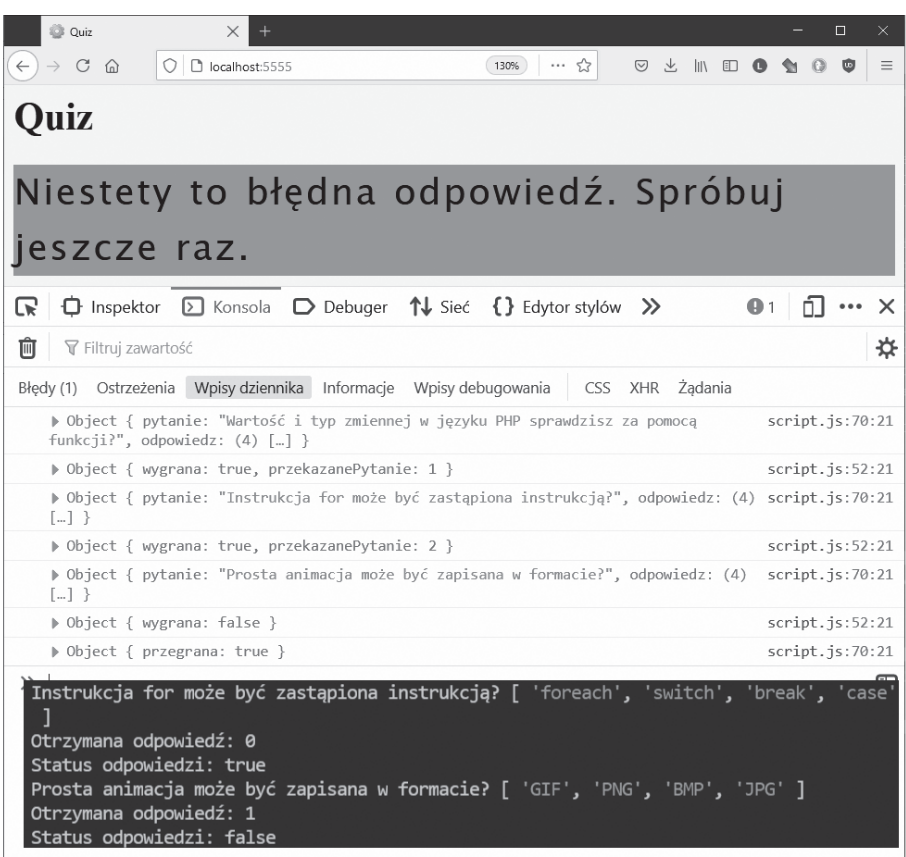
```
    {
        pytanie: 'Prosta animacja może być zapisana w formacie?',
        odpowiedz: ['GIF', 'PNG', 'BMP', 'JPG'],
        poprawnaOdpowiedz: 0,
    },
    {
        pytanie: 'Użytkownik wprowadził adres zasobu, którego nie ma na
serwerze. Próba połączenia wygeneruje błąd?',
        odpowiedz: ['400', '503', '500', '404'],
        poprawnaOdpowiedz: 3,
    },
]

app.get('/quiz_pytanie', (req, res) => {
    if (przekazanePytanie === pytania.length) {
        res.json({
            wygrana: true,
        });
        console.log(`Koniec gry. WYGRANA!!!`);
    } else
        if (koniecGry) {
            res.json({
                przegrana: true,
            });
        } else {
            let pytanie = pytania[przekazanePytanie].pytanie;
            let odpowiedz = pytania[przekazanePytanie].odpowiedz;
            res.json({
                pytanie, odpowiedz,
            });
            console.log(pytanie, odpowiedz);
        }
});

app.post('/quiz_odpowiedz/:i', (req, res) => {

    if (koniecGry) {
        res.json({
            przegrana: true,
        });
    }
    const { i } = req.params;
    const biezacePytanie = pytania[przekazanePytanie];
    console.log(`Otrzymana odpowiedź: ${i}`);
    console.log(`Status odpowiedzi: ${biezacePytanie.poprawnaOdpowiedz ===
Number(i)}`);
    if (biezacePytanie.poprawnaOdpowiedz === Number(i)) {
        przekazanePytanie++;
```

```
            res.json({
                wygrana: true,
                przekazanePytanie,
            });
        } else {
            res.json({
                wygrana: false,
            });
            koniecGry = true;
        }
});
```

Zadanie 4.3.

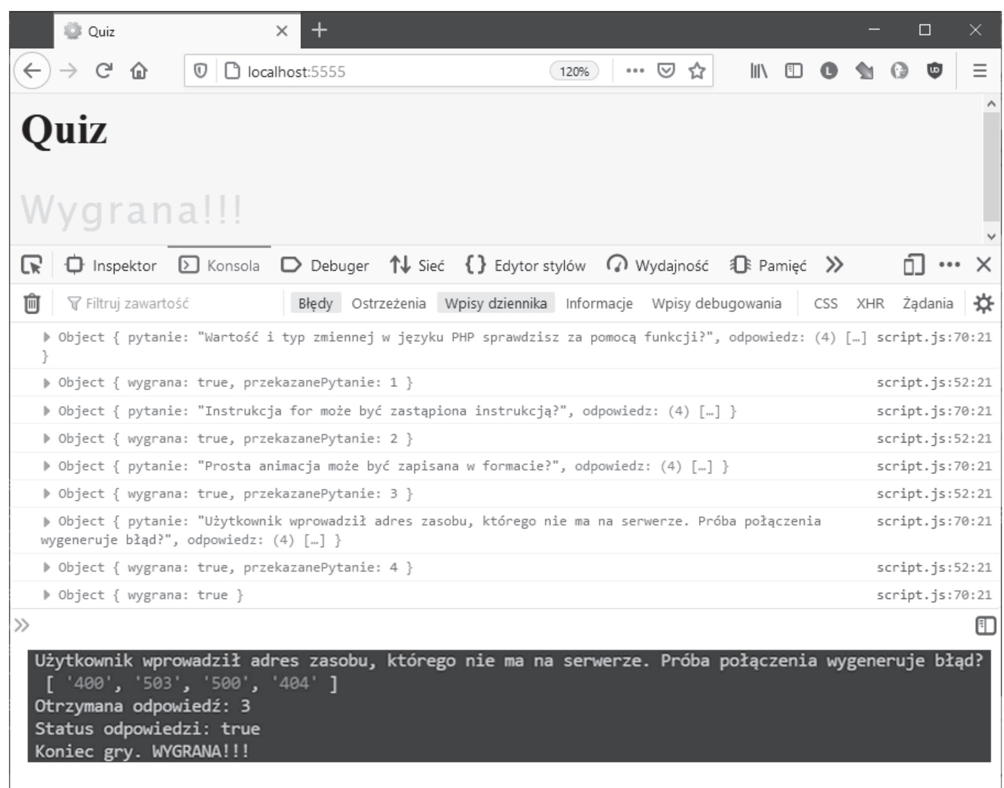
Kod odwołujący się do zdefiniowanych ścieżek wydziel jako osobny moduł i zaimportuj go do pliku *app.js*.

Udzielenie błędnej odpowiedzi skutkuje zakończeniem quizu — zostaje wyświetlony komunikat o niepowodzeniu (rysunek 4.10). Ponowne uruchomienie quizu wymaga zrestartowania serwera.



Rysunek 4.10. Udzielenie błędnej odpowiedzi

Udzielenie poprawnych odpowiedzi na wszystkie pytania również kończy quiz — zostaje wyświetlony migający napis `WYGRANA!!!` (rysunek 4.11).



Rysunek 4.11. Udzielenie poprawnych odpowiedzi na wszystkie pytania — wygrana

4.4. Szata graficzna — dołączamy framework Bootstrap i bibliotekę jQuery

Do działającego kodu quizu zostaje dołączony framework Bootstrap. Użycie go pozwala wystylizować wyświetlane elementy, a strona staje się responsywna. Dołączenie frameworka wymusiło aktualizację kodu w pliku *index.html*. Zmiany są pokazane na listingu 4.25.

Listing 4.25. Zawartość pliku index.html

```
<!DOCTYPE html>
<html lang="pl">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
shrink-to-fit=no">
    <title>Quiz</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/
bootstrap.min.css" rel="stylesheet" integrity="sha384-+0n0xVW2eSR5OomGNYDnhzAbD
sOXxcvSN1TPprVMTNDbiYZCxYbOOl7+AMvyTG2x" crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>
    <div id="container">

        <div class="row">
            <div class="col-md-4"></div>
            <div class="col-md-4 text-center">
                <h1>Quiz</h1>
            </div>
            <div class="col-md-4">
                <p class="h5 text-right m-2">Poprawne odpowiedzi: <span
id="poprawna">0</span></p>
            </div>
        </div>

        <div class="row">
            <div class="col-md-4"></div>
            <div id="plansza" class="col-md-4">
                <div id="pytanie" class="h4 py-2"> </div>
                <div class="text-center py-3">
                    <button id="odp_1" class="btn btn-primary m-2" data-
odp="0"></button>
                    <button id="odp_2" class="btn btn-primary m-2" data-
odp="1"></button>
                    <button id="odp_3" class="btn btn-primary m-2" data-
odp="2"></button>
                    <button id="odp_4" class="btn btn-primary m-2" data-
odp="3"></button>
                </div>
            </div>
            <div class="col-md-4"></div>
        </div>
        <div class="row">
            <div class="col-md-4"></div>
            <div id="podsumowanie" class="col-md-4 text-center my-4"></div>
            <div class="col-md-4"></div>
        </div>
    </div>
```

```
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/js/
bootstrap.bundle.min.js" integrity="sha384-gtEjrD/SeCtmISkJkNUaaKMoLD0//ElJ19sm
ozuHV6z3Iehds+3Ulb9Bn9Plx0x4" crossorigin="anonymous"></script>
    <script src="script.js"></script>
</body>

</html>
```

Oczywiście nic nie stoi na przeszkodzie, aby do quizu dołączyć również bibliotekę jQuery. To pozwoli użyć funkcji opisanych w rozdziale 1. Kod dołączający bibliotekę znajduje się przed zamknięciem sekcji `<body>`.

Aby sprawdzić, czy biblioteka jQuery działa, w pliku *script.js* umieszczono kod, który formatuje napis `Quiz` i ustawia kolor tła kontenera o identyfikatorze `plansza` (listing 4.26).

Listing 4.26. Dołączenie biblioteki jQuery (plik script.js)

```
$(function() {
    $('#plansza').css('background-color', '#00FFFF');
    $('h1').css('letter-spacing', '5px');
});
```

Ostateczny wygląd strony jest pokazany na rysunku 4.12.

Rysunek 4.12. Dołączenie bibliotek Bootstrap oraz jQuery

Zaproponowany przykład quizu może być wstępem do stworzenia większej aplikacji i może zostać rozbudowany o dodatkowe funkcje.

Zadanie 4.4.

Zmodyfikuj przedstawiony kod i zaproponuj własny wygląd strony wyświetlającej quiz.

Zadanie 4.5.

Zmodyfikuj kod tak, aby wraz z treścią pytania mogła zostać wyświetlona grafika.

Zadanie 4.6.

Zastanów się i podaj rozwiązania, które pozwoliłyby usprawnić działanie quizu. O jakie nowe funkcje może zostać on wzbogacony?

Pytania kontrolne

1. Za co odpowiada metoda `.json()`?
2. Wymień metody odpowiedzialne za pobranie elementu strony.
3. Opisz przeznaczenie i sposób działania metody `.fetch()`.
4. Czym jest obietnica (*promise*)?
5. Za co odpowiadają metody `.then()` i `.catch()`?
6. W jaki sposób korzystamy z atrybutu `data`?
7. Jaka metoda odpowiada za przypięcie zdarzenia do elementu strony?

Bibliografia

Literatura

Mike Cantelon, Marc Harter, TJ Holowaychuk, Nathan Rajlich, *Node.js w akcji*, Helion, Gliwice 2014.

Adam Freeman, *Angular. Profesjonalne techniki programowania, wyd. 2*, Helion, Gliwice 2018.

Adam Freeman, *TypeScript. Od początkującego do profesjonalisty*, Helion, Gliwice 2020.

Azat Mardan, *Full Stack JavaScript. Poznaj technologie Backbone.js, Node.js i MongoDB, wyd. 2*, Helion, Gliwice 2020.

Źródła internetowe

Bootstrap 3 Tutorial — *https://www.w3schools.com/bootstrap/*

Express — *https://www.codecademy.com/learn/learn-express*

Freecodecamp — *https://www.freecodecamp.org/*, *https://www.youtube.com/c/Freecodecamp*

GeeksforGeeks — *https://www.geeksforgeeks.org/*

jQuery user interface — *https://jqueryui.com/*

jQuery Tutorial — *https://www.w3schools.com/jquery/*

Nodejsera — *https://www.nodejsera.com*

Oficjalna dokumentacja Angular — *https://angular.io/docs*

Oficjalna dokumentacja Bootstrap — *https://getbootstrap.com/docs/4.6/getting-started/introduction/*

Oficjalna dokumentacja Express — *https://expressjs.com/en/5x/api.html*

Oficjalna dokumentacja jQuery — *https://api.jquery.com/*

Oficjalna dokumentacja MongoDB — *https://docs.mongodb.com/*

Oficjalna dokumentacja Node.js — *https://nodejs.org/en/docs/*

Oficjalna dokumentacja TypeScript — *https://www.typescriptlang.org/docs/*

Overment — *https://www.youtube.com/c/overment*

Programming with Mosh — *https://www.youtube.com/c/programmingwithmosh*

Reactgo — *https://reactgo.com/tutorials/angular/*

RisingStack Best of — The Most Popular Node.js Tutorials of 2017 — *https://blog.risingstack.com/the-most-popular-node-js-tutorials-of-2017/*

RisingStack Blog — *https://blog.risingstack.com/*

Samuraj Programowania — *https://www.youtube.com/c/SamurajProgramowania*

Technologie Internetowe dla developerów — *https://developer.mozilla.org/pl/docs/Web*

The Net Ninja — *https://www.youtube.com/c/TheNetNinja*

Skorowidz

A

adres URL, 232
Angular, 59
  błędy kompilacji, 68
  generowanie projektu, 64
  okno aplikacji, 67
  struktura plików, 122
Angular CLI
  instalacja, 61
animacja, 38 – 42
asynchroniczność, 198
atrybut, 22

B

backend, 262
baza danych MongoDB, 242
biblioteka jQuery, 7, 48, 286
Bootstrap, 49, 57, 286

C

CDN, Content Delivery Network, 9, 51
chwytanie elementów strony, 13
cookie sesyjne, 237
cookies, 232, 236
CRUD, 247, 256
CSS
  animowanie stylów, 38
  selektory, 10
  ustawienie właściwości, 21

D

dyrektywa
  ngClass, 150
  ngFor, 141, 144
  ngIf, 136
  ngStyle, 149
  ngSwitch, 145
dziedziczenie klas, 110

E

ekran small, 55
elementy formularza, 42
Express, 226
  mechanizm cookies, 236
  serwer HTTP, 226
  serwer Node.js, 260
  serwer WWW, 228, 230

F

format JSON, 193
formularze, 42 – 47, 174
  implementacja, 178
  walidacja, 182
  weryfikacja danych, 185
framework
  Angular, 59
  Bootstrap, 49, 57, 286
  Express, 226
frontend, 262, 270
funkcja, 92 – 98
  anonimowa, 99
  strzałkowa, 99
funkcje
  typ parametrów, 121
  zwrot wielu wartości, 102

I

inicjalizacja języka TypeScript, 71
instalacja
  Angular CLI, 61
  MongoDB, 243
  Node.js, 59
  TypeScript, 69, 70
  Visual Studio Code, 62
interfejsy, 118
interpolacja, 124

J

język TypeScript, 68
jQuery, 7, 48, 286
JSON, JavaScript Object Notation, 193

K

klasa, 23, 108
  container, 54
  container-fluid, 54
  row, 55
kolekcja, 247
komponent główny, 157
komponenty, 152
  budowa, 157
  cykl życia, 167
  kompozycja, 167
  odłączenie, 169
  przesyłanie danych, 160, 162
  zagnieżdżenie, 157
krotka, 90

L

lista
  numerowana, 18
  punktowana, 10
logowanie zdarzeń, 240

M

menedżer pakietów npm, 190
metoda
  .addClass(), 23
  .after(), 20
  .attr(), 12
  .before(), 20
  .blur(), 26
  .change(), 26
  .click(), 27, 28
  .css(), 12, 30
  .each(), 16, 24, 25
  .eq(), 26
  .fetch(), 262, 270
  .focus(), 26
  .hide(), 35, 37
  .html(), 16
  .keydown(), 30, 31
  .keypress(), 30, 31
  .mouseout(), 27, 28
  .mouseover(), 27, 28
  .not(), 15
  .odd(), 14
  .prepend(), 19, 20
  .ready(), 8
  .resize(), 32
  .scroll(), 32
  .show(), 35, 37
  .slideDown(), 38
  .slideToggle(), 38
  .slideUp(), 38
  .stringify(), 196
  .text(), 17
  .toggle(), 35, 37
  .update(), 250
  .updateMany(), 252
metody
  formularza, 42
  modułu os, 190
  tablicowe, 87
  zdarzeń, 25
middleware, 239
moduł, 192
  fs, 205
  http, 203
  nodemon, 74
  path, 211
  url, 204
modyfikator dostępu
  private, 115
  protected, 116
MongoDB, 242
  instalacja serwera, 243
  kryteria, 249
  polecenia, 247, 253
  w Node.js, 256
MongoDB Compass, 245, 254

N

narzędzie
  MongoDB Compass, 245, 254
  npm, 190
  REPL, 188
  Windows Terminal, 245
  yarn, 191
Node.js, 187
  asynchroniczność, 198
  instalacja, 59
  moduły, 201
  MongoDB, 256
  serwer HTTP, 214, 221
  serwer WWW, 220, 224
  własny moduł, 212

O

obiekt, 103
  event, 33
  process, 189
  query, 233
obiekty w tablicy, 106
obietnica, 268

P

pakiet, 192
parametr
  opcjonalny, 97
  resztowy, 98
pętla for in, 106
PID procesu, 190
platforma Node.js, 187
plik
  cookie, 236, 238
  index.html, 280
  script.js, 281
  style.css, 282
pliki statyczne, 241
pobieranie zawartości elementu, 16
polecenie
  .save, 189
  .load, 189
  cd, 64
  ng g c, 157
  n g s, 170
  ng new, 65
  node, 72
  npm init, 69
  npm version, 188
  npx nodemon, 74

R

REPL, 188

S

sekwencja żądanie-odpowiedź, 239
selektor #, 12
selektory, 43
serwer
  HTTP, 214, 221, 226, 263
  WWW, 220, 224, 228, 230

silnik V8, 199
słowo kluczowe
    const, 76
    extends, 110
    new, 109
    public, 110
    return, 95
    this, 24, 104
StackBlitz, 63
stos wywołań, 199, 200

T

tablice, 85, 86, 89
typ MIME, 218
TypeScript, 68
    dedukcja typów, 77
    funkcja, 92 – 98
        anonimowa, 99
        strzałkowa, 99
    inicjalizacja, 71
    instalacja, 69, 70
    krotka, 90
    łączenie typów, 81
    mechanizm typowania, 75
    metody tablicowe, 87
    tablice, 85, 86, 89
    tryb nasłuchu, 73
    typ
        any, 80
        boolean, 79
        number, 78
        string, 77
    typy jako zbiory wartości, 83
    wyliczenie, 90
typowanie obiektu, 107

U

usługi
    działanie, 172
    implementacja, 170

V

Visual Studio Code
    instalacja, 62
    okno terminala, 64
    zgłoszenie błędu, 77

W

wiązanie
    danych, 124
    dwukierunkowe, 131
    dwustronne, 133
    właściwości, 126
    zdarzeń, 129
wstawianie elementu, 19
wtyczka Popper, 51
wygląd strony, 54
wyszukiwanie elementów, 10
wyszukiwarka pakietów, 191

Z

zagnieżdżenie komponentów, 157
zasięg zmiennych, 101
zdarzenia, 25, 174, 270
    formularza, 42
    interfejsu, 25
    klawiatury, 30
    myszki, 27
    okna przeglądarki, 32
zmienna, 101
znacznik
    <article>, 124

, 20
    <button>, 129
    <div>, 42
    <form>, 178
    <hr>, 18
    <img>, 136
    , 142
    <ng-template>, 140
    , 18
    <script>, 8
    <span>, 277
    , 10
znak
    dolara, 16
    dwukropka, 16
    ucieczki, 44

PROGRAM PARTNERSKI

GRUPY HELION

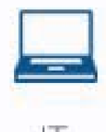
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

KOMPLEKSOWO SZKOLIMY
NOWOCZESNY BIZNES



IT

BIZNES

PROJEKTY

PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie
z dostępem do nowoczesnych narzędzi - wideokursów,
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL
WWW.HELIONSZKOLENIA.PL